Sistemas Informáticos
Curso 2003-2004

Proyecto de juego de Go.

Jesús Chavero García-Esteban
Juan Pablo Ramírez Pérez
Eduardo Sánchez Carballo

Dirigido por:
Prof. Oscar Dieste Tubío
Dpto. de Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Índice

Resumen

Este documento muestra cómo podría afrontarse computacionalmente el juego de Go mediante distintas métodos de inteligencia artificial. Se considera el estado actual de la cuestión y se evalúan las técnicas empleadas por distintos investigadores que han desarrollado aplicaciones para jugar al Go.

Se explica en detalle la arquitectura, tipos de datos y algoritmos para una implementación del juego del Go, así como posibles ampliaciones que mejorarían considerablemente el nivel de juego del sistema.

Abstract

This document shows a way to affront computer Go gaming by different artificial intelligence methods It considers the current state of the art and evaluates techniques used by different researchers who have developed applications of Go.

The architecture, data types and algorithms for an implementation of the game of Go are explained in detail, as well as possible add-ons which improve noticeably the game level of the system.

Palabras clave

Go, inteligencia artificial, árboles de juego, minimax, reconocimiento de patrones, descomposición de juegos, descomposición blanda

1. Introducción

El juego de Go constituye un gran desafío para la investigación en Inteligencia Artificial. Este proyecto muestra las enormes dificultades que supone el desarrollo de un sistema capaz de jugar bien a Go.

Puesto que el Go es un juego muy poco conocido en Occidente se explicarán las reglas básicas del juego, además de algunos de los conceptos más básicos del juego, antes de plantear aspectos referidos al Go en Inteligencia Artificial.

Para dar un marco adecuado al enfoque de Go de este proyecto se planteará la situación de las investigaciones actuales sobre la materia, partiendo de un resumen de la historia del juego de Go en computadores. Después se comentarán brevemente algunas de las técnicas empleadas por programas e investigadores, dando especial énfasis a las técnicas que se refieren a los aspectos más básicos del juego (evaluación de movimientos, vida y muerte, representación de conocimiento en Go y finales de juego).

Se prosigue planteando el problema en términos computacionales y explicando el porqué de su enorme dificultad, para a continuación describir las características del sistema planteado y los límites y objetivos concretos de este proyecto. Las características más destacables de nuestra aproximación son: que se trata de una aproximación software, que está centrada en el juego máquina contra jugador humano, que se le da mayor importancia a las mejoras de eficiencia sobre los algoritmos que a las mejoras de eficiencia logradas por aspectos de bajo nivel y que se busca lograr una implementación adaptable a la ejecución distribuida.

Los aspectos más relevantes que se explican en la aproximación a la solución de este proyecto incluyen la descripción de una arquitectura para un sistema de juego de Go flexible y que se pudiera adaptar para la ejecución distribuida, la descomposición de la partida en diferentes momentos de juego que requieren distinto tratamiento, el algoritmo de búsqueda en el espacio de estados empleado, la función de evaluación utilizada por dicho algoritmo, las estructuras de datos necesarias para dar soporte a los algoritmos y la metodología que se usa para descomponer el tablero en subjuegos.

La enorme amplitud de este proyecto impide llegar a una solución definitiva para implementar un sistema de juego de Go, así pues se detallan gran cantidad de ampliaciones posibles.

Se pretende con ellas aumentar la profundidad de la aproximación a la solución e incrementar la eficiencia del sistema.

Las ampliaciones principales incluyen el tratamiento distribuido de la aplicación y las técnicas para tratar las dificultades que surgen de paralelizar los cómputos en Go, el tratamiento de patrones que se emplea para diversos fines (inicio de partida, final de juego, división de tablero y escaleras) y el sistema de aprendizaje que se podría emplear para realizar diversas mejoras en el sistema (calcular los pesos de los parámetros de la función de aprendizaje, mejorar las podas de los árboles minimax y aprender patrones nuevos). Además se mencionan dos enfoques distintos del clásico basado en minimax que se podrían utilizar en un sistema de Go con diversos fines: redes neuronales y algoritmos genéticos.

Se complementa la información sobre el Go contenida en este documento con un apéndice que resume brevemente la historia de este juego y menciona las principales competiciones internacionales de Go.

2. Reglas básicas del Go

Existe un dicho sobre el Go que afirma que sus reglas se pueden aprender en cinco minutos, pero que hace falta toda una vida para poder dominar el juego.

2.1. Tablero y fichas del juego

El Go es un juego para dos jugadores que se juega en un tablero en el que se ha trazado una red de cuadrados de 19 x 19 líneas, aunque por motivos de claridad y sencillez los ejemplos de esta Sección están hechos en un tablero 7 x 7. Las fichas del juego se conocen con el nombre de piedras, y debe poder distinguirse entre piedras blancas y piedras negras. Las piedras son todas ellas del mismo valor, por lo que habitualmente son de la misma forma y tamaño.

2.2. Objetivo del juego

El objetivo del juego es cercar o rodear espacios libres de ocupación para constituir un territorio o suma de territorios mayor que el del contrario.

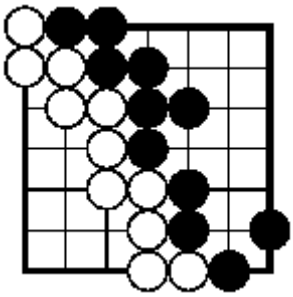

Fig. 1: Objetivo de Go

Vemos como se trata de conquistar territorio en el tablero, como se puede observar en este final de partida.

2.3. Mecánica de juego

Comienza jugando el jugador que juega con negras y alternativamente los jugadores van colocando una piedra cada jugador en alguna de las intersecciones entre líneas del tablero en la que

no haya ninguna otra piedra. Un jugador puede pasar si así lo desea. Cuando pasan los dos jugadores consecutivamente termina el juego

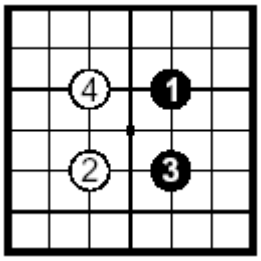

Fig. 2: Mecánica de juego de Go

2.4. Captura de piedras

Una piedra es capturada cuando todas las posiciones adyacentes en sentido vertical y horizontal (no en el diagonal) están ocupadas por piedras enemigas.

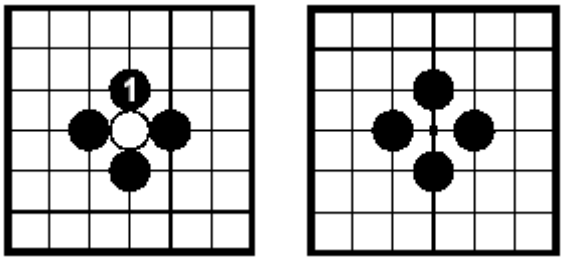

Fig. 3: Regla de captura de piedras

Varias piedras del mismo color que sean adyacentes horizontal o verticalmente constituyen un grupo indivisible que debe ser capturado simultáneamente.

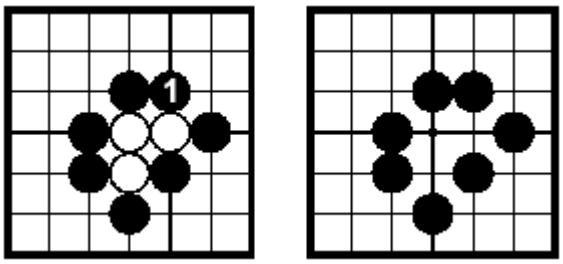

Fig. 4: Regla de captura de cadenas

Ninguna piedra puede colocarse en una posición que suponga una muerte automática para ella o un grupo de piedras de su mismo jugador, esto es no se puede poner una piedra en una posición

rodeada totalmente por piedras enemigas, salvo que al ocupar esa posición se complete una captura de manera inmediata.

Un jugador puede pasar de colocar piedra y el juego concluye cuando ambos jugadores pasan consecutivamente.

2.5. Regla de la captura mutua o ko

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 a la pieza que ha producido la captura sin cambiar la formación, se produce una serie de capturas denominada captura mutua o ko.

Para evitar que la serie de capturas sea interminable, el jugador que ha perdido la piedra debe esperar al menos una jugada para recuperar la posición. En concreto la regla del ko prohíbe la repetición de cualquier estado del tablero completo, aunque hayan pasado varias jugadas cuando se repite.

Veámoslo con un ejemplo:

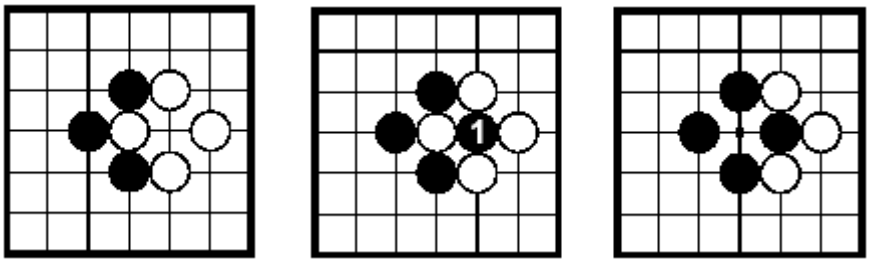

Fig. 5: Regla de ko: negras juegan y capturan

En este ejemplo observamos una situación en la que negras han capturado una piedra blanca. Ahora blancas podría volver a capturar de la siguiente manera:

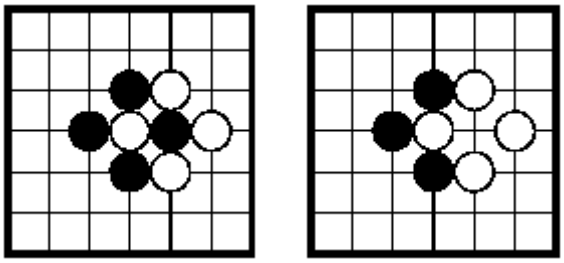

Fig. 6: Regla de ko: blancas juegan y se vuelve a la situación inicial

Pero como queda patente esto provoca una repetición de la jugada inicial, lo cual prohíbe la regla del ko. Si no se hubiese prohibido estaríamos ante una jugada infinita, si ahora volviese a capturar negras la piedra blanca.

Por ello la jugada anterior de blancas es ilegal bajo la regla del ko.

2.6. Situaciones de interés: Seki.

El seki es una configuración de cadenas de ambos colores que permite lo que se conoce como co-vida, es decir, que ambas cadenas están vivas en dependencia directa.

Veamos un ejemplo para dejarlo más claro:

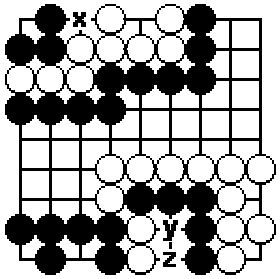

Fig. 7: Seki.

Aquí tenemos dos ejemplos de seki. Podemos observar como si cualquiera de los dos jugadores juega en las posiciones marcadas (x, y ó z) dicho jugador que intenta atacar la cadena del contrario pierde su cadena en la zona y le da el territorio a su rival.

El territorio intermedio no es ni de un jugador ni de otro, ya que queda rodeado por ambos.

2.7. Situaciones de interés: Escaleras.

Una escalera es una configuración en la que varias cadenas rodean otra de un solo color, de manera que ésta posee tan sólo dos libertades:

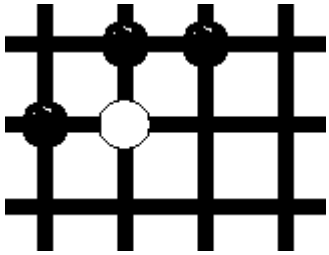

Fig. 8: Escalera básica.

Cuando tenemos esta jugada, la cadena que rodea (en el ejemplo las negras) pueden matar incondicionalmente la cadena interna siguiendo una forma de juego que recuerda a una escalera:

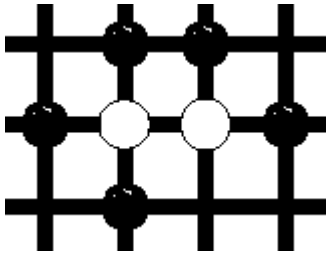

Fig. 9: Modo de jugar en escalera.

La idea es dejar una sola libertad a la cadena rodeada, de manera que la única posibilidad de aumentar sus libertades vuelva a dejarle en dos libertades a lo sumo.

Esta jugada tiene un problema, y es que para que podamos sacar ventaja de ella, y que la cadena interna esté verdaderamente muerta, no debe haber cadenas de ese color en el trazado de la escalera, ya que de lo contrario se podría capturar a gran parte del grupo de cadenas que estaban rodeando.

2.8. Conteo del resultado

Acabado el juego se procede a contar las posiciones que haya cercado cada jugador, contando únicamente aquellas completamente rodeadas, después se resta a esta cantidad el número de piedras que ha capturado su rival para obtener la puntuación total. Gana el jugador que haya obtenido una mayor puntuación.

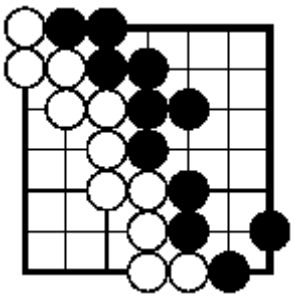

Fig. 10: Fin de juego y conteo

En este juego de ejemplo, suponiendo que no haya habido ninguna captura, observamos como negras tiene una puntuación total de 15 mientras que blancas de 11, que son las posiciones de territorio encerradas por cadenas de cada color. Por lo que habrían ganado las negras.

Este es el método de conteo japonés, que es el más difundido. Existen otros métodos en diferentes asociaciones de Go en el mundo, como la americana o la china.

2.9. Niveles de juego en Go

El baremo internacionalmente aceptado en Go clasifica a los jugadores en tres categorías: aprendiz, maestro amateur y maestro profesional. Los aprendices se dividen según su nivel de juego en 20 rangos de aprendiz, llamados *kyu*, que van de 20 *kyu* (menor nivel) a 1 *kyu* (mayor nivel). Los maestros amateurs tienen 6 categorías conocidas como *dan* que van desde 1 *dan* (menor nivel) a 6 *dan* (mayor nivel). Por último los maestros profesionales se dividen en 9 categorías que reciben el nombre de *dan* como en el caso de los maestros amateur y que van desde 1 *dan* (menor nivel) a 9 *dan* (mayor nivel).

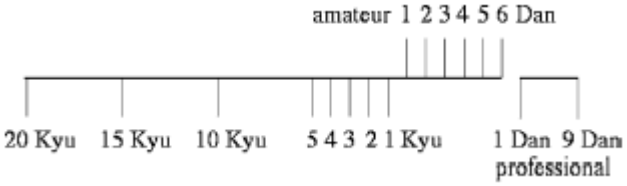

Fig. 11: Las categorías de los distintos niveles de juego en Go

3. Estado general del problema del Go

Esta panorámica acerca de la situación actual de las investigaciones sobre Go en inteligencia artificial comienza con un breve resumen de la historia de los programas de Go. Después se pasará a describir las aproximaciones que algunos investigadores y programas de Go han propuesto sobre aspectos concretos del juego de Go: en concreto las tendencias principales sobre evaluación de movimientos, vida y muerte, representación de conocimiento en Go y finales de juego. También se menciona algunas otras técnicas empleadas en Go: evaluación de metas y algoritmos de Monte Carlo.

3.1. Breve historia de los programas de Go

Los primeros programas de Go comenzaron a desarrollarse en la década de los 60. El primer documento sobre Go por ordenador, de D. Leftkowitz, fue publicado en 1963 y mencionaba la posibilidad de aplicar técnicas de aprendizaje a este juego.

La década de los 80 fue una etapa en la que los programas de Go empezaron a experimentar un gran impulso, gracias a la llegada de los ordenadores personales baratos y a los torneos patrocinados por organismos como la fundación Ing. En efecto, desde el año 1985, el Torneo Internacional de Go Computerizado (también llamada la Copa Ing por su patrocinador Ing Chang-Ki) supone uno de los mayores incentivos para la investigación sobre Go por Ordenador, ya que ofrece un premio de un millón de dólares a un programa de Go de nivel profesional.

Los primeros torneos de Go por ordenador estuvieron dominados por programas de origen taiwanés, tales como el Dragon de Hsu y Liu. Desde el 89 y hasta el 91 el programa Goliath de Mark Boom era el triunfador de todos los torneos, seguido de Go Intellect de Ken Chen y Handtalk y el más reciente Goemate de Chen Zixiang.

En los últimos años Go4++ de Michael Reiss, Many Faces of Go de David Fotlan y KCC Igo del equipo coreano KCC han obtenido victorias en los torneos más importantes. Hay en total unos diez programas de Go del nivel más alto conseguido, e incluyen Haruka de Ryuichi Kawa, Wulu de Lei Xiuyu, FunGo de Yong Goo, Star of Poland de Janusz Kraszek y Jimmy de Yan Shi-Jim. La mayor parte de estos programas se distribuyen de forma comercial.

Estos programas son seguidos de unos treinta programas de, digamos, gama media, que incluyen algunos escritos por investigadores universitarios o aficionados. Mención aparte en los programas de esta categoría merece el primer programa de Go de fuente abierta: GnuGo.

Respecto a enfrentamientos con seres humanos la mayor parte de los programas de Go tienen un nivel de juego de entre 15 kyu y 3 kyu, lo que implica que son muy inferiores a los jugadores competitivos de la escala amateur. (Ver Sección 2.9).

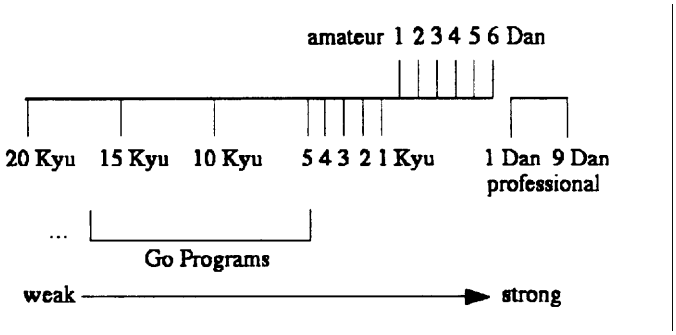

Fig. 12: Comparación entre el nivel de los Programas de Go y los jugadores humanos

Si hablamos de partidas concretas contra jugadores humanos y programas de Go, Goliath, en 1991, consiguió derrotar a tres jóvenes jugadores relativamente fuertes con una ventaja de diecisiete piedras y Handtalk logró la victoria en la partida de once piedras. Sin embargo es muy posible derrotar a todos los programas actuales con mucha mayor ventaja que once, especialmente si se los enfrenta con alguien que conozca su arquitectura y puntos débiles. Así por ejemplo Janice Kim derrotó a Handtalk con una desventaja de más de veinte piedras y Martin Müller ha derrotado a Many Faces of Go [19] dándole una ventaja de veintinueve piedras.

3.2. Evaluación de movimientos en Go

Existen dos aproximaciones a la evaluación de movimientos en Go, y ambas tienen sus ventajas y sus desventajas. El primer método es el de evaluación de posición y es semejante al empleado en otros juegos. Se considera un movimiento y una función estática computa una puntuación de todo el tablero estimando cada punto del mismo. En el caso del Go este tipo de función se basa en el cómputo de territorios.

La ventaja de este método es la precisión, ya que tiene en cuenta todo el tablero. Su gran desventaja es la gran complejidad en tiempo que lleva el recuento de territorios en un tablero tan amplio como el utilizado en el juego del Go de 19x19.

El segundo método consiste en la evaluación directa del movimiento. El valor relativo de un movimiento está estimado por el propio generador de movimientos que lo propone, basándose en heurísticas. Los movimientos propuestos por varios generadores acumulan un mayor valor total. La gran ventaja de este método es su velocidad. La gran desventaja es que es realmente difícil predecir las consecuencias que puede traer un determinado movimiento de una forma precisa.

Muchos programas, conocedores de las ventajas y desventajas de ambos métodos emplean un enfoque híbrido, bien a partir de evaluación de posición y primando mediante bonos y penalizaciones a ciertos movimientos basándose en heurísticas, bien a partir de evaluación de movimientos pero aplicando algunos pasos extra para dar mayor robustez, como, por ejemplo, con comprobaciones tácticas para evitar grandes errores.

3.3. Vida y muerte

El problema de Vida y Muerte en Go (o Tsume-Go) es fundamental en la potencia de un juego de Go por ordenador, efecto ya conocido desde los primeros momentos de la investigación sobre Go [1]. Esto es debido a la propia naturaleza del juego, ya que saber que un grupo de piedras está definitivamente muerto permite al jugador tener la iniciativa, al poder abandonar a tiempo las batallas perdidas para dedicarse a las que aún no se han decidido y, por otra parte, es vital conocer qué grupos siguen con vida (y de qué depende la vida de estos grupos) para no perder importantes grupos de piezas. En el siguiente ejemplo las piedras negras alcanzan vida incondicional con un solo movimiento, hagan lo que hagan blancas, de modo que para blancas es una mala zona para colocar y deberían luchar otras batallas para no perder la iniciativa:

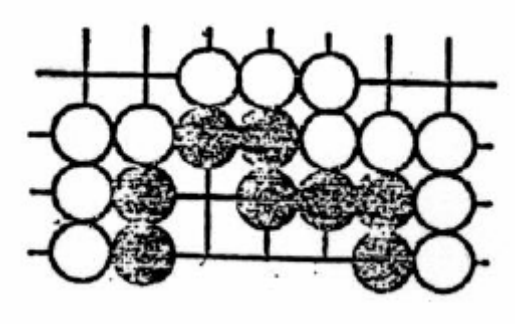

Fig. 13: Vida, muerte e iniciativa en Go

El mejor programa de resolución de problemas de vida y muerte en Go en la actualidad es GoTools, de Thomas Wolf, capaz de resolver problemas de Vida y Muerte de un nivel de 5 kyu. Este programa se basa en búsqueda alfa-beta, heurísticas de búsqueda y numerosos patrones para dirigir la búsqueda.

A continuación detallamos algunas ideas interesantes sobre el problema de Vida y Muerte en Go. El algoritmo más obvio para determinar si un bloque b es seguro es intentar jugar una secuencia de piedras en las intersecciones de las libertades del bloque. Si no existe ninguna secuencia legal de piedras que capturen al bloque, entonces éste es seguro. Ahora bien, este algoritmo resulta computacionalmente muy costoso. Además, no nos proporciona conocimiento útil acerca de qué posiciones resultan críticas para determinar la seguridad de un bloque. Esta búsqueda pura se puede refinar distinguiendo entre libertades interiores y libertades exteriores.

En el problema de la seguridad de las piezas resulta enormemente interesante el considerar conjuntos de bloques que se apoyan mutuamente unos a otros resultando en una estructura incondicionalmente viva. De este modo, mediante un algoritmo basado en detectar el apoyo entre las piezas, se resuelve el problema de la seguridad. Además, el algoritmo de apoyo necesita encontrar las intersecciones entre los bloques que se dan mutuo apoyo. Esta intersección es muy interesante para lograr un juego inteligente, ya que implica a los puntos que unen ambos bloques, esto es, las posiciones en las que el jugador que pretende mantener la seguridad no debe jugar. Por ejemplo, en la siguiente figura el punto C une los bloques de blancas y es fundamental para que blancas se mantengan con vida:

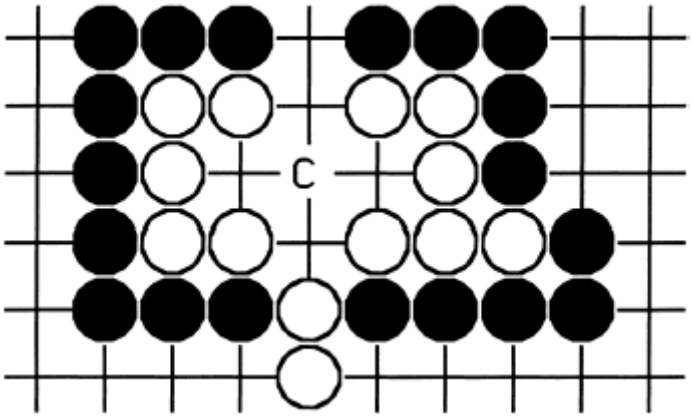

Fig. 14: Los puntos de unión de bloques de piedras y el problema de vida en Go

Por último, un análisis estático sobre Vida y Muerte en Go se puede basar en el análisis de las estructuras que dan seguridad a los bloques en Go: los llamados ojos [7]. Una posición vacía completamente rodeada por piezas de un jugador es un ojo. Un ojo es falso si está en el borde del tablero y cualquier diagonal adyacente al mismo está ocupada por una piedra oponente viva, o bien, si no está en el borde del tablero, si dos o más diagonales adyacentes están ocupadas por piedras adversarias vivas. Un ojo es parcial si puede ser cambiado por un movimiento del defensor en ojo real, y por un movimiento del adversario en ojo falso. Un ojo es real si el movimiento del oponente no puede cambiarlo en un ojo falso (aunque pueda ser capturado). Dos ojos reales unidos son siempre una estructura segura e incondicionalmente viva. El análisis de Vida y Muerte en Go pasa entonces por encontrar aquellas posiciones que posibilitan (o imposibilitan) la creación de ojos reales. Es este enfoque el que emplean los programas que analizan vida y muerte a partir de patrones. En el siguiente ejemplo si blancas juegan en A forman una estructura de dos ojos reales vivos incondicionalmente, mientras que si son las negras las que juegan en A amenazan la estructura de blancas al impedir la situación de vida incondicional:

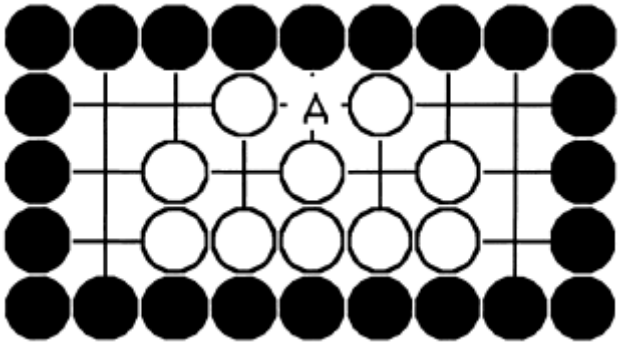

Fig. 15: Ojos y vida y muerte

3.4. Evaluación de metas en Go

Una de las conclusiones más claras de la investigación sobre Go es que los métodos basados en recorrer el espacio de los movimientos posibles por algoritmos tipo alfa-beta no son suficientes, por sí mismos al menos, para resolver el problema. Ante este hecho muchos investigadores se han planteado varias alternativas.

Una de estas alternativas consiste en modelar las metas de los jugadores y sus estrategias para lograr estas metas. Esta aproximación parte del hecho de que el espacio de metas posibles es mucho menor que el de movimientos posibles, lo que simplifica enormemente las búsquedas. En particular, el planteamiento exacto es el de redes jerárquicas de tareas, ya que la aplicación directa de técnicas de planificación no había obtenido resultados más que a un alto nivel estratégico.

La idea consiste a grandes rasgos en lo siguiente: Planteemos una meta, en el caso del Go el programa GOBI empleaba dos metas distintas, capturar y salvar grupos de piezas, dando énfasis al problema de Vida y Muerte. La meta salvar grupo necesitaría de tres esquemas, esto es, submetas que hay que conseguir para lograr la meta principal, que son encontrar ojos, hacer que el grupo escape y contraatacar. Como se puede observar la ramificación de la búsqueda que surge de este tipo de árbol es mucho menor que la de un árbol de juego de Go.

La principal desventaja de este tipo de aproximación es la necesidad de añadir una ingente cantidad de conocimiento al sistema, así como la necesidad de codificar las estrategias mediante descomposiciones de metas. A pesar de ello, parece bastante claro que un buen programa de Go se puede beneficiar en gran medida de la aplicación de un enfoque híbrido que integre ambos aspectos: búsquedas basadas en movimientos y búsquedas basadas en metas.

3.5. Representación de conocimiento y patrones en Go

Existen dos formas principales de representar el conocimiento en el juego de Go. Una forma sencilla es codificar este conocimiento mediante patrones, la otra manera parte de la definición y reconocimiento de una jerarquía de estructuras. Casi todos los juegos de Go contienen bases de datos de patrones y mecanismos de comprobación de patrones. La mayor parte de estos patrones se han introducido a mano y las bases de datos abarcan varios millares de ellos, aunque en los últimos tiempos se han desarrollado métodos para extraer patrones de partidas profesionales.

Para evitar el alto coste computacional que supone tratar de emparejar cada patrón con cada posición del tablero se emplean técnicas de filtrado basadas en tablas hash para reducir el número de patrones candidatos con los que hay que comparar la posición actual. Además hay una optimización clave en los patrones: cuando hay una situación de coincidencia con un patrón, esta se suele mantener a lo largo del juego. Aprovechar esta propiedad permite optimizar en gran medida las búsquedas de patrones, manteniendo aquellas que han surgido anteriormente.

Se pueden introducir elementos con un nivel de abstracción más alto que el representado por los patrones. Para tener el soporte adecuado para estos nuevos elementos es preciso definir previamente estructuras más complejas que representan el conocimiento contenido en el propio tablero. En primer lugar, la estructura básica en Go es el bloque, esto es, un conjunto de piedras conectadas entre sí.

Además podemos hablar de conexiones (o lugares donde se establece una unión entre bloques) y conexiones potenciales (o posiciones donde se puede establecer una conexión a través del juego). Por otra parte los divisores o barreras son bloques que detienen las conexiones del rival. Son estructuras mucho más débiles que las conexiones.

Siguiendo con esta formación de estructuras, las cadenas o grupos son conjuntos de bloques unidos por conexiones independientes. Hay que tener en cuenta a la hora de definir estructuras que la cohesión en Go no es transitiva (que si A está conectado con B y B con C, no necesariamente se deduce que A está conectado con C). Para definir estructuras se emplean métodos heurísticos que deciden qué subconjuntos de conexiones son más importantes [12]. En el siguiente ejemplo se muestran diferentes patrones de cohesión. C y D están conectados, así como D y E y E y F, aunque con menor cohesión, sin embargo no se puede inferir que C y F estén conectadas:

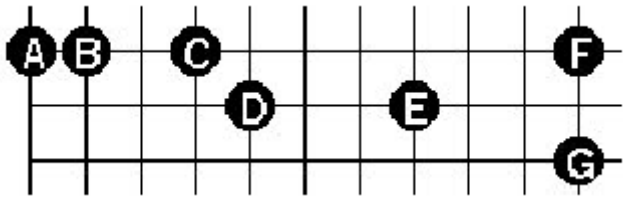

Fig. 16: Patrones de conexión que muestran que la conectividad no es transitiva

Por encima de las estructuras de cadena están las regiones, áreas, zonas, territorios o *moyo*. Este tipo de estructura se refiere a los territorios rodeados y su identificación resulta de vital importancia, ya que, al fin y al cabo, el objetivo principal es rodear la mayor extensión de territorio

posible. La identificación de territorios se puede realizar detectando zonas contiguas de alta influencia o encontrando fronteras que los definan como bloques o divisores. Un tercer método define los territorios a partir de la cohesión: un punto es territorio si ninguna piedra oponente colocada allí puede conectarse con otras piedras vivas del adversario, ni permanecer viva por sí misma..

Por último las mayores estructuras de piedras unidas o relacionadas son los grupos, ejércitos, unidades o dragones. Estas estructuras se pueden contemplar como regiones contiguas de cierta influencia, aunque sea mínima. Se reconocen mediante un proceso iterativo de crecimiento y decrecimiento, mediante otras medidas de distancia o mediante el uso de conexiones y divisores potenciales. Los grupos se constituyen en las principales unidades de ataque y defensa. Su fuerza relativa determina sus posibilidades de sobrevivir o ser capturados, si pueden dar apoyo a otros grupos cercanos o si pueden servir de base para atacar a grupos rivales. Las medidas de la fuerza de un grupo son su cohesión interna y su cohesión con otros grupos, las libertades de sus bloques y la presencia de ojos y potencial para construir ojos en su interior.

La mayor parte de los programas actuales emplean un intensivo análisis estático, apoyado en análisis de metas para determinar la seguridad de los conjuntos de piedras. Sin embargo todavía no se ha conseguido una manera eficiente de reconocer todas estas estructuras y las implementaciones actuales, aunque razonables, carecen de alguna de las sutilezas del juego. Así, tanto conexiones como regiones pueden no ser robustas debido a la presencia de amenazas dobles realmente difíciles de detectar. Los jugadores humanos son mucho más flexibles a la hora de establecer agrupaciones de piedras en sus análisis y emplearlas en sus tácticas.

En cuanto a la introducción de conocimiento especializado en Go hay que decir que resulta muy difícil encontrar reglas adecuadas, ya que no hay regla en Go que no tenga excepciones; sencillamente no hay reglas absolutas. Además, se debe hacer notar que el arduo trabajo que han empleado muchos ingenieros de conocimiento ha resultado mucho menos fructífero de lo esperado e, incluso, en algunas ocasiones, bastante inútil. Esto dificulta extremadamente la formación de bases de conocimiento, al menos las de tipo tradicional. Asimismo, la interacción entre reglas de Go produce a veces consecuencias impredecibles y, a menudo, malos resultados.

Una posible solución consiste en modelar el Go mediante aprendizaje de conocimiento de tipo declarativo [4], ya que parte del conocimiento en Go puede ser formalizado. Parte de este conocimiento se obtiene mediante el denominado análisis retrógrado, que consiste en partir de posiciones finales de juego y retroceder, reconstruyendo la cadena de posiciones que han conducido a

ella y asociarlas con un valor, que puede ser ganar, perder o empatar. Este método ha dado excelentes resultados en la fabricación de bases de datos de finales. Partiendo de este análisis se pueden formar algunas reglas tácticas, pero sólo se mantienen como relevantes aquellas que se asocian a condiciones de victoria, evitando así la gran cantidad de excepciones que habría que generar.

3.6. Finales de juego

Cuando se está llegando al final de una partida de Go, la situación se puede contemplar como una serie de luchas locales. En numerosas ocasiones, maximizar el resultado de una de estas luchas locales implica perder la iniciativa en otras, lo que puede significar perder puntos importantes para la victoria global, que es realmente el objetivo. Una exploración exhaustiva del tablero es inviable por la complejidad del problema.

Las modernas soluciones al problema del final del juego en Go se basan en la teoría matemática de juegos y en técnicas de búsqueda por descomposición en subjuegos [20, 21]. Dicho de otra forma, puesto que cuando se aproxima el final de un juego de Go existen múltiples luchas locales, la solución pasa por descomponer el tablero en estas luchas, analizarlas por separado y pelear por aquellas que supongan mayor puntuación. Este planteamiento no es exclusivo de los programas de ordenador, sino que los propios seres humanos lo emplean.

Un posible algoritmo de final de juego realizaría los siguientes pasos:

1. Dividir el tablero: encontrar bloques seguros, territorios seguros y zonas de fin de juego. Esta fase parte de algoritmos de tipo Vida y Muerte como los anteriormente expuestos.
2. Generar árboles de juego locales en cada zona de fin de juego. Hay que tener en cuenta que, llegados a este punto, los árboles generados pueden contener varios movimientos consecutivos del mismo jugador, ya que el otro jugador puede pasar a otra de las batallas locales.
3. Evaluar las posiciones terminales locales, esto es, aquellas posiciones que no permiten un buen movimiento porque todos los puntos queden ocupados o porque constituyan un territorio. Se calcula la puntuación local por cada posición terminal.
4. Transformar los árboles locales de juego en juegos matemáticos y simplificarlos.

5. Calcular el juego como suma de juegos locales. El juego total es la suma de los valores de territorio más todos los juegos locales.
6. Buscar un movimiento óptimo en el juego suma y jugarlo. Dado un juego suma y un jugador es fácil computar el valor minimax. Para encontrar el movimiento óptimo se prueban todos los movimientos y el óptimo es aquel que preserva el valor minimax.

Sobre este esquema se añaden refinamientos para obtener mejores resultados, tales como límites para evitar situaciones de *ko*, extensiones heurísticas y aplicando definiciones de seguridad más relajadas.

3.7. Go Monte Carlo

Durante la investigación sobre el Go se han ido aplicando buena parte de las técnicas de Inteligencia Artificial, tanto de tipo simbólico como de tipo no simbólico, desde búsquedas en espacios de soluciones hasta redes neuronales y algoritmos genéticos. Entre los enfoques más peculiares nos llamó poderosamente la atención el del programa Gobbler [6].

Gobbler es un programa de Go limitado a un tablero de 9x9, que consigue relativamente buenos resultados mediante la aplicación de una técnica sencilla: los algoritmos de tipo Monte Carlo. Si tan costosa es la evaluación y búsqueda de soluciones, la propuesta de este programa es hacer elecciones aleatorias de movimiento, llevándolas hasta el final, hasta terminar la partida, la evaluación del movimiento es, simplemente, el recuento del resultado del juego. Se hacen tantos juegos como se pueda o sea necesario y el mejor movimiento será aquel que consigue un mejor resultado final.

4. Descripción del problema

El problema, evidentemente, es realizar un sistema computacional que juegue a Go de manera automática. El juego de Go, sus reglas y algunos de sus conceptos principales quedan descritos y explicados en detalle en la Sección 2 de este documento, de modo que esta sección se centrará en los aspectos computacionales del problema.

Para explicar las metas y limitaciones de este proyecto se expondrá la problemática que hace del Go un juego tan difícil de afrontar computacionalmente. A continuación se referirán y delimitarán los objetivos y límites del proyecto.

4.1. Los grandes problemas que plantea el Go

El Go es el segundo juego de tablero en términos de esfuerzo de investigación, por detrás del ajedrez. Ahora bien, a pesar de que los programas de ajedrez pueden competir de forma digna con los maestros humanos, los programas de Go pueden ser derrotados con relativa facilidad por jugadores de nivel moderado. Es justo decir, sin embargo, que los programas de Go han experimentado un considerable avance en los últimos diez a quince años. Examinemos las causas de la debilidad de los juegos de Go por ordenador.

En primer lugar, la potencia bruta de cálculo y las técnicas de búsqueda alfa-beta y minimax por sí mismas no bastan para lograr un buen nivel de juego en Go. Mientras que el factor de ramificación del ajedrez, con sus 64 casillas, es de 35, el Go, con sus 361 casillas permite una media de 250 movimientos. Es decir, un ordenador con potencia similar a la de Deep Blue (capaz de analizar 200 millones de posiciones de una partida de ajedrez en un segundo) tardaría año y medio en procesar un único movimiento en Go.

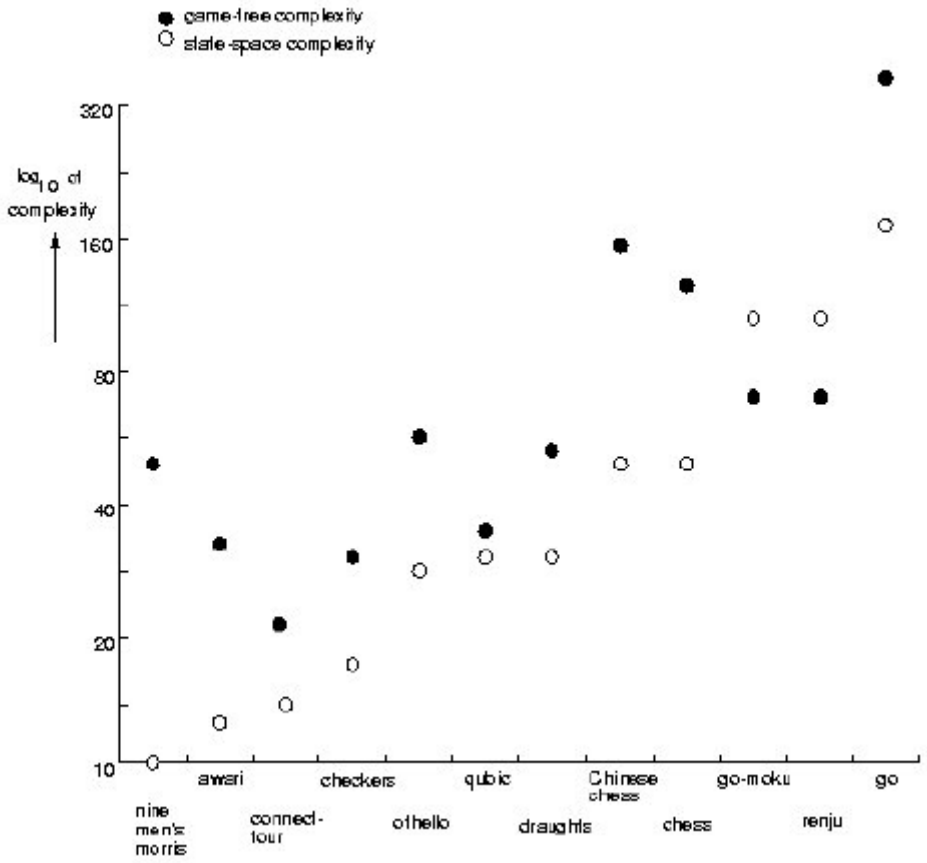

Fig. 17: Comparación entre las complejidades computacionales de varios juegos

El problema no acaba aquí. Supongamos que la potencia de cálculo, que al fin y al cabo prosigue su progresión ascendente, aumentara lo suficiente y llegara al nivel necesario. La cuestión no está en que los programas sean incapaces de evaluar 200 millones de movimientos, la cuestión está en que no son capaces de evaluar con precisión un único movimiento. Así pues el problema no sólo está en los enormes espacios de búsqueda del Go, sino que una de las mayores dificultades está en conseguir evaluar adecuadamente una única posición en el tablero.

Esta dificultad es consustancial a la propia naturaleza del Go. Mientras que en juegos como ajedrez la evaluación de una posición del tablero, descrita en términos humanos, dependería de las piezas del tablero, el valor de cada una de ellas y conceptos como tiempo, espacio, posición y tempo, en Go las piedras individualmente tienen el mismo valor y su valor real depende de sus relaciones con las piezas vecinas. Además de los conceptos mencionados para ajedrez, en Go tienen gran valor otros tales como "Densidad", "Fuerza", "Potencial", "Agilidad", "Peso", "Intercambio" y "Ambivalencia", esto es, términos que conllevan relaciones entre piezas y grupos de piezas, lo cual implica que cada evaluación debe incluir búsquedas tácticas auxiliares. Estas búsquedas no sólo

dificultan extremadamente la implementación de una función de evaluación adecuada, sino que penalizan de manera decisiva el tiempo de cálculo que requeriría una función de evaluación precisa.

Comparemos el nivel de los jugadores humanos con el de los actuales programas de Go, los mejores programas de Go se mueven en rangos que varían entre 15 kyu y 3 kyu, esto es que no pueden, ni de lejos, competir con jugadores de nivel competitivo ni siquiera a escala amateur. (Ver Sección 2.9)

Ante este desolador panorama, el Go se ha constituido en uno de los mayores desafíos para la investigación en Inteligencia Artificial sobre juegos. Además, del estudio del Go se desprenden numerosos subproblemas (Vida y Muerte, finales de juego,...) de gran interés en sí mismos.

4.2. Objetivos de nuestra aproximación

Puesto que el juego de Go es extremadamente complejo en términos computacionales y, hasta la fecha, no existe un programa que juegue a un nivel aceptable; el planteamiento no podía ser la realización de una aplicación que jugara al nivel de los mejores juegos de Go actuales, dada la naturaleza de este proyecto y su duración en el tiempo. Nuestros cálculos más optimitas nos llevaban a pensar que aprovechar bien las ideas contenidas en los documentos, que ya estábamos analizando, hubiera prolongado varios años la duración de este proyecto. Esto obligó desde un principio a plantear los requisitos y objetivos de este proyecto en términos mucho más modestos y realistas.

Sin embargo lo que sí consideramos más interesante fue orientar la aplicación al juego contra contrincantes humanos, porque el juego máquina contra máquina es menos desafiante. No se ha conseguido un programa que juegue a gran nivel en Go, que pueda competir contra los maestros humanos, así pues encontramos de mayor interés investigar los aspectos que hacen tan débil el juego de los programas de Go.

Otras características que consideramos deseables de nuestro sistema y que caracterizan el enfoque de la aproximación al Go computacional descrito en este documento son las siguientes:

Sistema software, no hardware. No es nuestra intención describir una arquitectura máquina especializada en resolver problemas de Go. El sistema debe ser independiente de la arquitectura

hardware, aunque sí consideramos como objetivo encontrar una arquitectura software que pudiera beneficiarse de los recursos de una red de computadores, pero todos ellos computadores convencionales y no especializados.

Desde un primer momento decidimos orientar nuestra investigación a encontrar soluciones que pudieran adaptarse para una ejecución distribuida, para aprovechar la mayor potencia de cálculo que proporcionan las redes de computadores. A este enfoque responde la ampliación mediante RMI propuesta en el punto 8.

Énfasis en la eficiencia de los algoritmos, no en mejoras de bajo nivel. No buscamos trucos que mejoren la eficiencia de la implementación, sino algoritmos eficientes en sí mismos. Las características de independencia de la plataforma software y el hecho de que ignoremos la eficiencia de los aspectos de bajo nivel de implementación nos hizo decidirnos por programar en el lenguaje Java.

Nos concentramos en investigar sobre técnicas y métodos empleados por diversos investigadores, atendiendo especialmente, como es obvio, a investigaciones recientes. Las soluciones propuestas por los distintos investigadores para el juego de Go implican un gran número y variedad de técnicas de inteligencia artificial. Se podría afirmar que casi todas las técnicas de la inteligencia artificial han sido aplicadas en uno u otro momento al juego de Go. Gran parte de este trabajo consistió en ponderar cuáles nos parecen más apropiadas y cuáles no, atendiendo al enfoque de nuestro sistema.

Aunque nos centramos en métodos de inteligencia artificial simbólica decidimos no rechazar las técnicas de inteligencia artificial no simbólica, como se puede comprobar en la ampliación mediante redes neuronales que proponemos en la Sección 6.4.

En cuanto a aspectos puramente de reglas de juego, decidimos ignorar las situaciones de *ko* y *seki* (ver Sección 2) porque provocan enormes dificultades. Contemplar la situación de *ko* fuerza a llevar un historial total de las posiciones del tablero completo, mientras que *seki* requiere aumentar extraordinariamente la complejidad de las búsquedas minimax.

Cuando tuvimos suficiente información y habíamos analizado ya un número suficiente de técnicas distintas aplicadas a Go, pudimos plantear una serie de metas y objetivos mucho más

concretos. Entre estas metas la primera era encontrar una arquitectura que pudiera incluir las técnicas que nosotros encontramos mejores para nuestro estudio. Esta arquitectura debía ser suficientemente flexible como para poder ser ampliada usando otras técnicas e ideas, ya que éramos conscientes de que buena parte de nuestro proyecto debía modificarse sobre la marcha según las dificultades fueran apareciendo, así como de que muchos aspectos quedarían como ampliaciones.

Otro aspecto clave era encontrar las estructuras de datos que permitieran dar soporte a los algoritmos de juego, atendiendo a consideraciones tanto de espacio como de coste temporal de los algoritmos que tenían que recorrerlas.

Entre nuestras primeras metas se encontraba hallar un algoritmo de minimax sencillo y flexible, así como describir una función de evaluación adecuada al problema. Ya que, aunque los enfoques clásicos de inteligencia artificial han probado no ser eficaces por sí mismos en el juego de Go, suponen un importante punto de partida para complicarlo aplicando metodologías de descomposición en subjuegos.

Encontrar el algoritmo de descomposición más idóneo era otro de los grandes objetivos de este proyecto, ya que enfrentarse a la enorme complejidad que supone evaluar un movimiento en el tablero completo de Go es inabordable computacionalmente para los algoritmos de búsqueda en un espacio de soluciones.

Dado que era imposible dar una solución definitiva al problema, decidimos que las posibles expansiones que se podrían hacer a partir de este trabajo serían descritas con suficiente grado de detalle como para poder retomarse en otros proyectos.

5. Aproximación a la solución

Nuestra aproximación a la solución del problema incluye varios aspectos fundamentales. Entre estas cuestiones está la descripción de la arquitectura de la aplicación, la división de una partida en varios momentos de juego, el algoritmo de minimax básico empleado, la función de evaluación que el algoritmo utiliza, las estructuras de datos necesarias para modelar los conceptos de cohesión de piezas en Go y los algoritmos de descomposición en subjuegos.

5.1. Arquitectura de la aplicación

La arquitectura del sistema de juego se puede describir a grandes rasgos a modo de un módulo complejo de juego que actúa como una caja negra que recibe entradas y genera salidas a varias interfaces.

El más importante de ellos, va a ser el controlador de partida. Este módulo se comportará como un observador de la partida en curso, y se limitará a transmitir los datos a quien se los pida.

Estos dos módulos forman la arquitectura básica del sistema, a ellos están conectados diferentes módulos de muy diversos propósitos, aunque, principalmente, serán interfaces externas. Entre los módulos que no estarían incluidos en este grupo estarían los de explicación y consejo, que interactuando con ambos módulos realizarían sus respectivas funciones.

El esquema de conexión básico sería similar al siguiente:

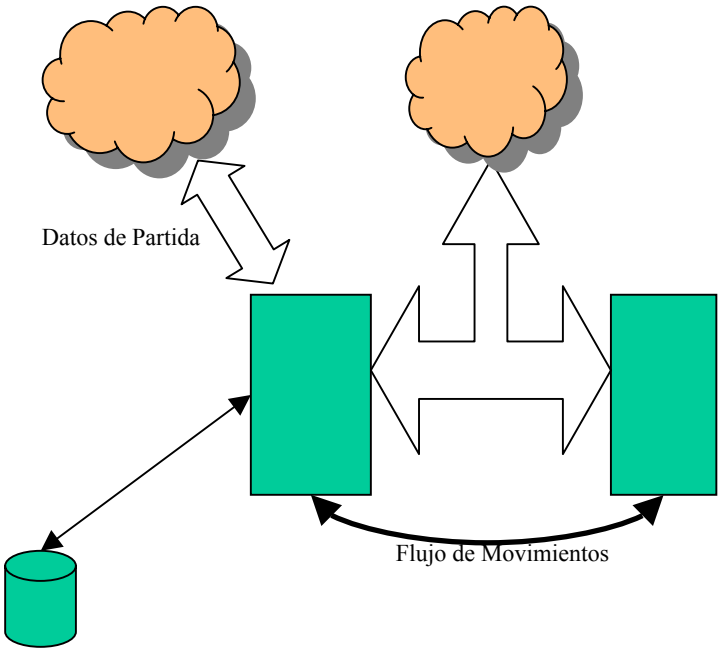

Fig. 18: Arquitectura del sistema

Podríamos expresar las interacciones entre los subsistemas principales mediante el esquema expuesto a continuación:

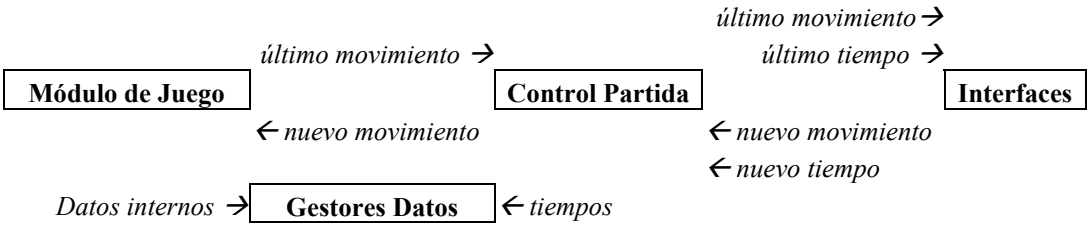

Fig. 19: Esquema de interacciones entre subsistemas principales

El módulo de juego incluye subsistemas dedicados a los aspectos más básicos del juego:

Explorador del espacio de estados: esta parte se diseña e implementa de acuerdo con los principios más básicos de inteligencia artificial. Esta basado en minimax y tiene implementada una función de evaluación. El algoritmo minimax no se puede aplicar directamente al tablero completo de juego, de modo que actuará únicamente sobre los subjuegos en que se descompone el tablero.

Divisor en zonas: el buen funcionamiento de esta parte es crítico para el rendimiento de las demás, ya que delimita las zonas sobre las que operan otros módulos. Reconoce de forma sencilla unos pocos patrones que separan y unifican zonas del tablero.

La base de datos del sistema de control de juego incluye las aperturas fundamentales de Go. *joseki* y *fuseki* (ver Sección 6.2 en el punto correspondiente a aperturas) y algunos finales de juego.

5.2. Momento de juego

La complejidad de Go impide dar un tratamiento uniforme a todas las situaciones de juego. Es preciso reconocer distintos momentos de partida tanto globales como locales, con diferentes enfoques estratégicos y tácticos, de manera similar a como actuaría un jugador humano.

Nuestra intención desde un principio fue emplear una descomposición temporal del juego basada en la forma de comportarse que se deseaba para el sistema, teniendo en cuenta especialmente consideraciones de complejidad temporal. Para describir las fases de juego emplearemos una terminología similar a la que se emplearía de forma intuitiva para describir el juego de un humano. De acuerdo con esta idea, los momentos de partida en que se descompone el juego de acuerdo con la aproximación descrita en este documento son los siguientes:

Comienzo de juego: se caracteriza por la utilización de aperturas y la toma de posiciones iniciales de los jugadores, al igual que en otros juegos de estrategia como ajedrez. Tradicionalmente en Go se emplea una de las dos aperturas clásicas: *joseki* y *fuseki* (ver Sección 6.2 en el punto correspondiente a aperturas). Atendiendo a este enfoque, el sistema simplemente reconocerá la utilización de una u otra apertura por parte del jugador humano y actuará siguiendo la apertura hasta que se terminen los movimientos correspondientes a la apertura reconocida, o bien la situación del tablero haya variado tanto sobre la situación descrita por la apertura que sea imposible continuar con ella.

Aunque esta forma de actuar da poca o ninguna flexibilidad al comienzo de partida del sistema, tiene unas ventajas muy importantes. No es posible aplicar minimax a una situación de juego en la que los movimientos posibles son prácticamente todo el tablero. La descomposición en subjuegos tampoco soluciona el problema, ya que las zonas serían casi todo el tablero.

La gran ventaja de emplear las aperturas clásicas es que contemplan aspectos primordiales de posicionamiento básico sobre el tablero y puntos estratégicos del mismo, así como el uso de una cierta estrategia desde el principio. Por supuesto un jugador humano que conozca el juego y sus aspectos teóricos conocerá perfectamente las dos aperturas clásicas y sabrá contrarrestarlas y aprovechar sus defectos, pero con una base de aperturas más amplia este defecto se atenúa. Aumentar la biblioteca de aperturas no supone incrementar la dificultad del sistema.

Desde un punto de vista puramente computacional, partir de una situación de apertura para aplicar juego medio supone grandes ventajas, ya que ya hay (o, en teoría, debería haber) suficientes piedras colocadas para que la descomposición del tablero sea eficiente y útil y se podrá aplicar la búsqueda en el espacio de soluciones a los subjuegos resultantes.

Por la propia naturaleza de esta aproximación al inicio de juego en Go se dejará de estar en una situación de comienzo de juego cuando no se pueda continuar con alguna de las variantes de una apertura incluidas en la base de datos correspondiente, porque la situación difiera sustancialmente de la apertura.

Otro problema obvio que supone este planteamiento es que un jugador humano que actúe ignorando las aperturas, ya sea por desconocimiento de las mismas, o por ser un jugador que conozca la arquitectura del programa y sepa jugar sin utilizar las aperturas, forzará a pasar inmediatamente a la situación de juego medio. Comenzar demasiado pronto el juego medio provoca descomposiciones del tablero en zonas demasiado grandes y prácticamente vacías, de modo que los recorridos sobre el espacio de búsqueda serán costosos y poco útiles porque, al no poder evaluar suficientes movimientos, ni poder apoyarse en otras técnicas (como los patrones de vida muerte) el sistema produce movimientos sin tener información suficiente. Si esto ocurre el jugador humano podrá derrotar al ordenador sin grandes problemas.

Para paliar este defecto, se describirán ampliaciones basadas en aprendizaje (ver Sección 6.3), que deberían disminuir la debilidad del sistema ante este problema, ya que si un jugador humano fuerza a abandonar demasiado pronto la fase de inicio de juego, el sistema podrá tener conocimiento de otras partidas en que le forzaron a abandonar la fase de inicio y podrá considerar como perjudiciales algunos movimientos hechos en momentos anteriores.

Juego medio: se caracteriza por la utilización de un algoritmo para descomponer el tablero en subjuegos y la aplicación de minimax a los espacios de búsqueda resultantes. Obviamente es clave la elección de la función de evaluación más adecuada al sistema. Será la fase más larga de juego y aquella en la que tendrán mayor importancia las cuestiones de complejidad en espacio de memoria y en tiempo de computación.

Las complejidades de tiempo y espacio en memoria motivaron un planteamiento del sistema que pudiera implementarse en una red en la que hubiera varios procesadores. Al fin y al cabo otros juegos, típicamente el ajedrez, se han resuelto mediante sistemas con varios procesadores. Aunque no hubo tiempo de realizar una implementación completa del sistema que pudiera ejecutarse de forma diferida, en este documento se describe en detalle el uso de RMI para la implementación de la red del sistema (ver Sección 6.1).

A pesar de todas las mejoras que se empleen las soluciones a las que llegue el algoritmo minimax no podrán ser óptimas con total seguridad, en primer lugar porque la función de evaluación no podrá ser perfecta, ya que en un solo movimiento influyen demasiados factores complejos como para que se puedan evaluar de forma eficiente en tiempo y, en segundo lugar, por el elevado factor de ramificación que puede tener un árbol de juego que impide llegar a evaluar absolutamente todos los niveles del árbol de juego resultante, aunque estemos trabajando términos locales en zonas no muy amplias y aunque se esté aprovechando toda la potencia de una red grande de computadores..

Otro de los problemas del planteamiento de esta fase está en que por bueno que sea el algoritmo de descomposición en subjuegos es tremendamente ingenuo pensar que los subjuegos son disjuntos, aunque se deban tratar como si lo fueran por motivos de complejidad computacional. Los conflictos librados en una zona del tablero pueden repercutir enormemente sobre la situación global del juego. Un jugador humano de nivel suficiente tiene una visión no sólo local de cada zona de conflicto, sino que es consciente de cómo afecta al tablero completo y podría aprovecharse de esta ventaja para derrotar al sistema.

Para minimizar este problema decidimos incluir factores globales en la función de evaluación. Esto penaliza enormemente el tiempo de evaluación de cada movimiento, pero consideramos que una función de evaluación puramente local no proporciona suficiente información como para ser adecuada.

Además de considerar aspectos globales en la evaluación de los movimientos, tuvimos en cuenta las investigaciones sobre descomposición blanda[1]. Esta técnica considera la cuestión de la repercusión global de los subjuegos locales y da una aproximación basada en bosques para enfrentarse a ella. En la Sección 6.1 se consideran las repercusiones que tendría añadir descomposición blanda al sistema.

A diferencia de la división entre comienzo de juego y juego medio, saber si se está en una situación de final de juego resulta demasiado difuso a escala global desde el punto de vista computacional. Consideramos que el mejor planteamiento era pensar en finales de juego locales. Se abandona el juego medio en una zona, cuando esa zona responde a un patrón de final basado en vida o muerte incondicional, o el árbol minimax para esta zona alcanza nodos realmente terminales.

Final de juego: Esta fase se caracterizará por determinar qué grupos de piezas viven y qué grupos de piezas mueren, analizando después la repercusión que tiene esta situación de vida y muerte sobre la puntuación global de los jugadores. En la aproximación del problema descrita en este documento, se considera que los finales de juego se producen tan solo localmente.

Para un jugador humano de alto nivel, reconocer que la partida ha terminado es mucho más sencillo que para un computador. Llegado un momento el jugador humano comprende que hay zonas que están perdidas o ganadas y se centra en aquellas que están en conflicto. Cuando la posesión de todas las zonas queda clara, el juego consiste únicamente en buscar ganar puntos, reduciendo los del contrario o ampliando los propios. Si ya no puede encontrar mejoras para su puntuación el jugador humano pasa. En el momento en que ambos contendientes pasan se acaba el juego.

El problema de saber cuándo estamos en una situación de final de juego depende del problema de vida y muerte en Go. Cuando de alguna manera se sabe que un grupo de piezas está vivo o muerto de forma incondicional se ha producido una situación de final, en la que no se puede hacer otra cosa que maximizar los puntos propios o minimizar los del rival, la posesión de la zona ya se ha decidido. La cuestión es que el problema de vida y muerte en Go es tan complejo computacionalmente hablando como el propio juego. Si existiera un algoritmo eficiente y rápido para resolver el problema de vida y muerte, el Go estaría resuelto.

[1] Hemos considerado acertada la traducción *descomposición blanda* para el término inglés "*soft descomposition*", sobre otros intentos como descomposición flexible o descomposición parcial, por no ajustarse éstos últimos completamente a lo que queríamos dar a entender con este término. ***NdA.***

Las soluciones que hasta el momento han probado ser mejores para descubrir si un grupo de piezas está vivo o muerto pasan por reconocimiento de patrones y se explican en la Sección 6.2 de este documento, ya que por problemas de limitación de tiempo del proyecto no pudieron ser implementadas.

En cualquier caso la forma de proceder del sistema sería detectar un final por reconocimiento de patrones, esto es, detectar una situación de muerte incondicional o de vida incondicional de las piedras de uno de los dos jugadores en la zona, y tener en cuenta cuántos puntos se han ganado o perdido, para abandonar cuanto antes las zonas perdidas y evitar ceder la iniciativa al rival. La maximización de ganancias y minimización de pérdidas se realiza mediante el propio minimax.

Como ya se explicó en la Sección 4.2 no se han considerado las situaciones de *ko* y *seki* en este proyecto. Lógicamente las situaciones cíclicas en la partida pueden provocar graves problemas para saber si se está o no en un final local, así como en el nivel de juego del programa; pero la limitación en tiempo impedía que nos planteáramos soluciones para *ko* y / o *seki*. Este enfoque es bastante habitual en Go computacional porque las dificultades que provocan ambas situaciones aumentan en gran medida las complejidades en espacio y tiempo del problema.

5.3. Algoritmo de minimax

La técnica de minimax es clásica en inteligencia artificial y este proyecto incluye una gran variedad de técnicas de inteligencia artificial distintas y variadas, de ahí que no nos centráramos en encontrar un minimax enormemente eficiente, si no en que el minimax se pudiera adaptar al enfoque de nuestro problema y fuera suficientemente flexible como para ser modificado varias veces.

Por ello el algoritmo de minimax que implementamos se basa en la separación entre el propio algoritmo y la función de evaluación que emplea. Por otro lado el generador de movimientos posibles del minimax debía depender de la zona en la que se ejecuta el minimax de modo que también separamos el algoritmo minimax del generador de movimientos. Otra de las características necesarias del algoritmo de minimax era que debía estar limitado en profundidad. El algoritmo de minimax que resultó de estas consideraciones es un minimax bastante genérico. Los aspectos específicos de función de evaluación y generación de movimientos se comentan más adelante en esta misma sección:

```
public static Resultado busqueda(int pos, int prof, int maxprof,
boolean jugador, int use_t,
                                 int pass_t, JuegoInt ji) {
      Object[] hijos;
      Resultado rsuc=new Resultado();
      int i;
      int valor,pt;
      LinkedList mejorCamino=new LinkedList();

      pt=pass_t;
      if (prof>=maxprof) {
            System.out.println("Profundidad "+prof+" mayor que
"+maxprof);
            return (new Resultado(ji.parametro(pos,jugador),null));
      }
      else {
            hijos=ji.genmov(pos,jugador);
            if (hijos==null) {
                  System.out.println("Mala entrada, no funciona el
generador");
                  return (new
Resultado(ji.parametro(pos,jugador),null));
            }
            else {
                  for(i=0;i<Array.getLength(hijos);i++) {
                        rsuc=busqueda(i,prof+1,maxprof,!jugador,-
pass_t,-use_t,ji);
                        valor=rsuc.getValor();
                        if (valor>pass_t) {
                              pt=valor;
                              if (rsuc.getCamino()==null)
                                    mejorCamino=new LinkedList();
                              else
                                    mejorCamino=new
                              LinkedList(rsuc.getCamino());
                                    mejorCamino.addFirst(hijos[i]);
                              }
                              if (pt>=use_t) {
                                    return (new
Resultado(pt,mejorCamino));
                              }
                        }
                        return (new Resultado(pt,mejorCamino));
                  }
            }
      }

}
```

5.4. Función de evaluación

El comportamiento que deseábamos para la función de evaluación determinaba que fuera simple, flexible y no excesivamente costosa en tiempo. Por otra parte se requería que tuviera en cuenta aspectos globales de la partida, así como aspectos locales del subjuego sobre el que estuviera actuando.

Así pues consideramos que debía el resultado de la suma con pesos de los siguientes parámetros:

Piezas perdidas por el jugador humano y piezas perdidas por el computador: obviamente el comportamiento deseable del juego valoraría como positivo capturar piezas al rival y como negativo perderlas.

Las piedras capturadas constituyen un factor de importancia bastante grande y su peso sería relativamente elevado. Sin embargo no es el factor determinante de la evaluación de un movimiento del juego porque el objetivo de Go no es la captura de piezas, sino la mayor dominación de territorio al menor coste en piezas posible, de modo que, si sólo se tuviera en cuenta el número de piedras perdidas o capturadas, un jugador humano derrotaría con gran facilidad al sistema simplemente poniendo señuelos y sacrificando grupos de piedras para conseguir más territorio.

El cálculo de este factor resulta inmediato y basta con añadir dos variables que sirvan como contador, por otra parte el sistema de control de la partida necesita conocer las piedras capturadas por ambos bandos.

Piedras poseídas por ambos jugadores: este factor era necesario añadirlo porque en una situación muy inicial, a la que se llegue sin poder aplicar las aperturas, pudiera darse el caso de que al computador le resulte indiferente poner piedra o no, haciendo que el sistema pasara al comienzo de la partida, perdiendo por completo toda la iniciativa y dando ventaja de número al rival.

Es obviamente un factor poco determinante, que solo sirve para discernir entre situaciones muy similares. De hecho puede resultar engañoso, ya que pasar de colocar pieza en una zona local

puede permitir ganar la iniciativa en otra zona en la que se decidan más puntos o que esté en mayor peligro. Este hecho genera el concepto de temperatura de una zona (ver Sección 6.1).

Se calcula de forma inmediata a través del uso de dos variables que cuenten el número de piedras de un color y otro que hay sobre el tablero, o bien de la resta del número de piedras colocadas por un jugador (información relevante para el control de partida) y restando el número de bajas que ha sufrido.

Territorio controlado por un jugador y por otro: es el factor más importante del juego, ya que es el objetivo mismo del juego maximizar el territorio controlado por un jugador y minimizar el controlado por el otro jugador. El peso de este factor es, obviamente mucho más elevado que el de los otros factores.

Por desgracia es un factor costoso de calcular y además difícil de determinar algorítmicamente. De hecho es algo tan difícil de calcular que muchos programas de Go son incapaces de calcular la puntuación actual suponiendo que se declarase final de juego en un momento cualquiera.

Esto es debido a que solo se puede saber si un jugador controla un territorio o no, si se conoce la situación de vida o muerte de sus piedras en dicho territorio, así como la situación de vida o muerte de las piedras del rival. Como ya se verá en la discusión relativa a vida y muerte, el problema es tan complejo como el propio juego en sí mismo (ver Sección 6.2).

De modo que tratar constantemente de resolver el problema de vida muerte del grupo de piedras actual en cada una de las evaluaciones de cada uno de los posibles movimientos es totalmente inviable, así pues era necesario definir un parámetro que tuviera en cuenta el territorio que controla un jugador, aunque no fuera de forma completamente precisa.

Decidimos tomar un punto de vista optimista: asumir que todas las piedras están vivas incondicionalmente a efectos del cálculo de territorio. De esta forma, para el cálculo de este parámetro empleamos el siguiente método: consideramos para en el conteo de territorio únicamente las piezas vacías. Una zona en la que solamente hay piedras de un jugador pertenece a ese jugador, así el total del territorio es el número de espacios vacíos que quedan dentro del mismo. Si dentro de un territorio delimitado por piedras de un jugador hay piedras de otro consideraremos que las piedras

del otro jugador, así como el territorio interno que puedan delimitar pertenecen al otro jugador y aplicaremos este proceder de forma recursiva.

Como resulta evidente, si se tuviera información fiable relativa a vida y muerte, el valor del territorio sería mucho más cercano a la realidad y proporcionaría un conocimiento mucho más relevante acerca de la situación en la partida. Sin embargo, por el propio planteamiento de nuestro sistema en cuanto a momento de juego no consideramos que esto sea un problema, ya que si el sistema es capaz de detectar una situación de vida y muerte, los cálculos minimax tendrán menor importancia y el territorio se podrá calcular de forma precisa aplicando el algoritmo descrito (una vez retiradas las piedras muertas). Si el sistema no puede detectar una situación de vida y muerte debe proseguir con la evaluación minimax. Evidentemente el sistema carece en este punto de información suficiente sobre vida y muerte y no se puede considerar el efecto que tiene vida y muerte sobre el territorio.

Grado de cohesión de las piedras: es un aspecto de relativa importancia, ya que las piedras de un jugador tienden a ser menos vulnerables si están unidas. Aunque hay que tener en cuenta que es un aspecto secundario del juego, ya que son mucho más importante el territorio y las piedras capturadas que sí tienen relación directa con el final del juego.

En cuanto a su expresión en términos computacionales, el grado de cohesión es un aspecto un tanto difuso y, como se puede ver en la discusión sobre la división de espacios no es demasiado fácil de averiguar. Por ello sacrificamos precisión en la información contenida en este parámetro para obtener eficiencia. Como buscábamos un parámetro sencillo de calcular, decidimos que el grado de cohesión de las piedras se referiría únicamente a si las piedras están formando cadenas. Así pues el grado de cohesión de las piedras de un jugador es el número de piedras total dividido entre el número de cadenas.

Se puede calcular de forma muy rápida si las estructuras de datos definidas en el sistema incluyen las cadenas de piedras de los jugadores. Si es así podemos considerar que el cálculo del grado de cohesión es inmediato.

Determinar qué pesos debían tener cada uno de los factores es una cuestión compleja. La técnica habitual para dar pesos a los parámetros de una función de evaluación suele calcularse de forma experimental. Por motivos de limitación en tiempo no nos era posible realizar las pruebas

necesarias para afinar los pesos de la función de evaluación. El progresivo refinamiento de los pesos de los parámetros de la función de evaluación queda como ampliación de este proyecto (ver Sección 6.3), mientras que los pesos expuestos a continuación son una intuición a priori basada en nuestras consideraciones preliminares acerca de la importancia de los parámetros expuestos.

```
// La evaluación se basa en el territorio que ocupan las blancas, el
que
// ocupan negras, el numero de blancas, el numero de negras, las bajas
de
// negras y blancas (a lo largo de la partida) y el grado de cohesión
de
// blancas y negras
// Los pesos de los parámetros se proporcionan externamente
public float evalua(Superestructura superest, Tablero tablero, float[]
pesos, boolean negras){
      float ev;
      ev=pesos[0]*(superest.territorioNegras()-
superest.territorioBlancas())
            + pesos[1]*(bajasBlancas-bajasNegras)
            +                          pesos[2]*(superest.totalNegras()-
superest.totalBlancas())
            +
pesos[3]*(tablero.getNumNegras()/tablero.getNumCadenasNegras()
                        -
tablero.getNumBlancas()/tablero.getNumCadenasBlancas());

      if (!negras){
                  ev=-ev;
      };
      return ev;
}
```

5.5. Estructuras de datos

Durante el desarrollo de las estructuras de datos que deben dar soporte a la implementación de los algoritmos propuestos, nos hemos encontrado en todo momento con uno de los problemas clásicos en programación: si se aumentas la eficiencia en tiempo también aumentas la cantidad de espacio de almacenamiento requerida.

Tradicionalmente la estructura de datos empleada por la mayor parte de los programas de Go para representar el tablero de juego es un simple array lineal de 361 (19x19) posiciones que contiene la información de si la posición x, y está vacía, ocupada por blancas o por negras. La gran ventaja de este enfoque es que minimiza el consumo de memoria, para poder explorar el mayor número de niveles mediante minimax sin que lo impida el tamaño de la memoria.

Sin embargo nuestra intención era disminuir el tiempo para analizar estáticamente el tablero, especialmente en las operaciones de poner piedra por parte del jugador humano. Además teníamos siempre en mente que el consumo de memoria no era tan preocupante, ya que nuestra intención era que la implementación ideal del sistema se pudiera ejecutar sobre una red de ordenadores, en la que los minimax no se ejecutaran solo sobre un ordenador. Por ello decidimos realizar unas estructuras de datos más complejas y que agilizaran lo más posible las evaluaciones estáticas y la detección de piedras muertas.

La primera idea que se nos ocurrió se basaba en una matriz dispersa que sólo representara las posiciones ocupadas del tablero. Las matrices dispersas se emplean para representar estructuras en las que la mayor parte de la información son 0. Cuando se da esta condición se puede representar la información mediante una lista enlazada que no incluye los nodos de información vacía. Las matrices dispersas que decidimos emplear incluían información punteros adicionales para saber las piedras colocadas siguientes y anteriores tanto en horizontal como en vertical. De esta manera los nodos responden al siguiente esquema:

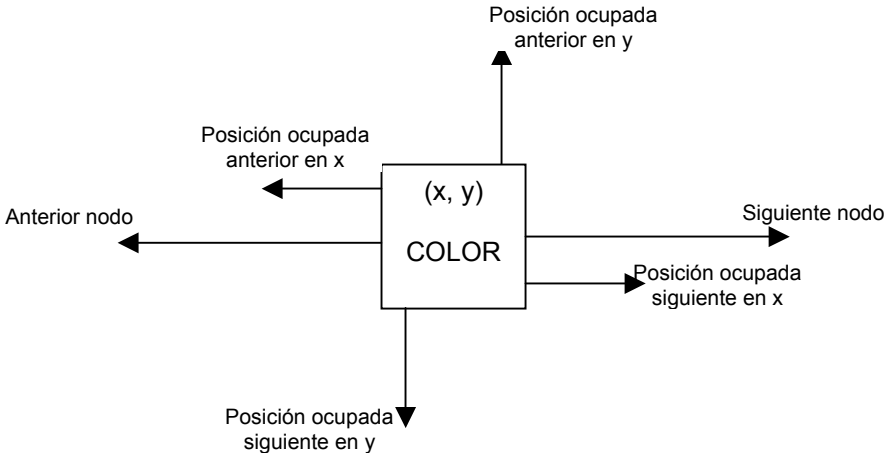

Fig. 20: Nodo de la matriz dispersa para Go

El motivo de que pensáramos utilizar estas matrices dispersas para representar el tablero de Go es que incluyen información de cohesión entre piedras en la propia estructura y que tanto en la fase de apertura como en la de juego medio, el tablero se encuentra casi vacío, lo cual disminuye la penalización en espacio por tener cuenta los punteros necesarios.

Sin embargo pronto comprendimos los grandes problemas que se planteaban con este tipo de implementación, pues si bien permitía rápidamente localizar y formar estructuras de datos que representaran cadenas de piedras se ignoraba mucha información relevante, se ignora la información

que proporcionan las casillas vacías. En Go las posiciones vacías tienen mucha importancia, en primer lugar porque las posiciones vacías contiguas a una cadena son sus libertades, en segundo lugar porque el objetivo del Go es rodear la mayor cantidad de territorio y las casillas que cuentan a efectos de conteo de territorio son casillas vacías, pero sobre todo porque un jugador solo puede poner ficha en una posición vacía y por tanto los movimientos posibles en un momento dado son siempre posiciones vacías.

La implementación mediante matrices dispersas nos obligaba a trabajar deduciendo las posiciones vacías y forzando recorridos adicionales de la matriz completa para hallar las libertades y decidir si una posición está vacía o no. Esto penalizaba grandemente el tiempo necesario para producir las consecuencias de un movimiento, no solo para las jugadas de la máquina, sino también para las del jugador.

Nuestra siguiente decisión fue tratar de aprovechar las ventajas de tener almacenadas las posiciones del tablero en una estructura de array, con las ventajas de estructuras de datos más complejas que pudieran representar cadenas de piedras.

Siguiendo estas ideas el tablero se implementa con dos representaciones que duplican información, por un lado el tablero es un array bidimensional que indica qué hay en cada posición y por otro el tablero es la colección formada por las cadenas de uno y otro jugador:

```
/**
 *    Clase que implementa un tablero de GO
 */
public class Tablero {

      //Constantes
      public static final int DIMENSION=19;
      public static final byte VACIA=-1;
      public static final byte BLANCA=0;
      public static final byte NEGRA=1;

      //Array bidimensional que contiene la informacion de casillas
ocupadas y vacias
      private byte[][] marca_tab;
      //Cadenas del jugador blanco
      private LinkedList cadenasBlancas;
      //Cadenas del jugador negro
      private LinkedList cadenasNegras;

      //Numero de piedras blancas
      private short numBlancas;
      //Numero de piedras negras
      private short numNegras;
      //Numero de piedras blancas capturadas
      private short bajasBlancas;
      //Numero de piedras negras capturadas
      private short bajasNegras;
```

En un principio pensamos que las propias posiciones que forman parte de la cadena y el color de la misma eran la única información relevante para las cadenas de piedras; pero cuando comenzamos a trabajar sobre esta estructura observamos que cuando se colocaba una piedra se debían hacer muchas operaciones que, con un poco más de coste en espacio, se podían acelerar enormemente, por ello decidimos llevar cuenta de las libertades de cada cadena en su estructura. De esta forma saber si un movimiento mata a una cadena es muy rápido y también se agiliza mucho descubrir si un movimiento es ilegal. La ganancia de rapidez en a la hora de poner piedra nos resulta muy atractiva, sobre todo porque consideramos que un jugador humano debe esperar lo menos posible para ver el resultado de efectuar su movimiento.

Las cadenas constan de dos listas, la lista de piedras que la componen y la lista de libertades que tiene:

```
/**
 *    Clase que implementa una cadena de piedras y
sus libertades
 */
public class Cadena {
      //Color de la cadena
      private byte color;
      //Lista de piedras que componen la cadena
      private LinkedList componentes;
      //Lista de libertades (posiciones) de la
cadena.
      private LinkedList libertades;
```

Las cadenas se forman de manera constructiva a medida que se realizan movimientos, esto es, no se analiza estáticamente a cada momento qué cadenas están en juego, sino que cuando un jugador pone piedra se actualiza la información de las cadenas, matando a las cadenas que son eliminadas. A cambio de la ganancia en tiempo, el coste en espacio es muy notable y solo puede soportarse si tenemos en cuenta que la implementación ideal se apoya en una red de computadores y que el minimax no necesita todas las cadenas en juego, sino solo las correspondientes a una zona.

Utilizando esta estructura de datos la realización de movimientos se hace de la forma siguiente: se debe comprobar si el movimiento es legal, para ello se comprueban las cuatro casillas que hay alrededor de la casilla sobre la que se quiere jugar. Si una de las casillas adyacentes está vacía el movimiento es legal, si las cuatro casillas adyacentes están ocupadas hay que comprobar si es una posición de suicidio y por lo tanto el movimiento es ilegal. Para ello se actúa de la siguiente forma, se comprueba si se elimina a una cadena enemiga al realizar el movimiento (si se le quita la última libertad a alguna de las cadenas enemigas adyacentes), hecho esto se toman las cadenas aliadas adyacentes y se unen a la casilla actual, si la cadena resultante no tiene libertades el movimiento es ilegal.

```
//Procedimiento que pone una piedra
//Comprueba si el movimiento es legal y realiza el movimiento
public void ponerPiedra(byte x,byte y,byte color) throws
IllegalAccessException{
      Punto posicion=new Punto(x,y);
      Punto pAux;
      boolean movimientoValido=true;
      boolean mata=false;
      byte xAux;byte yAux;
      LinkedList libertades;
      LinkedList colisiones;
      LinkedList fusiones;
      byte[] cadenasColision;
      byte[] cadenasFusion;
      byte[] cadenasMatar;
```

```
      byte numColisiones;
      byte numFusiones;
      Cadena cadNueva;
      int i=0;int j=0;
      Casilla casilla=new Casilla(posicion,color);
      //Si el jugador intenta poner en una casilla ocupada el
movimiento es
      //ilegal
      if (marca_tab[x][y]!=VACIA)
            throw new IllegalAccessException("Posición del tablero ya
ocupada");
      else{
            //Creamos las posiciones adyacentes
            libertades=casilla.creaAdyacentes();
            //Obtenemos el array de casillas que se fusionan con esta
            cadenasFusion=this.obtenerFusiones(libertades,color);
            //Obtenemos el array de casillas que se colisionan con
esta
            cadenasColision=this.obtenerColisiones(libertades,color);
            numFusiones=cadenasFusion.length;
            numColisiones=cadenasColision.length;
            //Despues de pasar por fusiones y colisiones lo que queda
en
            //libertades son libertades
            cadNueva=new Cadena(casilla,libertades);
            //Fusionamos las cadenas amigas a la nueva
            this.fusionar(cadNueva,cadenasFusion,casilla);
            //Resolvemos las colisiones con cadenas enemigas
            cadenasMatar=this.colisionar(cadenasColision,color);
            mata=cadenasMatar.lentgh>0;
            //Compruebo si el movimiento es valido y si lo es se
realiza
            //esto es, se recuentan las blancas y las negras y se
eliminan
            //las cadenas sobrantes, las fusionadas y las muertas
            if ((cadNueva.getNumLibs()<=1)&&(!mata)){
                  movimientoValido=false;
                  throw new IllegalAccessException("Posicion de
suicidio");
            }
            else {

      this.devuelveLibertades(cadenasMatar,Casilla.enemigo(color));
                  this.elimina(cadenasFusion,color);
                  this.elimina(cadenasMatar,Casilla.enemigo(color));
                  if (color==BLANCA)
                        cadenasBlancas.add(cadNueva);
                  else if (color==NEGRA)
                        cadenasNegras.add(cadNueva);

            }

      }
}
```

Lo más costoso en tiempo de un movimiento es matar a una cadena adversaria, ya que hay que devolver las libertades a las casillas adyacentes. La devolución de libertades consiste en lo siguiente: se toman todas las posiciones que componen la cadena, así como su última libertad. Se localizan las cadenas adyacentes a la actual y se les añade cada una de las posiciones anteriores a sus libertades.

```
//Procedimiento que devuelve las libertades a las cadenas adyacentes de
//las cadenas muertas y modifica el numero de bajas
private void devuelveLibertades(byte[] cadenasMatar,byte color){
      int numCadena;
      Cadena cadenaAuxiliar;
      LinkedList adyacentes;
      boolean modificada;
      Casilla casillaActual,casillaAuxiliar;
      for (int i=0;i<cadenasMatar.length;i++){
            numCadena=cadenasMatar[i];
            if (color==BLANCA)
                  cadenaAuxiliar=((Cadena)cadenasBlancas.get(numCadena));
            else if (color==NEGRA)
                  cadenaAuxiliar=((Cadena)cadenasNegras.get(numCadena));
            //Recorremos cada casilla obteniendo sus adyacentes
            for (int j=0;j<cadenaAuxiliar.getNumPiedras();j++){
                  casillaActual=(Casilla)(cadenaAuxiliar.getCasilla(j);
                  adyacentes=casillaActual.creaAdyacentes();
                  //Si alguna de las adyacentes pertenece a una cadena
enemiga
                  //se le devuelve la libertad
                  for (int k=0;k<4;k++)
                        if (color=BLANCA)
                              for (int p=0;p<cadenasNegras.size();p++){
                              casillaAuxiliar=new Casilla
                        (casillaActual.getPosicion(),NEGRA);
                              if
                              ((Cadena)cadenasNegras.get(p).pertenece(casillaAuxili
                              ar))

      (Cadena)cadenasNegras.get(p).addLibertad(casillaActual.getPosicion());
                              }
                              else if (color==NEGRA)
                                    for (int
p=0;p<cadenasBlancas.size();p++){
                                          casillaAuxiliar=new Casilla
                                          (casillaActual.getPosicion(),BLANCA
                                          );
                                          if
                                          ((Cadena)cadenasBlancas.get(p).pert
                                          enece(casillaAuxiliar))

      (Cadena)cadenasBlancas.get(p).addLibertad(casillaActual.getPosicion());
                                    }
                  }
            }
      }
```

Además de las estructuras de datos de Tablero y Cadena, era necesaria una estructura que diera soporte a la división del tablero en subjuegos, así como a la generación de movimientos para el minimax.

La división de espacios actúa localizando patrones de división de zonas y patrones de unión de zonas, para después aplicar una serie de algoritmos para fusionar alguna de las zonas encontradas. Por ello los subjuegos debían tener información acerca los límites de la zona (las casillas donde se encontraban las divisiones), las cadenas incluidas dentro de la zona, las casillas vacías y el territorio de unos y otros necesario para la función de evaluación.

```
/**
 *
 * Clase que implementa las superestructuras compuestas por cadenas de
distintos jugadores.
 */
class Superestructura {
      //Casillas que delimitan la superestructura
      LinkedList casillasFrontera;
      //Lista de posiciones vacias: movimientos posibles para el
minimax
      LinkedList posicionesVacias;
      //El territorio total encerrado por casillas blancas en la
superestructura
      byte territorioBlancas;
      //El territorio total rodeado por piezas negras en la
superestructua
      byte territorioFrontera;
      //Las cadenas blancas pertenecientes a esta superestructura
      LinkedList cadenasNegras;
      //Las cadenas negras pertenencientes a esta superestructura
      LinkedList cadenasBlancas;
```

5.6. Divisor de zonas de juego

La eficiencia del sistema, como se puede observar, depende en gran medida de la capacidad para descomponer el tablero en subjuegos de una manera adecuada. Basamos nuestro sistema en los métodos de descomposición del tablero a partir de sencillos patrones que aparecen en varios documentos de otros investigadores. A continuación describimos los sencillos patrones que se emplean para unir y dividir zonas.

El patrón de unión más elemental es el que indica que dos piezas se unen formando una cadena. Sin embargo partimos de la base de que antes de realizar la descomposición espacial del tablero ya tendríamos formadas todas las cadenas de piedras, como se explicará en el momento de tratar las estructuras de datos que dan soporte a los algoritmos.

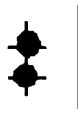

Fig. 21: Patrón de cadena

Estos dos patrones indican unión de las dos cadenas negras y separación de la blanca:

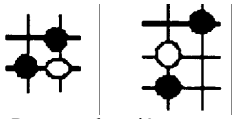

Fig. 22: Patrones de unión y separación 1

Estos dos patrones indican separación entre las cuatro cadenas:

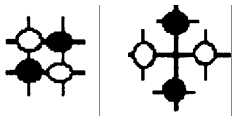

Fig. 23: Patrones de unión y separación 2

Este patrón indica separación de las cadenas negras:

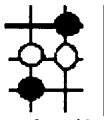

Fig. 24: Patrones de unión y separación 3

Este conjunto de patrones indica contactos entre distintas zonas:

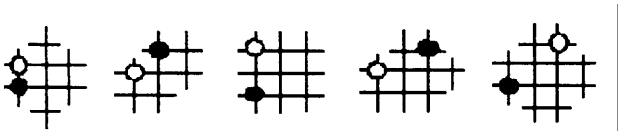

Fig. 25: Patrones de unión y separación 1

Una vez localizados los divisores y puntos de contacto del tablero y las cadenas que los constituyen, haciendo una división preliminar en la que tendremos una situación similar a la siguiente:

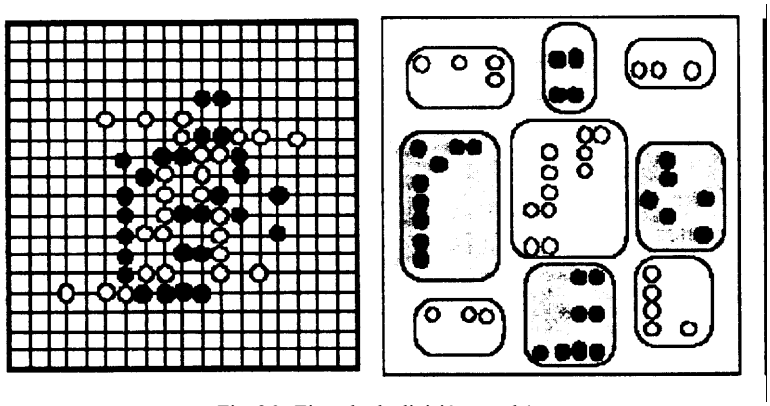

Fig. 26: Ejemplo de división zonal 1

Sin embargo las zonas que hemos encontrado no se constituyen en los subjuegos, porque algunas zonas se fusionaran entre sí basándonos en los patrones de contacto. Las zonas que están en contacto con otras enemigas y se encuentran rodeadas por ellas se fusionan, como se explica en el siguiente diagrama:

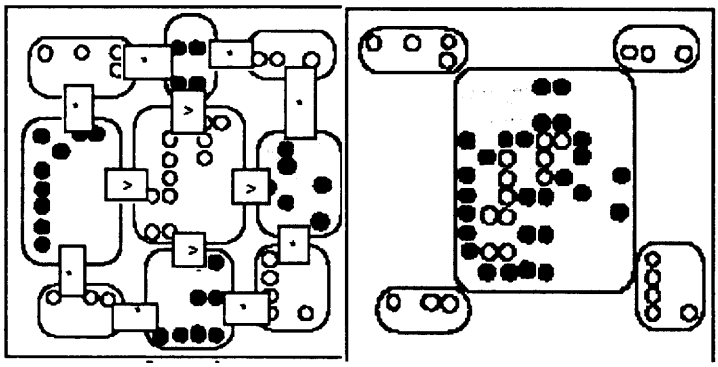

Fig. 27: Ejemplo de división zonal 2

Para saber si una zona está rodeada completamente por otras con las que está en contacto determinamos sus esquinas más exteriores, a continuación se observa como encajan entre sí las zonas rectangulares. En el siguiente esquema la zona A está completamente rodeada por 1, 2, 3 y 4, de modo que si 1, 2, 3 y 4 están ocupadas por piedras enemigas de las de la zona A, hay que unir ambas zonas.

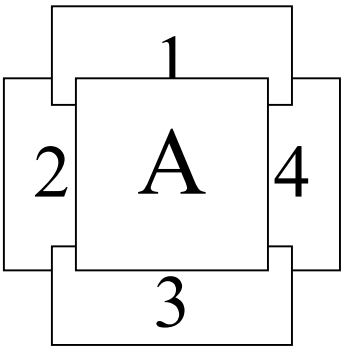

Fig. 28: Zonas que hay que fusionar formando una única superestructura

Una vez encontrados los límites de cada zona y una vez que hemos fusionado las zonas que han de unirse, hay que realizar un análisis estático dentro de cada zona a partir del array bidimensional del tablero para determinar las casillas desocupadas incluidas dentro de cada zona. Las casillas desocupadas son los movimientos posibles para el minimax que se ejecuta localmente en la zona. El generador de movimientos suministra al minimax cada una de las casillas vacías.

Además a cada una de las casillas vacías de cada zona influye en el territorio hay que aplicarle el algoritmo que indica si está rodeada o no por piedras de un jugador para calcular los territorios dominados por cada jugador. El territorio de un jugador es la suma de las casillas vacías rodeadas por piedras de ese jugador.

6. Ampliaciones y mejoras

6.1. Distribución de la carga de proceso

La división del tablero en zonas de juego independientes proporciona la posibilidad de distribuir la carga de cálculo computacional entre varios hilos de proceso, repartidos en varios procesadores. Esta mejora permite paralelizar el cálculo de varias zonas a la vez, reduciendo considerablemente el tiempo de proceso. Además, con la división, se obtendrá la ventaja añadida de reducir la memoria utilizada en un computador, repartiendo la carga de las estructuras entre todos los computadores de la red.

Esta mejora consistirá, básicamente, en realizar una implementación del algoritmo del minimax básico utilizado para cada zona del tablero en varios computadores de la red, que se dedicarán a calcular árboles minimax de forma exclusiva siempre que sean requeridos para tal fin. Los resultados de las zonas se enviarán al computador que ejecuta la hebra principal para ser combinados y que sea posible la selección de la mejor jugada encontrada.

Pero estas mejoras no son tan importantes como podría parecer: La distribución de tareas de esta forma encaja bastante con el paradigma de la programación paralela, con ciertas salvedades.

El proceso principal almacena todas las variables de la partida, así como las estructuras del estado global de la misma. La creación de estructuras para el proceso de cada una de las partes del tablero es un proceso que ocupa tiempo y espacio en memoria, ambos necesarios para la posterior aplicación del algoritmo de minimax. Pero para que los procesos puedan ser independientes completamente, se debe proporcionar a cada uno una copia de las variables que le resulten necesarias para hacer los cálculos, por lo que el ahorro de memoria no es tan importante como podría parecer en primer momento.

Aún así y pese al aumento de ocupación de la memoria que supone la copia de los datos en los computadores dedicados al cálculo del minimax, gracias a la dedicación exclusiva y al paso únicamente de las estructuras necesarias para el funcionamiento del algoritmo, se consigue que la memoria de los procesadores dedicados esté suficientemente libre como para soportar el cálculo de un árbol de minimax mayor que con un solo procesador. La mejora que se ha obtenido en las pruebas realizadas resultó ser de entre 6 y 10 niveles para el minimax por zonas, frente a 3 o 4 niveles para el algoritmo con el tablero completo.

La mejora en tiempo varía según el número de procesadores con los que se cuente, resultando viable a partir de 2 procesadores dedicados al cálculo, con una mejora de un 30% aproximadamente. Para la obtención de una mejora significativa, sería deseable contar con entre 8 y 10 procesadores, que es la cantidad promedio de subjuegos que pueden esperarse en una partida de Go habitual. Esta mejora depende del tiempo que tarde el proceso principal en realizar la copia de los datos para cada proceso, por lo que llegará un momento en que la cota del tiempo vendrá dada por el retardo de la realización de dicha copia para el último de los procesos más el tiempo que tarde éste en realizar el cálculo del minimax. Con una búsqueda exhaustiva, la construcción de los árboles por parte de los procesadores dedicados será superior a este tiempo, pero en términos prácticos, no será deseable ni, probablemente representará mejora alguna aún con tiempo de copia cero, un número de procesadores dedicados superior a 15, que es el máximo esperable de zonas que podemos encontrar en una partida.

Para la implementación del algoritmo se necesita un mecanismo de reconocimiento de los computadores dedicados, que controle si están ocupados o no, a qué se dedican y, en caso de que se utilice una red, cuál es la dirección de red de cada uno.

Para esto, es posible utilizar una pequeña base de datos u otro mecanismo de almacenamiento de información. Al elegir el lenguaje Java para las pruebas de implementación, se han utilizado los mecanismos que éste ofrece para la realización del mecanismo distribuido (Distribución mediante clientes-servidores con RMI). Para las pruebas se utilizó una implementación sobre base de datos Access, con una tabla que almacenaba todos los datos necesarios para cada uno de los servidores y una casilla de ocupado.

El algoritmo para el cliente, en seudo-código, sería el siguiente:

```
Tablero T = Posición actual del tablero;
Zonas Z = ∅;
DivisorTablero(T, Z);

Resultados R[1..N]; // almacenará los resultados y
las jugadas a las que se refieren

//Ahora Z = {z1, z2, ... , zn};

para cada Zona zi de Z hacer
      Crear nueva hebra de proceso.
            si hay      un servidor libre Servk
entonces
                  Obtener datos para llamada remota a
Servk
                  Servk.resultado = Evaluar (T, zi);
// llamada remota al servidor
                  R[i] = resultado;
            sino
                  esperar;
            fsi;
      Muerte de la hebra.
fpara;
```

Este código lanzará las peticiones de evaluación a los servidores cuando estos no estén ocupados. La opción de enviar en orden a los servidores (esto es, z1 a servidor 1, z2 al servidor 2, etc.) es menos eficiente, puesto que no se garantiza que los servidores terminen de evaluar en orden, pudiendo un servidor haber terminado de evaluar y estar libre, mientras que otro tenga varias hebras bloqueadas en espera de que se libere para poder continuar.

En cuanto al código de los servidores, será básicamente el mismo código que el del algoritmo utilizado para la evaluación del problema completo, pero adaptado a la aplicación sobre una zona.

El algoritmo en seudo-código sería el siguiente:

```
// procedimiento que calcularía la jugada siguiente
en el servidor RMI de evaluación

proc sincronizado Evaluar(Tablero T, Zona z) :
{float mejora, Posicion jugada);
begin
      Marcar este servidor como ocupado en la BD;
      Calcular(T, z);    //Realiza cálculos de
árboles u otras formas de evaluación


      Marcar el servidor como desocupado en la BD;
      Devolver resultado;
end;
```

La reducción del problema a subdivisiones mejora significativamente el rendimiento del programa, pero conlleva otro problema añadido: En 1998, el autor del juego Intellect realizó esta mejora en la segunda versión del juego, Intellect II. El resultado fue un completo desastre: El juego cayó de la segunda plaza, obtenida en la copa FOST en 1997, al sexto puesto en la edición de la copa del año 1998.

El problema encontrado consiste en que la división en zonas no resulta invariablemente en problemas separados, sino que el desarrollo del juego en una región puede tener efecto en las otras zonas, tanto cercanas, mediante la eliminación de libertades de piedras que están en los bordes de las zonas contiguas, debido a que se jugó en el borde de la zona, como lejanas, mediante la posibilidad de captura o afirmación de una cadena que se extiende por una o más zonas en el tablero completo y de la cual, sólo una porción pertenece a la zona actual. Otra posibilidad es que los movimientos correctos son calculados en una zona, pero después continúan en otra, obteniendo árboles incorrectos, pues el juego en una zona implica que el jugador ya ha realizado su movimiento y que no puede jugar en otra. Esto, además, complica la obtención del minimax correspondiente, puesto que ya no puede basarse en la sucesión de jugadas negro-blanco, al tener un jugador la posibilidad de pasar o de jugar en una zona diferente de la que jugó su contrincante.

Este cambio en la estructura del árbol de búsqueda lleva a la investigación de un nuevo método de búsqueda de soluciones. La técnica conocida como "*soft descomposition search*", que podríamos traducir como búsqueda mediante descomposición blanda, resuelve los problemas antes mencionados, gracias al empleo de varias técnicas para identificar la urgencia de jugar en una zona determinada frente a las demás y para reflejar los efectos locales, colaterales y globales de una jugada en una zona, además de generar los movimientos más prometedores independientemente de la división en dichas zonas.

Este cambio radical en el planteamiento de la implementación de los algoritmos de juego lleva a la construcción de nuevas estructuras de datos que puedan contener la información necesaria. Estas estructuras son conocidas como árboles binarios de juego (Binary Game Tree, a partir de ahora BGT).

Un BGT es capaz de almacenar las jugadas mejores para cada zona. Siguiendo la notación propuesta en [2], en jugador de negras será el situado a la izquierda (jugador L) y el de blancas se colocará a la derecha (jugador R). Para el minimax realizado por el BGT, el jugador maximizador será el jugador negro (L), mientras que el blanco será minimizador.

Aquí se expone un claro ejemplo de la construcción de un BGT:

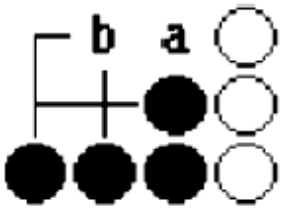

Fig. 29: Ejemplo de construcción de un BGT 1

Tenemos la posición de la imagen y juegan negras. Si L juega en el punto a, obtendrá 4 puntos, como puede verse en el siguiente diagrama.

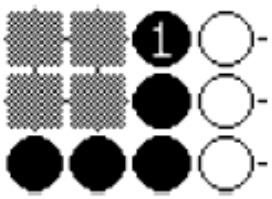

Fig. 30: Ejemplo de construcción de un BGT 2

Estos son los cuatro puntos obtenidos por L si éste juega en a, pero existe la posibilidad de que le toque a blancas jugar en esta posición. Si R juega en la casilla a, entonces L debe jugar en b para bloquear la zona y poder puntuar, en este caso, 3 puntos

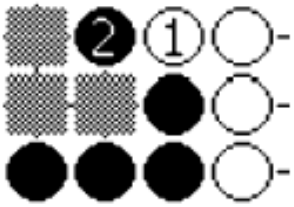

Fig. 31: Ejemplo de construcción de un BGT 3

Sin embargo, si L ignora esta zona después de la jugada de R en el punto a, R podrá volver a jugar en ella, evitando que L puntúe en esta zona.

El siguiente BGT describe la situación expuesta anteriormente. Las ramas izquierdas representan movimientos de L y las ramas derechas movimientos de R.

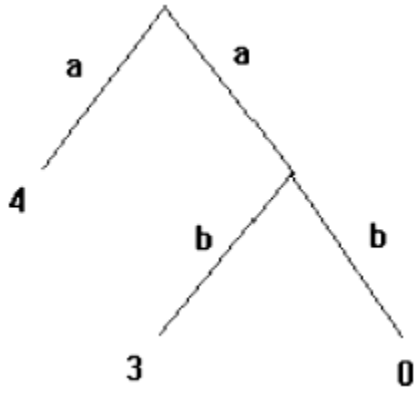

Fig. 32: BGT que ilustra la situación de ejemplo

Los nodos terminales están marcados con sus respectivos valores de la función de evaluación y la jugada realizada está anotada sobre la rama correspondiente.

En la notación de la teoría combinatoria de juego, éste BGT se puede representar mediante la siguiente expresión: T1 = {4 || {3 | 0} }

El ejemplo siguiente es más complejo:

Fig. 33: Un BGT más complejo 1

Partiendo de esta posición, el mejor movimiento que pueden realizar las negras es a. Tras esta jugada, viene la secuencia Blancas b, Negras e, terminando la situación de esta forma:

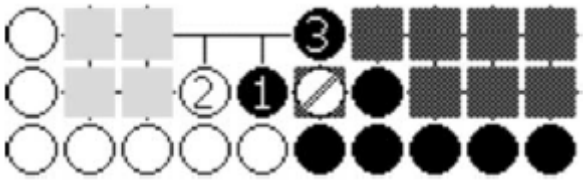

Fig. 34: Un BGT más complejo 2

El jugador L tiene 9 puntos (considerando la puntuación japonesa, cada captura es un punto adicional), mientras que R tiene solamente 4 puntos. Esto da una ventaja de +5 a favor de L (maximizador)

Pero si L deja pasar la oportunidad en la zona y R juega primero, lo hará sobre e, llegando a la siguiente situación:

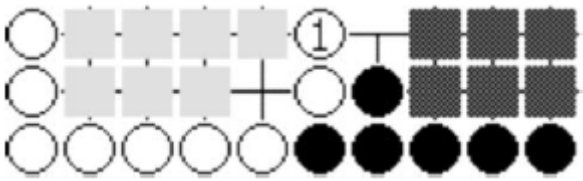

Fig. 35: Un BGT más complejo 3

En esta situación, R tiene 7 puntos y L tiene 6, lo que da un balance de –1.

Sin embargo, si es R el que deja pasar la zona tras la jugada de L, L volverá a jugar, esta vez en b, obligando a R a jugar sobre c para no perder su territorio y su cadena.

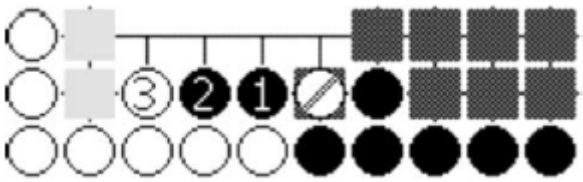

Fig. 36: Un BGT más complejo 4

Ahora, L tiene 9 puntos y R sólo 2, por tanto el balance es de +7

Por supuesto, en las zonas de la partida donde el juego está aún en una fase temprana de desarrollo, resultará imposible realizar la búsqueda hasta nodos terminales del árbol, por lo que dicha búsqueda debe suspenderse y los nodos calculados en el último nivel deberán ser tratados como nodos terminales y evaluados y etiquetados como tales.

El siguiente BGT representa esta situación con algo más de profundidad:

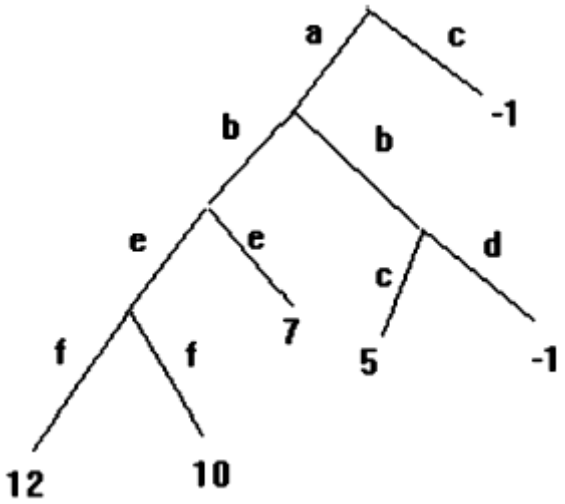

Fig. 37: Un BGT más complejo 5

T2 = { { { {12 | 10} || 7} || {5 | -1} } || -1}

Una vez explicada la formación de un BGT, podrá darse una definición formal de éste:

Un árbol binario de juego (BGT) es un árbol que cumple que:

- Cada nodo no terminal tiene dos hijos, uno izquierdo y otro derecho
- Cada nodo terminal está asociado con un número racional (normalmente, un entero)

Si T es un BGT que contiene un nodo terminal únicamente, será denotado con el número n, que es el valor asociado al nodo por la función de evaluación. Si no, se utiliza la notación combinatoria descrita anteriormente: $T = \{ T^L \parallel T^R \}$ donde T^L será el subárbol izquierdo y T^R el derecho.

Dado un BGT de nombre T, la definición de valor derecho y valor izquierdo del BGT es la siguiente:

$$L_V(T) = \begin{cases} v & \text{si T es un nodo hoja y v es su valor} \\ R_V(T^L) & \text{en otro caso} \end{cases}$$

$$R_V(T) = \begin{cases} v & \text{si T es un nodo hoja y v es su valor} \\ L_V(T^R) & \text{en otro caso} \end{cases}$$

Por lo tanto, L_V (análogamente para el otro lado con R_V) representa el valor de jugar en esta zona asumiendo que L juega primero y tras un intercambio de movimientos en esta zona.

El cálculo por zonas obliga al programa a realizar la construcción de un BGT para cada una de las zonas. Al conjunto de los árboles binarios de juego generados se le da el nombre de Bosque Binario de Juego (Binary Game Forest, a partir de ahora BGF)

El nuevo problema, por lo tanto, es conseguir ordenar el BGF, de tal forma que los jugadores realicen sus movimientos en la zona del tablero que más puntuación pueda proporcionarles teniendo en cuenta, además, la "urgencia" de juego en dichas zonas, es decir, lo mucho o poco que una zona puede aguantar sin un movimiento o también lo deseable que es obtener el ella la iniciativa.

El algoritmo que se describe a continuación es conocido como Mean-Temperature (algoritmo de Relevancia-Temperatura, a partir de ahora M-T)

La temperatura y la relevancia son dos términos formales definidos en [2, 8] para la teoría de juegos combinatorios. Hablando en términos sencillos, la temperatura indica la urgencia de jugar en determinado subjuego, mientras que la relevancia es una estimación del incremento de la puntuación que esta zona reportará al jugador L. El nivel de los jugadores humanos de Go se basa en su habilidad para reconocer la urgencia de ciertas posiciones y los beneficios que se obtendrán de ellas. La temperatura servirá para propósitos de ordenación de los árboles dentro del BGF, basándose en calcular una solución para un movimiento global con técnicas basadas en alfa-beta sobre el BGF.

El cálculo del algoritmo M-T se realiza mediante la construcción de unas estructuras que analizan la temperatura (termogramas). Pero pueden omitirse gracias a los teoremas de estabilidad, cuya demostración aparece en [14, 15] y que serán mostrados más adelante.

Sea T un BGT, llamaremos hijos alternativos izquierdos de T a los hijos T^{L}, T^{LR}, T^{LRL}, ...

De la misma forma, se definen los hijos alternativos derechos de T: T^{R}, T^{RL}, T^{RLR}, ...

Los niveles del árbol se numeran comenzando por el cero, asignando éste a la raíz.

Un hijo de T se dice que es estable si su temperatura no excede la de T. Sea T' el primer hijo alternativo izquierdo estable y T'' el primer hijo alternativo derecho, también estable, la temperatura y relevancia de T viene determinada por T' y T'' como sigue:

Teorema de estabilidad:

Si T' y T'' están ambos en un nivel impar:
$Temp(T) = (Sign(T') - Sign(T'')) / 2$
$Sign(T) = (Sign(T') + Sign(T'')) / 2$

Si T' está en un nivel impar, pero T'' está en uno par:
$Temp(T) = Sign(T') - Sign(T'')$
$Sign(T) = Sign(T'')$

Si T' está en nivel par y T'' está en nivel impar:
$Temp(T) = Sign(T') - Sign(T'')$
$Sign(T) = Sign (T')$

Si T' y T'' están ambos en niveles pares:
Temp(T) = max (Temp(T'), Temp(T''))
Sign(T) = (Sign(T') + Sign(T'')) / 2

Volviendo al BGT del primer ejemplo:

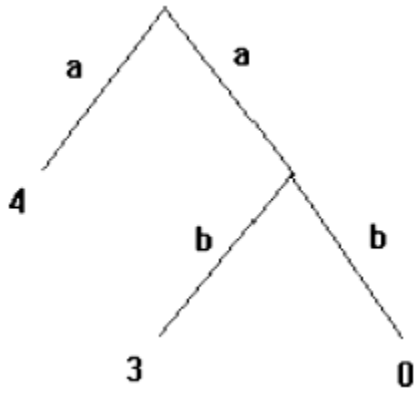

Fig. 38: BGT que ilustra la situación de ejemplo

Se parte de la premisa de que todos los nodos terminales tienen temperatura 0. Los valores de los nodos hoja son las relevancias respectivas. Procediendo en orden de abajo a arriba, se obtienen los siguientes resultados:

$Temp(T^R) = (3 – 0) / 2 = 1.5$

$Sign(T^R) = (3 + 0) / 2 = 1.5$

Considerando ahora T, se asume que los nodos hoja son estables, así que T^L lo es. Asumiremos también que T^R es estable también, por ello, la temperatura de T es Temp(T) = (4 – 1.5) / 2 = 1.25 < $Temp(T^R)$, por lo tanto, T^R no puede ser estable. Así que, el primer nodo estable alternativo derecho será T^{RL} (es una hoja, por definición es estable) y Temp(T) = 4 – 3 = 1 y Sign(T) = Sign (T^{RL}) = 3.

Estas aproximaciones sirven para realizar los cálculos M-T en la mayoría de BGT. En particular, la aproximación es correcta para árboles cuyos nodos disten de las hojas en 2 niveles o

menos y bastante buena para varios niveles más, dependiendo de la complejidad del BGT a analizar. Si se quieren mejoras más acusadas para la realización de estos cálculos, existe una forma diferente descrita por Kao en [14, 15]

El modelo de BGF se basa en los cálculos de M-T para los BGT que representan las diferentes zonas de tablero de la posición actual de la partida de Go. Aún se debe tener en cuenta, que la temperatura de los nodos hoja en una partida sólo es realmente cero cuando son verdaderos nodos hoja, no cuando lo son debido al límite de la profundidad de búsqueda. Por lo tanto, aún se debe llevar la cuenta de un valor que podría ser llamado Temperatura de calma actual y que representa la temperatura del juego cuando se llega a los nodos terminales no reales.

Un Bosque Binario de Juego (BGF) se define de la manera siguiente:

Es un conjunto de BGT o una suma de BGT como se define en teoría de juegos combinatorios. En notación:

$$F = \{ T_1, T_2, T_3, \ldots \} = T_1 + T_2 + T_3 + \ldots$$

Donde cada Ti es un BGT

Habitualmente, un BGF está compuesto por entre 6 y 8 BGT, además del valor de la Temperatura de calma del juego. Este valor es aproximadamente 8 para inicios de partida, disminuyendo a 6 cuando se abre la partida a los bordes del tablero, a 4 para finales de juego tempranos y a 1 o incluso 0.5 para finales de juego antes de rellenar los huecos sobrantes.

Es posible definir dos tipos de subjuegos para el Go: subjuegos calmados, donde no hay verdaderas amenazas entre cadenas, sino sólo captura de territorio por ambas partes. Estas zonas pueden resolverse mediante una búsqueda local, considerando, probablemente, sólo movimientos que aumenten el territorio propio o disminuyan el territorio enemigo.

En los subjuegos urgentes, sin embargo, las piedras de ambos bandos forman cadenas que no están seguras, siendo necesario llevar a cabo búsquedas locales selectivas para cada una de estas zonas, donde es necesario conocer los resultados de jugar primero, de jugar segundo y de ignorar la jugada del contrario o que el contrario ignore la jugada. La evaluación de un nodo terminal se realiza globalmente, como una evaluación del tablero completo, para poder encontrar posibles efectos laterales de la jugada en otras zonas del tablero.

- ***Meta-Búsqueda:***

Tras la exposición de los algoritmos necesarios para el cálculo por zonas, resulta necesario hacer una pequeña ordenación dentro del BGF para poder realizar algoritmos de búsqueda con poda de candidatos, es decir, una aproximación al algoritmo alfa-beta, pero para BGF. Existen muchas estrategias heurísticas en la teoría de juegos combinatorios, así como multitud de algoritmos específicos para BGF. Aquí se darán las claves para la realización de un algoritmo basado en la poda alfa-beta, que puede dar solución exacta al problema si tiene los recursos computacionales suficientes.

Primeramente, se deben ordenar los BGT por temperatura, los de temperatura más alta primero.

Debido a que nuestro objetivo principal es la realización de movimientos de forma inteligente y dado que este modelo es una aproximación, se debería ajustar la evaluación, por ejemplo, por Temperatura_de_Calma / 2 dependiendo de quién será el próximo en jugar tras el último nivel de árbol obtenido, para simular así posibles (y muy a menudo seguros) niveles por debajo del obtenido.

A continuación se muestra el algoritmo de Meta-Búsqueda en seudo Pascal:

```
MetaSearch (F: BGF; depth: integer; toplay: BlackWhite; var
bestmov: integer; alfa, beta: real): real;

var
i, k, bc: integer;
m, t: real;
F': BGF;

begin
   bestmov := 0;
k := Número de BGT en F;
if (Todos los BGT de F son de un solo nodo o depth >= DepthLimit )
then
begin
   if (toplay = Black) then
         return( Sumatorio de Sign(Ti) + Temperatura_de_calma/2 );
   else
         return ( Sumatorio de Sign(Ti) - Temperatura_de_calma/2 );
end;

Ordenar los BGT de F por orden de temperatura de mayor a menor;

if(toplay = Black) then
begin
   m := alfa;
   for j := 1 to k do
```

```
    begin
          F' := {T1, T2, ... , Tij-1, Tij^L, Tij+1, ... Tk};
          t := MetaSearch(F', depth + 1, White, bc, m, beta);
          if(t > m)
          begin
m := t;
bestmov := ij;
               end;
if( m >= beta)
return m;
          end;
          return m;
    end;

    else
    begin
          m := beta;
          for j := 1 to k do
          begin
               F' := {T1, T2, ... , Tij-1, Tij^R, Tij+1, ... Tk};
               t := MetaSearch(F', depth +1, Black, bc, alfa, m);
               if(t< m)
               begin
                    m := t;
                    bestmov := ij
               end;
               if(m<=alfa)
                    return m;
          end;
    end;
end;
```

La llamada inicial para este procedimiento será la siguiente:

MetaSearch (BGF, 0, sig_jugador, mejor_mov, -361, 361);

Se puede considerar, sin excesivo grado de error, que esta aproximación reduce drásticamente la complejidad de la resolución del problema del Go. La calidad de la captura de los mejores movimientos locales es esencial para el buen funcionamiento de la solución mediante BGF aquí descrita, aunque una implementación eficiente, correcta y funcional de este paradigma llevaría más de un año.

6.2. Descripción del uso de patrones

- **Apreciaciones generales.**

Al afrontar el problema de la resolución de una partida de Go, nos vemos en la necesidad de realizar búsquedas por patrones de juego. Éstos van a acelerar el cálculo y resolución de problemas concretos en momentos concretos de juego, que van a ser:

Inicio de partida

Existen ciertas aperturas estándar que definen cuales son los mejores movimientos iniciales en función de algunas configuraciones del tablero. Éstas pueden ser aplicadas de manera directa en forma de patrones.

División de Zonas

Cuando deja de ser aplicable la apertura "estándar" nos encontramos con un tablero que es demasiado extenso como para tratarlo directamente, por ello se procede a una división por zonas de juego, donde afrontamos pequeños subjuegos que sí son tratables. Para realizar esta partición recurrimos a unos pequeños patrones de división.

Escaleras

Las escaleras son un tipo especial de jugada que permite una ventaja estratégica muy amplia cuando son aplicables y dan una captura segura de cara a la resolución de la partida.

Finales de Partida

Cuando un subproblema llega a un momento terminal (cuando un minimax puede resolverlo en 5 o 6 niveles) se puede acortar este calculo con patrones, que nos dan un resultado idéntico al que nos daría un minimax.

- **Reconocimiento de patrones en Go.**

En líneas generales un patrón va a ser un conjunto de puntos en rectángulo, que va a representar un posible fragmento de tablero, donde cada punto va a tomar uno de los valores vacío, negro, blanco o irrelevante.

Si denotamos el estado de cualquier punto en el tablero como Estado(x, y) y al estado de cualquier punto del patrón Valor(x, y), podemos decir que un patrón es reconocido si, para todo par (x, y) dentro del rango en el que trabajamos, cumple que (Valor(x, y) = irrelevante) \/ (Valor(x, y) = Estado(x, y)) [22].

Todo patrón tiene además una acción, es decir, aquello que debe ser realizado si el patrón casa. Esta acción viene a ser habitualmente una posición relativa al patrón donde debe ser colocada la siguiente piedra.

No obstante, cuando los patrones pueden estar siendo usados en paralelo a un método de búsqueda heurística, como en el caso que nos ocupa, es conveniente devolver también una evaluación de la jugada que hemos reconocido.

En este caso concreto también se trata la posibilidad de que existan más cosas asociadas al patrón que un movimiento simple, como se verá más adelante, como una secuencia de jugadas ya evaluadas.

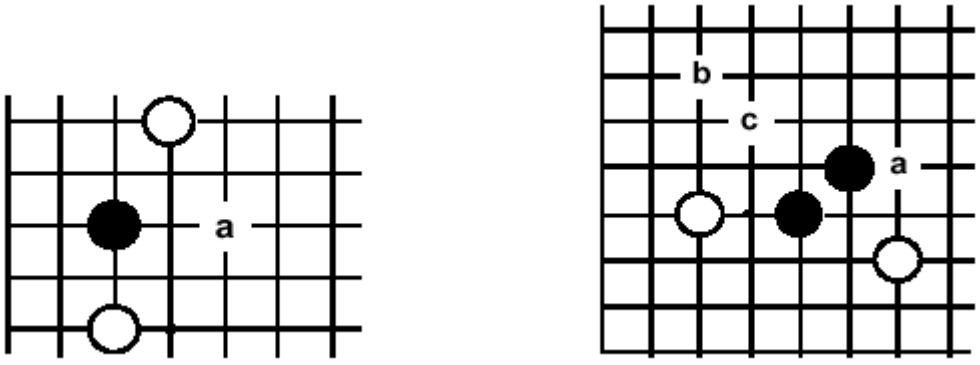

Fig. 39: Patrones ejemplo de jugadas preevaluadas

Estos patrones son un ejemplo de lo comentado. Las posiciones a, b y c serían las zonas donde se debe jugar si el patrón es reconocido.

A la hora de afrontar el reconocimiento de patrones lo más habitual es usar un método de tabla hash que permita una búsqueda rápida y eficiente. No obstante esto es únicamente útil cuando todos nuestros patrones sean del mismo tamaño, esto ocurre con los patrones de inicio de partida que guardan el tablero completo o en los de división de zonas que son 2x2 ó 3x2.

Para el caso en que los patrones no tengan un tamaño uniforme, como en los de final de partida o los de escalera (e incluso en un diccionario Joseki avanzado como se puede ver en el siguiente apartado), algunos investigadores versados en el tema [22] recomiendan los árboles Patricia, que es el método usado en el Diccionario de Inglés de Oxford.

Este método se basa en indexar los puntos del patrón, convirtiéndolo así en una cadena de puntos ordenados que sirven de coeficientes al árbol. Éste luego se aplica al tablero usándolo como guía de búsqueda.

Por último, queda la elección entre varios patrones cuando todos ellos casan en una zona concreta del tablero (cosa que es bastante habitual si existen patrones pequeños). Para ello vamos a distinguir tres tipos de patrones, los de centro, los de lateral y los de esquina. Ahora seguimos estas reglas básicas:

- Se descartan aquellos que no conforman una estructura.
- Se eligen patrones de esquina sobre los de lateral, y laterales sobre los de centro.
- Se eligen los patrones más grandes frente a los más pequeños.

Además de estás sencillas reglas, si el patrón tiene una evaluación también se tiene en cuenta, aunque habitualmente en último lugar.

Por otro lado, en un sistema distribuido pude mejorarse la aplicación de patrones mediante el uso exclusivo de una o varias máquinas de la red para el reconocimiento y la aplicación de patrones.

- **Patrones de inicio de partida.**

Como ya hemos comentado, existen ciertas "salidas estándar" para las partidas que han sido definidas por expertos humanos en la partida a fin de enseñar la mecánica del Go a los nuevos jugadores.

Básicamente existen dos de estas salidas, Joseki y Fuseki. Consisten en seguir una serie de movimientos considerados acertados. Cada uno de estos patrones o estrategias tienen una dirección distinta. Pasemos a considerarlas de manera individual:

Fuseki.

Fuseki es la apertura propiamente dicha. Se basa en ir tomando posiciones en las esquinas y laterales del tablero para afianzar el territorio en los bordes y, a partir de esos puntos fuertes, tratar de conquistar el resto del tablero.

Éste es un patrón sencillo, son unos 25 movimientos sin ninguna clase de conflicto que permite distinguir muy bien el inicio de juego del juego medio, y preparar después una división del tablero en zonas con mayor facilidad.

Para que sea aplicable, el contrario también debe seguir esta filosofía de juego, es decir afianzar primero y atacar después.

A continuación podemos ver algunos ejemplos de lo que podría ser un inicio de partida con Fuseki:

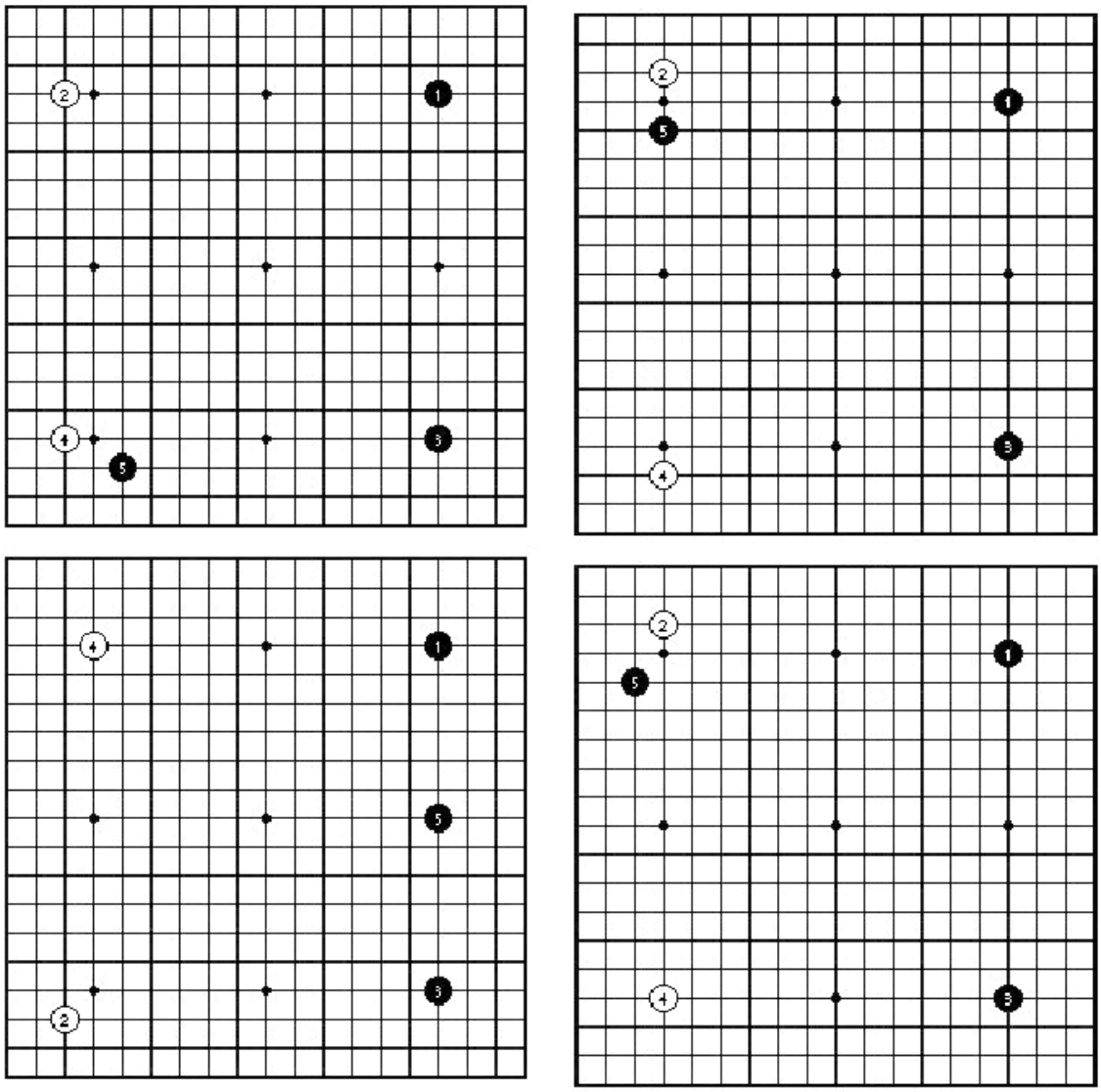

Fig. 40: Fuseki

Como podemos ver en el ejemplo tenemos el tablero en “calma”, es decir, los jugadores toman posiciones que van definiendo un territorio inicial para cada jugador.

Estos son sólo unos ejemplos de cuatro posibilidades con Fuseki, existen muchas otras más o menos agresivas dependiendo del jugador.

La ventaja que ofrece Fuseki desde el punto de vista de los patrones es que es muy sencillo de usar y de casar. Son patrones de tablero completo con pocas piezas en las que la mayoría del tablero es irrelevante y en el mismo instante en que no se puede aplicar se pasaría a juego medio.

Joseki.

Joseki es una visión radicalmente opuesta a Fuseki y también más compleja.

La idea es pelear cada lateral del tablero uno a uno para después pasar al juego medio. Así pues un diccionario de patrones Joseki puede incluir desde simples inicio de disputa hasta patrones complejos con escaleras, problemas de vida y muerte e incluso los más extensos hasta *ko* y *seki*.

Veamos las siguientes posibilidades de inicio con Joseki en una de las esquinas:

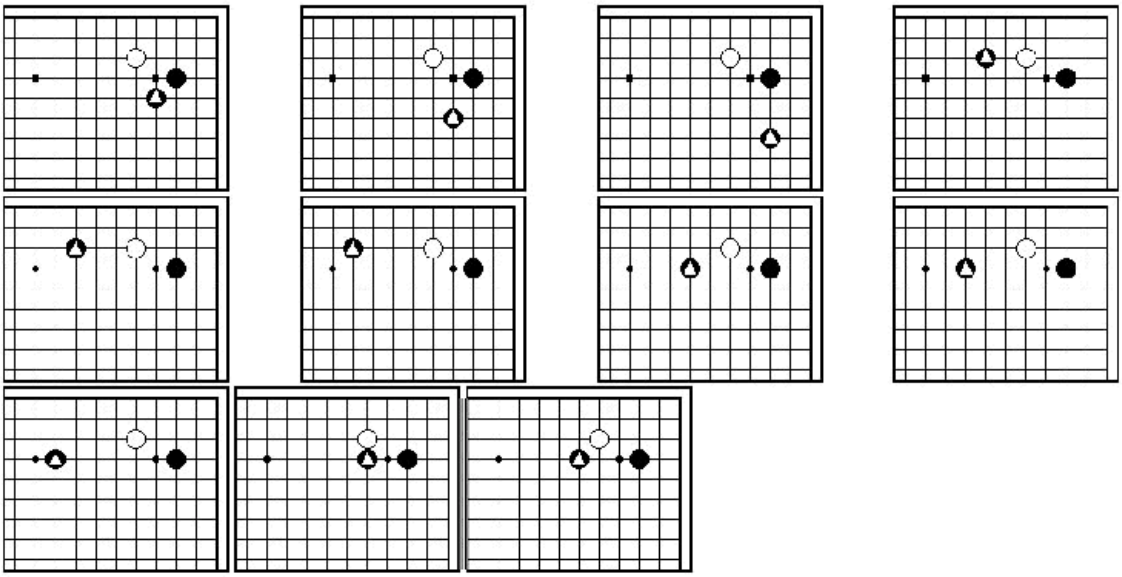

Fig. 41: Una variante de Joseki

En este caso el patrón es para negras, y observamos en él como se está peleando la esquina sin pensar en el resto del tablero de momento.

Observemos ahora las posibles respuestas para blancas que recomendaría una salida en Joseki:

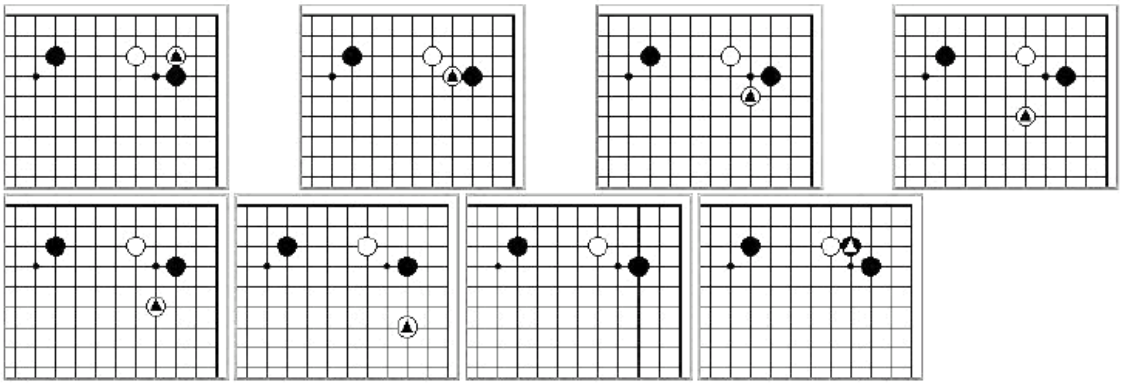

Fig. 42: Otra variante de Joseki

Como vemos se intenta conquistar cada esquina y cada lateral sin importar en modo alguno el resto del tablero, y porque puede incluir escaleras o problemas de vida y muerte.

Con Joseki el juego medio no se distingue del inicio salvo porque aquí en principio tenemos una sola zona, cuando esta se resuelve se pasa a la siguiente y así.

La desventaja de Joseki es que no queda claro cuando se pasa a juego medio, ya que en el momento que se esté disputando una zona y uno de los jugadores pase a otra ignorando la disputa, los patrones dejan de ser aplicables, por lo que un jugador suficientemente experimentado puede hacer que este patrón ya no pueda ser aplicado por la máquina.

Por el contrario un diccionario Joseki suficientemente extenso puede ser aplicable en muchos más momentos de partida que Fuseki que sólo es valido en salidas y si estás son “tranquilas”.

- **Escaleras**

Como ya se ha dicho, las escaleras son una jugada estratégica que permite obtener ventaja en un momento dado. Se trata de un patrón del juego en el que una cadena rodea a otra dejándole tan sólo dos libertades contiguas.

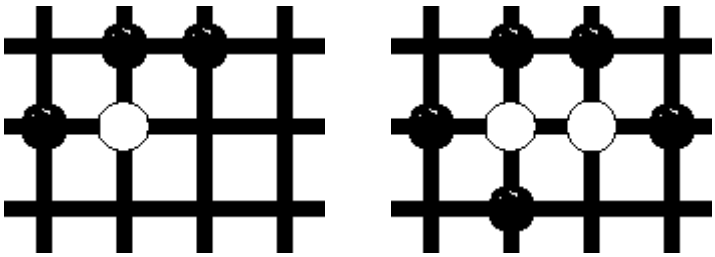

Fig. 43: Patrón básico de escalera

Este es el patrón básico de una escalera. La cadena blanca sólo posee dos libertades y las negras llevan la iniciativa. Por ello pueden empujara a la cadena blanca a ser una escalera que al topar con un de los bordes estará muerta.

El problema de esta jugada es que sólo es aplicable si en todo el recorrido de la escalera no existe ninguna otra cadena blanca con libertades suficientes, ya que de lo contrario las piedras negras se verían comprometidas. Es decir, como la estructura que forman las piedras negras es muy débil (no conforman cadena entre ellas), en el momento en que una piedra blanca se encuentre en el recorrido de la escalera, las negras pierden la iniciativa de ataque y las blancas la ganan pudiendo capturar una parte importante de las piedras negras.

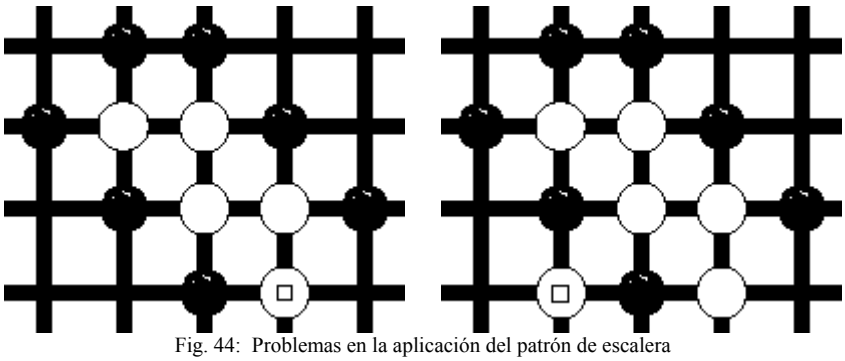

Fig. 44: Problemas en la aplicación del patrón de escalera

Si la piedra blanca marcada estaba ya puesta en el tablero y se llegara a esta situación de escalera, las blancas podrían colocar en la posición marcada en la siguiente figura, amenazando dos piedras negras y rompiendo toda la estructura.

Debido a esta debilidad, los patrones de escalera son bastante extensos en zona de tablero. Ya que tienen que prever este tipo de situaciones.

Existe una segunda opción al respecto, que se basa en incluir un patrón pequeño que identifique la escalera y varias funciones más o menos complejas que determinan si el patrón es aplicable en función del estado actual del tablero.

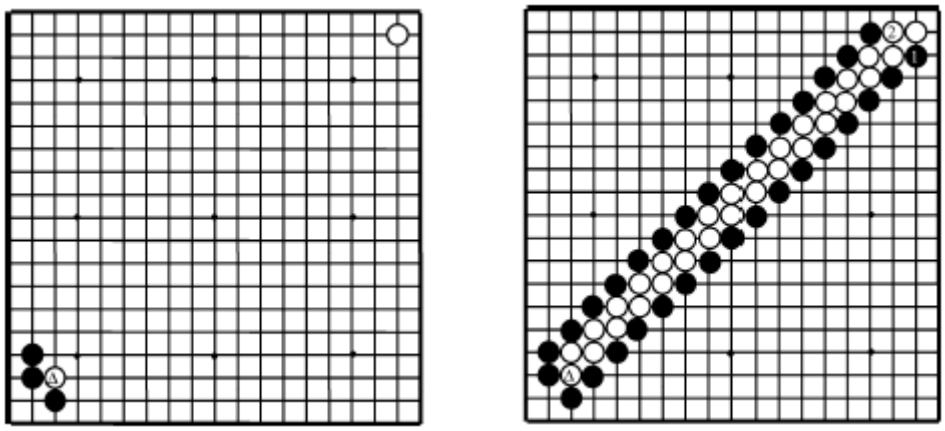

Fig. 45: Escalera que necesita una función porque abarca la diagonal del tablero

En este ejemplo vemos un caso clásico en el que se necesitaría una función. Dado que no es aceptable incluir un patrón de tablero completo (ya que podría haber más piezas en lugares próximos que no tendría una influencia directa en la escalera), no obstante una función que evalúe la diagonal de la escalera (y sus adyacentes) detectaría que el patrón no es aplicable.

Ambas situaciones son bastante complejas, en el primer caso porque la complejidad de encajar el patrón aumenta al ser un patrón tan grande, y en la segunda porque este tipo de funciones recorren el tablero varias veces buscando cadenas en el trayecto de la escalera y su influencia en la misma.

- **Patrones de división de zonas.**

Intentar hacer cálculos al nivel de tablero completo es extremadamente ineficiente, debido a ello tenemos que realizar divisiones en subjuegos suficientemente pequeños y relevantes como para que los algoritmos de búsqueda obtengan resultados útiles en tiempos reducidos.

Por ello se han usado patrones de división de zona, que son unos pequeños patrones (de 4 ó 6 posiciones) que nos ayudan a distinguir las zonas en que cadenas de distinto color están en conflicto, y por lo tanto, zonas donde va a haber un conflicto potencial en el juego.

Así obtenemos los siguientes patrones de división:

Contactos elementales.

Este tipo de patrón se usa para encontrar conexiones básicas entre cadenas de distintos colores, marcando así relaciones entre ellas que nos permitirán definir la zona.

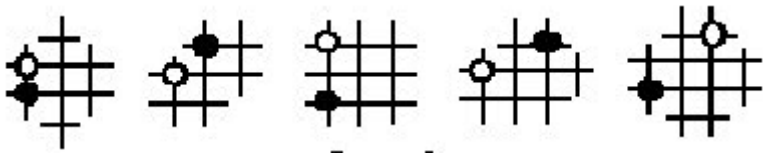

Fig. 46: Contactos elementales

Conectores y divisores.

Son patrones que nos indican el nivel de cohesión de varias cadenas.

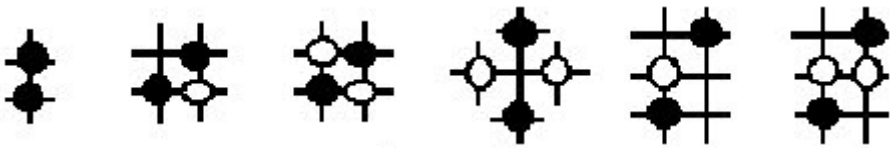

Fig. 47: Conectores y divisores

Aplicando ambos tipos de patrones de manera iterativa, logramos una división en zonas del tablero de la siguiente manera:

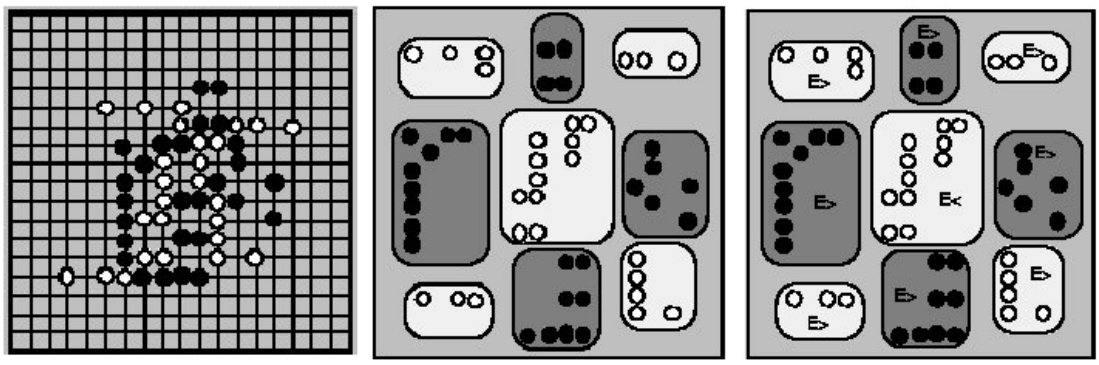

Fig. 48: División del tablero por patrones 1

De la situación inicial reconocemos los patrones consecutivamente para obtener conexiones y divisiones en grupos de piedras de cada color, y nos detenemos cuando ya no podamos dividir más en grupos de cadenas de un mismo color, definiendo así zonas monocromáticas.

Ahora aplicaríamos los patrones de contacto (que implican distancia entre cadenas) para buscar la relación entre estas zonas monocromáticas. Al hacerlo obtenemos lo siguiente:

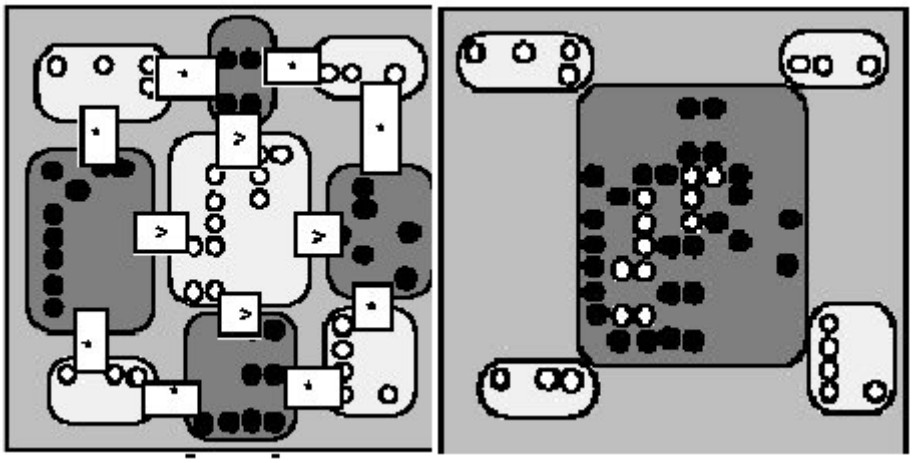

Fig. 49: División del tablero por patrones 2

Podemos observar como nos han quedado cinco zonas claramente diferenciadas, que son verdaderamente las interesantes.

Ahora cada zona con cadenas de ambos colores (en el ejemplo únicamente la central) será la que denotemos como zonas de conflicto.

- **Patrones de final de partida.**

Cuando uno de los subjuegos llega a un punto en el que el problema es conocido, entonces se puede aplicar el patrón correspondiente y solucionarlo sin necesidad de seguir haciendo cálculos que ya no son necesarios, ahorrando así tiempo de ejecución para el resto de problemas.

Por ello se tiene una base de datos de patrones con problemas resueltos en tableros limitados. Cada vez que se reconoce uno de esos patrones se aplica inmediatamente obteniendo de la base de datos el mejor movimiento y su puntuación.

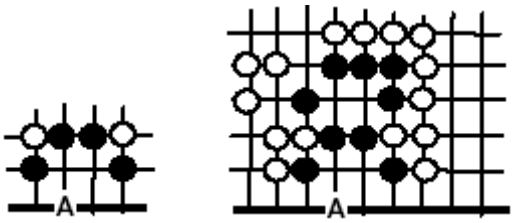

Fig. 50: Patrones de final 1

Aquí podemos observar un par de patrones clásicos, siendo 'A' el punto donde se debe jugar. Además del patrón hay que introducir en este tipo de patrones el jugador activo, ya que un patrón puede casar para un jugador y dar vida incondicional, pero para el otro provocar una muerte incondicional.

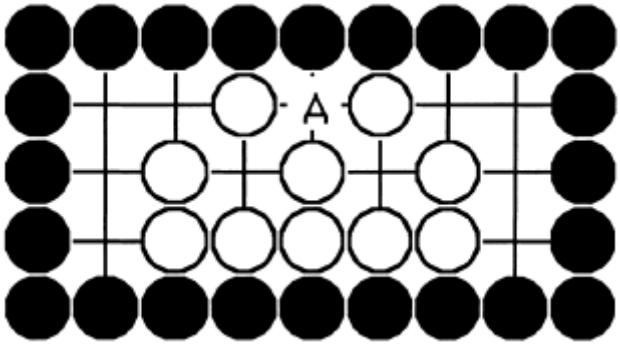

Fig. 51: Patrones de final 2

En este nuevo ejemplo lo vemos claro, A es la posición en la que debe jugar blancas para lograr vida incondicional, pero es también el punto donde debe jugar negras para matar el grupo de blancas. Por ello es necesario incluir esta información para una aplicación eficiente de patrones de Vida y Muerte o de Final de Partida (Ambas denominaciones son correctas).

Si se usase, como se ha comentado al principio, un sistema distribuido con una o varias máquinas dedicadas a patrones en tableros divididos en subjuegos permitirá descubrir más eficientemente algunos patrones.

La evaluación local, por su parte, colaboraría en la búsqueda de patrones locales que pudieran modificar, tanto las jugadas realizadas a continuación, como proporcionar información de poda al algoritmo minimax que se está llevando a cabo sobre esa zona.

Como esta posibilidad implica una base de conocimiento muy grande de la que disponíamos se ha estudiado con muy buenas perspectivas la posibilidad de incluir la construcción de ésta por un posible módulo de aprendizaje (Ver descripción del módulo de aprendizaje).

6.3. Descripción del sistema de aprendizaje

A la hora de introducir aprendizaje en el juego del Go a partir del desarrollo actual, se presentan tres grandes posibilidades:

- Mejorar la función de evaluación mediante ajuste de coeficientes.
- Crear una base de conocimiento que permita mejorar las podas en los árboles de búsqueda.
- Mejorar los patrones, aprendiéndolos por sí mismo.

Con estas premisas vamos a estudiar con detalle cada una de ellas:

- **Aprendiendo a mejorar la función de evaluación.**

  La función de evaluación básica usada en este proyecto se basa en calcular un valor en función de varios parámetros. La idea ahora es cambiar lo que influye cada uno de ellos mediante ciertos coeficientes, que se ajustan con un entrenamiento adecuado.

  Para realizar el entrenamiento no parece muy adecuado usar partidas completas, ya que la función de evaluación va a ser usada para evaluar jugadas aisladas y problemas locales. Por ello es mucho más conveniente en nuestro caso realizar el ajuste de coeficientes mediante pequeños problemas con solución conocida.

Con este método lo que logramos es que la función pondere lo mejor posible la influencia del territorio conquistado, piedras capturadas y piedras colocadas en subproblemas. Dado que (como ya se ha discutido, ver Sección 6.1) la aproximación más optima es la basada en problemas locales, de este modo la evaluación va a estar ajustada a esta clase de problemas que es donde va a ser usada.

La idea es usar una función de evaluación parametrizada, como es nuestro caso (ver Sección 5.4), de manera que se asocia a cada parámetro un peso. En lugar de fijar estos pesos de manera inamovible (como hacemos sin aprendizaje), usamos entrenamiento para que sea el propio sistema el que los ajuste. De esta forma el sistema irá mejorando su capacidad de evaluar las jugadas con la experiencia.

En el caso que nos ocupa, el conjunto de ejemplos de entrenamientos serán pequeños problemas de Go ya resueltos, como por ejemplo nuestra base de datos de patrones. Así podrá ajustar esos valores de la forma más conveniente.

- **Realizando mejores podas en el Minimax**

Es posible incluir una base de conocimiento que incluya técnicas o estrategias básicas del Go, y que, mediante un pequeño algoritmo de entrenamiento, el programa la complete y la mejore, depurando las jugadas de las técnicas que se utilicen en las partidas.

Se trataría de un sistema de reglas aplicables en el generador de movimientos y/o en la poda α-β. De este modo se reduce el coste en tiempo y espacio (en el caso de la poda) y los movimientos redundantes o triviales (en el caso del generador de movimiento).

Esta opción es muy extensa y puede ir desde simples reglas que poden en función de la experiencia (no repetir jugadas con finales malos) hasta un modelado completo con búsqueda por objetivos y metas.

Con estas técnicas conseguimos un sistema estratégico que nos permitiría alcanzar mayor profundidad con la búsqueda minimax, ya que las reglas son utilizadas para podar caminos que, a priori, sabemos incorrectos. Además nos permite dirigir la búsqueda de acuerdo con ciertos objetivos que resultan invisibles para un minimax puro de profundidad limitada.

- **Mejorando los patrones.**

Como ya se ha visto, existen patrones fijos que se han de introducir en la base de datos. No obstante existe otro tipo de patrones que se basan en problemas resueltos. Esta base de patrones puede ser aumentada con cada ejecución.

Cada vez que el sistema resuelve un problema que no estaba entre los patrones, puede recordarlo y almacenarlo, de modo que la próxima vez que se encuentre en la misma situación no necesita volver a calcularlo.

Cada vez que se consigue solucionar un subproblema, este se transforma en un patrón que pueda ser aplicado, obteniendo así la próxima vez que lo encontremos el movimiento y su puntuación automáticamente.

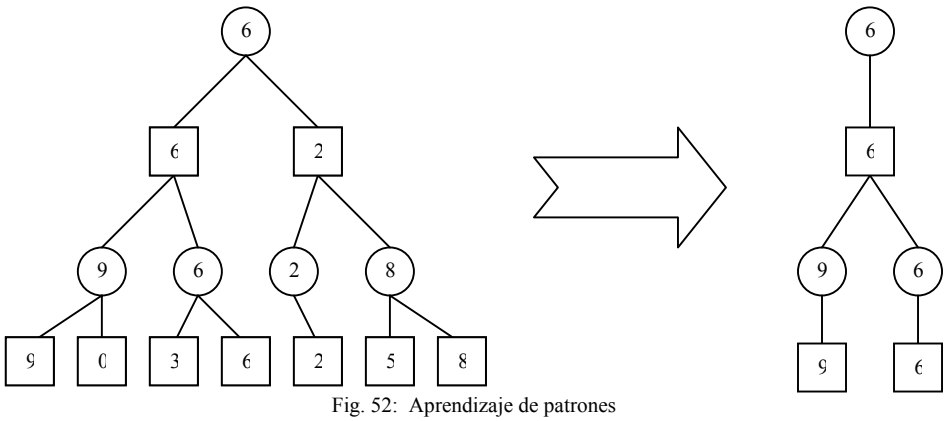

Fig. 52: Aprendizaje de patrones

En este ejemplo de Minimax, observamos el funcionamiento básico del sistema de aprendizaje de patrones. A la izquierda vemos el árbol minimax que podría haber generado el sistema de búsqueda para un problema concreto. En función del mismo, se

ha inferido el árbol de la derecha, en el que las jugadas de la máquina se han convertido en fijas, con la idea de que siempre juegue hacia la mejor opción.

Para que este método funcione el último nivel del minimax debe ser terminal, esto es, todas ellas son auténticas hojas. Esto se logra, bien cuando el tablero correspondiente a este subjuego se ha llenado completamente, bien cuando uno de los dos jugadores ha capturado todas las piezas del otro jugador en esta zona.

Con estas premisas podemos afirmar que nuestro árbol simplificado es perfectamente sustituible por el árbol completo para este problema concreto, y que siempre obtendremos al menos la mejor jugada que el minimax hubiera encontrado.

De este modo en un momento cercano al final de partida (cuando es posible que haya hasta 15 subproblemas pequeños en el tablero) se puede ahorrar tiempo, porque una parte de éstos ya está resuelta y se conoce qué evaluación conlleva.

6.4. Enfoques alternativos al minimax

- **Aplicación de redes neuronales.**

  Las redes neuronales son de una tremenda utilidad para el reconocimiento de todo tipo de patrones y, una vez entrenadas, son extremadamente eficientes. El empleo y entrenamiento de una red neuronal para realizar búsquedas de patrones, tanto ya creados como en estado de creación en el tablero reduciría bastante el tiempo de proceso que conlleva el análisis del tablero de Go en busca de cierto patrón extraído de una base de datos. Los jugadores humanos realizan esas búsquedas de patrones, mediante las cuales son capaces de eliminar posibles jugadas y de considerar como más interesantes otras que construyen patrones propios o que tratan de destruir o de impedir la construcción de los del adversario. Estos hechos llevan a considerar que existen ciertos patrones o combinaciones de piedras deseables y otros que deben ser evitados, por lo que la función de evaluación debería considerar estas cuestiones en orden a una mejor evaluación de la posición actual y de los movimientos siguientes.

El único problema derivado del empleo de las redes neuronales para este trabajo es la cantidad de tiempo que se tardará en entrenar la red y la imposibilidad de añadir patrones nuevos después del entrenamiento, puesto que se requeriría un nuevo entrenamiento con el patrón específico a reconocer, además de un refuerzo adicional de los otros patrones ya reconocidos.

Estos problemas pueden ser reducidos si se utiliza un módulo de aprendizaje que mantenga la red en continuo entrenamiento tras las partidas. El entrenamiento supervisado dependerá tanto de la función de evaluación, que encontrará interesantes ciertos patrones por sí misma, como del jugador o un experto, que introducirá nuevos casos de entrenamiento para la red y supervisará el aprendizaje.

Una estimación para una búsqueda en un tablero completo podría ser de 3 capas completamente conexas de 361 neuronas cada una, aunque después del entrenamiento podría reconocerse como demasiado compleja o quizá como insuficiente. Las estimaciones realizadas con redes neuronales en cualquier problema indican que un gran número de capas no genera un gran aumento en la eficiencia, pero el problema del Go es tan complejo que el número de neuronas de las capas ocultas, así como el número de capas podría variar según el número de patrones que quieren ser reconocidos por la red y de la complejidad o simplicidad de los mismos.

Para el paradigma de programación distribuida propuesto anteriormente, existe la posibilidad de hacer una red neuronal más pequeña, que sea capaz de reconocer patrones pequeños y que pueda ser aplicada en una zona pequeña del tablero, realizando la exploración en intervalos.

La realización de esta mejora conllevaría un mejor aprovechamiento del tiempo de proceso, así como una mejor evaluación de las posiciones, dando como resultado una mejora en el nivel de juego del programa.

- **Aplicación de algoritmos genéticos.**

Una aproximación no simbólica que podría resultar interesante es la dada por la programación genética. Se basa en interpretar parte del problema como un cromosoma, generar una población de soluciones y seleccionar la mejor haciéndolas evolucionar.

Existen varias posibilidades de aplicación:

- Sólo a la función de evaluación.
- Para entrenar una aproximación en redes neuronales.
- Buscar una solución sobre el problema completo.
- Para buscar la solución a subproblemas de Go.

Cada una de estas cuatro aplicaciones es diferente y consigue unos resultados distintos.

Cuando este paradigma es aplicado tan sólo a la función de evaluación, lo que hacemos es ajustarla de manera que sea mejor. Es un método parecido al que ya hemos descrito en el sistema de aprendizaje. Usamos distintos valores de los coeficientes como cromosomas y evolucionamos la población hasta que la función de evaluación nos dé mejores resultados.

El problema de esta aproximación es que, aplicados de manera tan local, los algoritmos genéticos tardan más en ajustar la solución que un sistema de aprendizaje clásico, por lo que no nos aportan ninguna mejora.

En el segundo caso, la mejora frente a un entrenamiento clásico sí es relevante, no obstante tenemos el problema de que las redes neuronales no han demostrado una mejora relevante en cuanto a la evaluación de tableros completos y descubrimientos de patrones en los mismos se refiere (ver el punto anterior sobre redes neuronales para una explicación detallada de su aplicación a Go). Por lo que aquí tampoco nos ofrecen una mejora relevante los algoritmos genéticos.

La tercera posibilidad empieza a resultar interesante. El tablero de Go, a diferencia de otros juegos, se presta muy bien a una representación como cromosoma (cada gen tiene tres estado blanco, negro o vacío). Ahora bien, la complejidad de un algoritmo genético y su optimalidad están relacionadas directamente con los tamaños del cromosoma y de la población. En este caso tenemos un cromosoma enorme (361 genes cada uno), esto nos obliga a generar una población igualmente grande para obtener resultados relevantes, ya que de otro modo la población no convergería hacia una solución óptima.

Además, para realizar la evaluación de cada solución generada y saber cual es mejor habría que comprobar a qué resultado se llega, de manera parecida al método que usa el Gobbler (Monte Carlo Go). Como ya se demuestra en este sistema, el tamaño máximo de tablero para lograr resultados aceptables es de 9x9, por lo que no es eficiente en tableros completos.

La cuarta propuesta utilizaría los algoritmos genéticos como alternativa a la búsqueda minimax, dejando el resto del sistema tal y como esta descrita en este documento. Es decir, aplicamos patrones de inicio, división de zonas y después se plantea la búsqueda en cada zona. La idea es aplicar algoritmos genéticos solamente a la solución de pequeños problemas de Go, donde sí sea eficiente el tamaño de cromosoma y la evaluación de individuos.

A todo esto hay que añadir que la conversión de una solución a un subproblema en patrón de final de partida es inmediata, ya que el algoritmo genético ya la representa como una secuencia de valores del tablero. Lo cual puede hacer que sea aún más eficiente.

En su contra se debe comentar que cuando se llega demasiado pronto al juego medio (y la densidad de piezas en el tablero es muy baja) tendríamos el mismo problema que cuando queremos aplicar los algoritmos genéticos al tablero completo y podría ocurrir que se enfocase la partida desde el inicio incorrectamente, siendo después imposible de solucionar aplicando técnicas convencionales para la obtención de un buen resultado en el final de partida.

7. Conclusiones

Este proyecto no ha concluido obteniendo una aplicación de juego de Go con un nivel de juego comparable a los programas de Go de gama media. La duración limitada a un año del proyecto impedía que la implementación de un juego de Go fuera factible. Calculamos que se hubiera necesitado entre uno y tres años más poder podido llegar a producir una aplicación completa que aprovechara las técnicas aquí expuestas. Esto era esperable y ya se había contemplado en los objetivos de nuestro proyecto.

Usando como base este proyecto se conseguiría una aplicación capaz de ejecutarse de forma distribuida, aplicando las ideas contenidas en la ampliación basada en RMIs (ver Sección 6.1). Aunque no pudimos desarrollar suficientemente el sistema para poderse hacer efectivo, se consiguió una arquitectura que pudiera ser adaptada fácilmente a esta implementación. Los algoritmos y estructuras de datos básicos descritos e implementados a lo largo de este proyecto podrían ser adaptados con facilidad a la ejecución distribuida, o buscando mayor eficiencia se podrían tomar los algoritmos distribuidos con que se puede ampliar el sistema (ver Sección 6.1). En conclusión, consideramos que hemos conseguido encontrar técnicas, algoritmos y arquitecturas adecuadas para confeccionar un sistema distribuido de Go.

A lo largo del proyecto creemos que hemos abarcado suficientes técnicas distintas como para que este documento pueda servir como muestra de la mayor parte de las metodologías de investigación en inteligencia artificial aplicadas a Go. Se puede tomar la documentación generada, las ideas y las consideraciones contenidas en este proyecto como panorámica de partida para la realización de otras investigaciones y proyectos. Consideramos que en este aspecto la investigación realizada a lo largo de este proyecto ha conseguido ser completa, actual y abarca la mayor los aspectos relevantes del problema.

La arquitectura descrita en este documento (ver Sección 5.1) creemos que es suficientemente flexible como para poder emplearse en otros sistemas de Go con no demasiadas modificaciones. Las modificaciones más pertinentes serían fundamentalmente particularizar ciertos los detalles que fueran específicos del sistema objetivo y desarrollarlos con mayor profundidad.

Las estructuras de datos desarrolladas en este proyecto (ver Sección 5.5) son suficientemente potentes como para dar soporte a los algoritmos descritos. Si bien es cierto que nos hemos decantado más por la eficiencia espacial que por obtener estructuras reducidas en espacio, creemos que la

memoria es un recurso mucho menos importante que el tiempo porque la ejecución distribuida disminuiría mucho las limitaciones espaciales que surgen de emplear estructuras de datos complejas y tener mucha información repetida en ellas. Tal vez sería necesario realizar un estudio mucho más profundo para reducir el espacio requerido, pero los cambios resultantes en las estructuras del sistema no serían demasiado grandes y se referirían sobre todo a qué datos deben mantenerse y duplicarse en la ejecución del minimax.

El algoritmo de minimax que hemos desarrollado (ver Sección 5.3) es una aproximación extremadamente simple. Sin embargo consideramos que las ampliaciones y modificaciones propuestas (ver Secciones 6.1, 6.2 y 6.3) incluyen suficientes variantes como para poder elegir una implementación más compleja y eficiente de minimax a partir de ellas.

Los parámetros de que consta la función de evaluación (ver Sección 5.4) son bastante básicos. La función de evaluación se puede completar añadiendo nuevos parámetros, se puede complicar empleando parámetros más precisos y sofisticados y sobre todo investigar aprendiendo los pesos de sus componentes mediante técnicas de aprendizaje (ver Sección 6.3) pero consideramos que la que se ha propuesto es sencilla, eficaz y contempla los principales aspectos del juego.

El reconocimiento de patrones debería haber sido implementado en nuestra aproximación a la solución por la importancia que tiene, sin embargo la limitación en tiempo nos ha impedido trabajar más sobre este tema.

De hecho de las expansiones que se sugieren en este proyecto el punto que quizás debería haber sido desarrollado con mayor profundidad es el relativo a los patrones, tanto los aspectos de reconocimiento como los relativos a la base de dato de patrones concreta. No pudo ser así porque la introducción de una base de datos de patrones no sólo implica el diseño de la misma, sino introducir un número suficiente de patrones, lo cual era realmente imposible dados los límites de tiempo de este proyecto. Además Go emplea patrones muy distintos para aspectos muy distintos de juego, de modo que, muy probablemente, no habría que diseñar y rellenar una única base de datos de patrones, sino varias específicas a distintos problemas y, por lo tanto, habría que dedicar una cantidad de tiempo muy considerable al tema, tiempo que no teníamos para realizar este proyecto.

En resumen se puede concluir que los puntos fuertes de este proyecto son la amplitud de miras en cuanto a las fuentes que se emplearon para su desarrollo y el hecho de que se enfrenta a las

cuestiones principales del problema de desarrollar una aplicación que juegue a Go, teniendo siempre como contexto un sistema de juego distribuido.

8. Bibliografía

1. Benson, D, Live in the game of go, Information Science 10 (1976) 17-29.
2. Berlekamp,E , Conway, J.H., Guy, R.K., Winning Ways, Academic Press, New York, 1982.
3. Björson, Y, Marsland, T, Multi cut alphabeta-pruning in game tree search, Theoretical Computer Science 252 (2001) 177-196.
4. Bouzy, B, Cazenave, T, Computer go: an AI oriented survey, Artificial Intelligence 132 (2001) 39-103.
5. British Go Asociation, How to play Go, http://www.britgo.org/intro/intro2.html (2002).
6. Brügmann, B, Monte Carlo go, bruegman@iws170.mppmu.mpg.de (1993).
7. Chen K, Chen Z, Static analysis of life and death in the game of go, Information Science 121 (1999) 113-134.
8. Chen, K.H., Soft descomposition search and binary game forest model for move decision in go, Information Sciences 154 (2003) 157-172.
9. Conway, J.H., On Numbers and Games, Academic Press, London, 1976.
10. Fotland, D., Knowledge Representation in the Many Faces of Go, ftp://bsdserver.ucsf.edu/Go/comp/mfg.Z (1993).
11. Gao, X, Iida, H, Uiterwijk, J.W.H.M., van den Herik, H.J., Strategies anticipating a difference in search depth using opponent-model search, Theoretical Computer Science 252 (2001) 83-104.
12. Goldis, B, Towards abstraction in go knowledge representation, Advanced Artificial Intelligence CAP 6680 (1999).
13. Inuzuka N, Fujimoto H, Nakano T, Itoh H, Prunning nodes in the alpha-beta method using inductive logic programming, http://citeseer.ist.psu.edu/198154.html (1999).
14. Kao, K, Sums of hot and tepid combinatorial games, Ph.D. Thesis, University of North Carolina at Charlotte, 1997.
15. Kao, K.Y., Mean and temperature search for go endgames, Information Sciences 122 (2000) 77-90.
16. Kojima, T, Yoshikawa, A, Knowledge Acquisition from Game Records, http://citeseer.ist.psu.edu/kojima99knowledge.html (1999).
17. Mechner, D, All systems go, The Sciences 38 Nº 1 (1998).

18. Mechner, D, Klinger, T, An architecture for computer go, http://cns.nyu.edu/~mechner/compgo/ (1996).
19. Müller, M, Computer go, Artificial Intelligence 134 (2002) 145-179.
20. Müller, M, Gasser, R, Experiments in computer go endgames, Games of No-Chance 29 (1996) 273-284.
21. Müller, M, Global and local game tree search, Information Science 135 (2001) 187-206.
22. Müller, M, Pattern matching in Explorer. Extended abstract, En *Proceedings of the Game Playing System Workshop*, pages 1-3, Tokyo, Japan, 1991. ICOT.
23. Plaat A, Research, Re: Search and Re-Search, http://citeseer.ist.psu.edu/plaat96research.html (1996).
24. Reitman, W, Wilcox, B; Pattern Recognition and Pattern-Directed Inference in a Program for Playing Go, Pattern-Directed Inference Systems (D. Waterman and F.Hayes-Roth, Eds). Academic Press (1978).
25. Uther, W, Veloso, M, Adversarial Reinforcement learning (1997).
26. van der Steen, J, The Rules of Go, http://gobase.org/studying/rules/doc/a4.pdf (1995).
27. Willmott, S, Richardson, J, Bundy, A, Levine, J, Applying adversarial techniques to go, Theoretical Computer Science 252 (2001) 45-82.

Apéndice: Breve historia del Go

Los orígenes

Los orígenes del Go se desconocen a ciencia cierta. El juego procede de China y se cree que fue creado entre el 3000 aC y el 2000 aC. Algunos piensan que era originariamente un precursor del ábaco, otros, que las piedras blancas y negras (en representación del ying y el yang) se utilizaban como medio para adivinar el futuro. Una famosa leyenda habla del un mítico emperador que creó el juego para mejorar la inteligencia de su no muy brillante hijo.

Fuera como fuera, en tiempos del filósofo Confucio (600 a.C aproximadamente), el Go era una de las "Cuatro Aptitudes" (junto con la pintura, la poesía y la música) que debía dominar cualquier intelectual en la antigua China.

Merced a la formidable influencia cultural china, el Go (Weiqi o Wei-chi para los chinos) pasó a Corea (donde se le conoce como Baduk) y Japón (donde adquirió el nombre de Go). En China permaneció como entretenimiento de las clases pudientes, como se puede observar en el arte Chino, en donde ocasionalmente aparecen representados nobles (tanto hombres como mujeres) jugando a Weiqi.

Go en Japón

El Go está presente en Japón con total seguridad desde al menos el año 1000, porque se alude a él en la obra de Murasaki *El Relato de Genji*. El Go tuvo un impulso formidable alrededor del 1600, cuando el señor de la guerra Tokugawa decretó la creación de Cuatro Escuelas. Cada año, representantes de las cuatro escuelas competirían entre sí y el ganador obtendría el título de go-doroko (ministro de Go) para el resto del año. Este sistema elevó enormemente la popularidad del Go en Japón.

Durante la restauración Meiji a finales del siglo XIX, el Go cayó en un período de relativa crisis, pero volvió a la vida en 1920 con la formación de la Asociación Japonesa de Go. Los periódicos empezaron a patrocinar torneos y hoy en día hay más de una docena de títulos de gran importancia, con columnas y análisis de partidas en las principales publicaciones de prensa diaria. Los mejores jugadores japoneses de Go se convierten en celebridades.

Go en China en la actualidad

Durante la Revolución Cultural China, el Go estaba perseguido, por ser considerado un "pasatiempo burgués" y los jugadores tenían que citarse en secreto. A pesar de ello, en el año 1978 se estableció un sistema profesional moderno y pocos años más tarde se crearon las Súper Series de Go Japón-China, un certamen anual en que los mejores jugadores de ambos países compiten en un torneo eliminatorio.

Go en Corea en la actualidad

El sistema profesional coreano se estableció en la década de l950, cuando Cho Nam-chui regresó de su entrenamiento como profesional en Japón. Hoy en día Corea es quizás el lugar del mundo donde el juego del Go tiene mayor popularidad (con permiso de los japoneses). Se estima que entre el 5 y el 10 por ciento de la población coreana juega al Go de manera regular.

Al igual que en Japón, hay muchos torneos patrocinados por la prensa en Corea, con gran seguimiento popular. En los últimos años algunos de los mejores jugadores coreanos han logrado grandes victorias en las competiciones internacionales.

Las competiciones internacionales de Go

Hasta los últimos 20 años, las competiciones entre maestros de Go de diferentes países eran prácticamente desconocidas. Los últimos 10 años han conocido una enorme proliferación de campeonatos internacionales, donde los grandes jugadores de Japón, China y Corea, así como del resto del mundo, compiten para ser vistos como el mejor jugador de Go. La Copa del Mundo Ing, que se celebra cada cuatro años y tiene un primer premio de un millón de dólares, encabeza la lista de torneos, que incluyen la Copa Fujitsu y la Copa Dongyang Securities.

Otras competiciones mundiales a escala amateur incluyen el Campeonato del Mundo Amateur de Go, organizado por la Federación Internacional de Go, el Campeonato Juvenil de Go, organizado por la Fundación Educativa de Go que se encuentra en Taipei, el Campeonato Femenino

de Go organizado por la IGF y el Campeonato para equipos mixtos, patrocinado por el Club de Go La-LaLa de Japón.

Índice de figuras