Universitat Politècnica de Catalunya – ETSETB
Proyecto de Final de Carrera

Desarrollo del módulo de resolución de semeai Semeai-01ES

? de septiembre de 2004

Ricard VILÀ
ritx@ya.com

bajo la dirección de Marco A. PEÑA
y de Tristan CAZENAVE

Université Paris 8
2, r. de la Liberté
93526 Saint-Denis Cedex

ETSETB, UPC
C/ Jordi Girona, 1-3
08034 Barcelona

A mi padre, por ser un modelo
de lo que quiero llegar a ser.

Índice general

Prefacio

> Un libro abierto es un cerebro que habla;
> cerrado un amigo que espera;
> olvidado, un alma que perdona;
> destruido, un corazón que llora.
> Proverbio hindú

El presente trabajo es el resultado de seis meses de investigación en el laboratorio de Inteligencia Artificial de la universidad Paris VIII bajo la dirección de Tristan Cazenave. Tras encontrar por azar su página web fui a visitarle en el mes de octubre de 2002. Me aceptó como estudiante y en el mes de febrero me incorporé al grupo de juegos del laboratorio de IA. Han sido seis meses intensos, que han pasado como un soplo de viento, en los que he tenido la oportunidad de estar con unos compañeros inigualables que me han enseñado casi tanto como cinco años de universidad.

En este medio año, además de encontrar algún que otro hueco para descubrir los cafés donde se desarrollaron las más intensas tertulias artísticas del XIX, las cuestas de Montmartre, las calles llenas de vida del Marais o la curiosa costumbre parisina de hacer un picnic en el primer metro cuadrado de hierba que encuentren en la selva de asfalto, he tenido la oportunidad de aprender muchas cosas que, sin estar del todo relacionadas con Computer Go, forman parte del oficio. He dado la bienvenida a Linux (ya soy más libre), he cambiado al Developer Studio por el cóctel gcc-make, he descubierto lo útil que puede llegar a ser citeseer y me he enamorado de LaTeX.

Como resultado de este trabajo tuve la oportunidad, en noviembre de 2003, de estar en la «10th Advances in Computer Games Conference» celebrada en Graz. Allí puede conocer un poquito más a fondo el mundillo y los protagonistas de este apasionante reto que es Computer Go.

En cuanto al proyecto, he adoptado algunas convenciones tipográficas que he intentado respetar a lo largo de toda la memoria. Los términos técnicos de Go, normalmente en japonés, aparecen así: *término*, y ello nos indica que podemos encontrar su definición en el apéndice C. Los términos relativos a nombres de

variables y funciones del código de *Semeai-01ES* aparecen así: `término`. Por último, excepto en casos donde la traducción es obvia y no implica pérdida de matices, he optado por no traducir los términos técnicos de Computer Go que aparecen en inglés en la bibliografía, simplemente los he resaltado en *cursiva* .

¿Cómo leer esta obra? El Go es un juego minoritario y Computer Go es a su vez un dominio de investigación minoritario. Para el lector profano en la materia se recomienda iniciar la lectura por el apéndice A y el capítulo 2. Al lector familiarizado con la materia se le recomienda una lectura lineal. Si por motivos de tiempo se desea hacer una lectura rápida el grueso del proyecto se encuentra en los capítulos 3, 4 y 5.

El proceso de redacción se ha desarrollado a caballo entre Barcelona y el Solsonès entre los meses de febrero y abril de 2004.

Barcelona, abril de 2004

Parte I

Preliminares

Capítulo 1

Introducción

El ir un poco lejos es tan malo
como no ir todo lo necesario.
Confucio

Un *semeai* es una carrera de libertades entre dos grupos completamente rodeados, cuya única manera de vivir es matando al adversario. Tras esta, aparentemente, sencilla y concisa definición se esconden multitud de situaciones de alta dificultad incluso para jugadores humanos.

Nuestro objetivo en este proyecto ha sido desarrollar un módulo autónomo capaz de analizar satisfactoriamente el mayor número posible de situaciones calificadas como *semeai*.

En la primera parte de esta memoria presentamos, en el capítulo 2, una breve descripción del estado del arte en Computer Go basada en el completo artículo de Cazenave y Bouzy [9]. En el capítulo 3 definimos de manera clara cuál es el problema abordado y qué objetivos nos proponemos alcanzar. Discutiremos la importancia que tiene dicho problema y nos detendremos brevemente a examinar las enormes dificultades que puede plantear.

En la segunda parte, describimos la solución propuesta y los resultados obtenidos. En el capítulo 4 empezamos describiendo nuestro punto de partida; el trabajo de Müller y Cazenave nos permitió iniciar este trabajo con una considerable ventaja. A continuación desmenuzaremos las entrañas del corazón de este proyecto: el módulo *Semeai-01ES* . Veremos su arquitectura general y los detalles de cada una de las clases que lo integran. Dedicaremos especial atención a la clase Eye, sus fundamentos teóricos fueron publicados en la «10th Advences in Computer Games Conference» celebrada en Graz en Noviembre de 2003 [35]. En la sección 4.2.1 hemos incluido la traducción de ese artículo.

En el capítulo 5 veremos la colección de tests de *semeai* creada para poner a prueba el módulo desarrollado y una comparativa de los resultados que obtiene respecto a GnuGo, el vigente campeón de la *Computer Olympiad*.

El tiempo, verdugo inexorable de las expectativas, nos obliga a incluir el capítulo 6, en él indicamos posibles caminos a seguir para retomar este proyecto en el futuro.

Finalmente, en la tercera parte, hemos incluido cinco apéndices. En el apéndice A hemos incluido un pequeño resumen de la historia y las reglas del juego del Go para los lectores profanos en la materia. A continuación describimos el formato SGF, la manera más extendida de almacenar de manera electrónica una partida de Go. En el apéndice C presentamos un pequeño glosario de los términos técnicos de Go utilizados a lo largo de la presente obra. En el apéndice D mostramos las distintas clases de equivalencia de los poliominoes de tamaño 5, 6 y 7 según la *Neighbour Classification* definida en la sección 4.2.1. Por último, en el apéndice E mostramos información detallada sobre los resultados obtenidos por *Semeai-01ES* frente a la colección de tests STS-RV.

Capítulo 2

Computer Go: Estado del arte

Amo a los que sueñan con imposibles.
J. Goethe

Los juegos de habilidad mental como el ajedrez han sido considerados desde el principio de la inteligencia artificial (AI) como campos de aplicación de gran relevancia. Actualmente hay programas mejores que los jugadores humanos en la mayoría de juegos clásicos: ajedrez, damas, damas francesas, othello, gomoku,...

Los métodos estándar de la AI para la resolución de otros juegos (función de evaluación más exploración Alpha-Beta global), aún siendo útiles para los programas de Go, no permiten, por sí solos, la construcción de un programa de Go de alto nivel. Los problemas relacionados con Computer Go , dada su abundancia y la variedad de posibles soluciones, exigen nuevos métodos de resolución y hacen de este campo un interesante dominio de investigación para la AI.

2.1 Go, el rey de la complejidad

Los resultados en Computer Games han sido espectaculares en tan sólo cuarenta años. En un gran número de juegos la habilidad e inteligencia humanas no pueden ni tan siquiera rivalizar con la potencia de cálculo de las máquinas. Sin embargo, el Go sigue siendo una asignatura pendiente y los mejores programas actuales no llegan a alcanzar, ni tan siquiera, el nivel de un jugador aficionado medio de club. ¿Por qué? Intentaremos dar respuesta a esta pregunta en la sección 2.1.1 pero veamos antes los avances en otros juegos:

- **Go-moku** se juega en una cuadrícula de tamaño variable y el objetivo consiste en colocar cinco piezas en línea horizontal, vertical o diagonalmente. Ha sido computacionalmente resuelto en su variante más simple por L.V. Allis [1].

- **Othello** también llamado reversi es otro juego completamente dominado por la máquina. *Logistello* desarrollado por M. Buro es el programa más fuerte actualmente. Tras ganar numerosos torneos entre programas desafió en agosto de 1997 al campeón mundial Murakami. *Logistello* ganó el desafío de manera incontestable 6-0.

- **Damas.** Desde 1988 el grupo de computer games de la universidad de Alberta está desarrollando el programa *Chinook*. Este programa es el actual campeón del mundo de computadores y ha competido en una ajustada final con el campeón del mundo humano. En 1993 completaron la computación de las bases de datos de 2 a 8 piezas en el juego de damas (checkers). En la «10th Advances in Computer Games Conference» de 2003 presentaron la finalización de la base de datos de 9 piezas y el inicio de la de 10 dando como resultado una base de datos de 13 trillones de posiciones y 148 GB de datos comprimidos en un intento de resolver el juego definitivamente [24].

- **Ajedrez.** Los inicios de computer chess se remontan a 1950. Desde entonces la investigación en este campo ha sido exhaustiva y hay un claro paradigma de aproximación al problema: extensa búsqueda en árbol y Alpha-Beta. En mayo de 1997 *Deep Blue*, IBM hardware y software, derrotó al campeón del mundo Gary Kasparov 3.5-2.5 utilizando un algoritmo Alpha-Beta y explorando alrededor de un billón de posiciones por segundo.

En Go el mejor programa sigue aún a muy bajo nivel comparado con los jugadores humanos. La razón se encuentra por un lado, en la elevada complejidad teórica del juego y por otro, en lo que podríamos denominar una “complejidad inherente” al juego.

2.1.1 Complejidad teórica

Una buena clasificación respecto a la complejidad de los juegos de dos jugadores, información completa y suma cero, es la propuesta por Allis [2]. Define la complejidad del espacio de estados (E) como el número de posiciones alcanzables desde el punto de partida, y la complejidad del árbol de juego (A) como el número de nodos en el árbol más pequeño necesarios para resolver el juego.

En la tabla 2.1 presentamos, ordenados según los estándares de complejidad de Allis, algunos de los juegos anteriormente comentados. Si obviamos por un

JUEGO	$\log(E)$	$\log(A)$	MÁQUINA VS HOMBRE
Damas	17	32	Chinook $> H$
Othello	30	58	Logistello $> H$
Go 9x9	40	85	Mejor Programa $\ll H$
Ajedrez	50	123	Deep Blue $\geq H$
Go 19x19	160	400	GnuGo 3.4 $\ll H$

Tabla 2.1: Complejidad de los juegos según los estándares de Allis

momento la fila de Go 9×9, es fácil observar una correlación entre la complejidad y la inferioridad de las máquinas contra el hombre (H simboliza el mejor jugador humano). En damas, othello y ajedrez el modelo clásico de programación de juegos funciona. Consiste en la terna función de evaluación, generación de posibles movimientos y búsqueda en árbol.

Sin embargo para Go 19×19 este modelo clásico ya no es válido. Podríamos pensar en un primer momento que la razón es que el modelo clásico tiene un techo de validez en función de la complejidad y que el Go esta por encima de ese techo de validez. Pero si nos fijamos ahora en la fila de Go 9×9 veremos que ha de haber algo más. La complejidad en 9×9 es significativamente inferior a la del ajedrez y sin embargo los resultados en uno y otro se encuentran a años luz. ¿Qué más hay entonces?

Cazenave y Bouzy [9] proponen como explicación la complejidad inherente de la función de evaluación en computer go y afirman que un nuevo modelo es necesario para superar la complejidad del juego de Go.

2.2 Evolución histórica

Los inicios de Computer Go se remontan a 1960. El primer programa de Go capaz de vencer a un humano (un principiante absoluto) fue creado por Zobrist a finales de los 60 y se basaba fundamentalmente en la computación de una función que aproximaba la influencia radiada por cada piedra. Otra gran contribución de Zobrist al dominio de computer games fue la definición de un método eficiente de *hash* de posiciones.

Desde el principio de las investigaciones en Computer Go varios trabajos se han publicado sobre subproblemas del juego de Go: estudios sobre tableros de reducido tamaño [33, 34], sobre la vida y la muerte [5], sobre *semeai* [30], sobre el *yose* [7]...

El primer programa de Go que consiguió jugar mejor que un principiante absoluto fue diseñado por Bruce Wilcox y marcó la pauta de la siguiente generación de

programas: utilización de una representación abstracta del tablero y razonamiento sobre los grupos.

El uso intensivo de patrones (*patterns*) marcó el siguiente gran avance en la disciplina. Fue introducido por Mark Boon creador de Goliath.

Los programas actuales utilizan todas estas técnicas y se basan en rápidas búsquedas tácticas y en una búsqueda global. Los estudios actuales se basan en teoría combinatoria de juegos [28], aprendizaje automático [12] y en modelización cognitiva [8] entre otros.

2.3 Torneos de Computer Go

Durante muchos años el desarrollo de los programas de Go tuvo el estímulo de un suculento premio de un millón de dólares para el programador que desarrollara una máquina capaz de ganar a un niño de nueve años. Un estímulo que caducó en el año 2000 sin que nadie hubiera conseguido ni tan siquiera aproximarse. De hecho, teniendo en cuenta que en Asia podemos encontrar niños de nueve años que ya han conseguido la categoría de profesionales, el estímulo era del calibre del de la zanahoria y el caballo.

La competición de computer Go con más historia es la *Ing Cup* que se celebró regularmente desde 1987 hasta el año 2000.

La década de los noventa ha visto tres grandes programas que han dominado la escena: *Goliath* escrito por Mark Boon que ganó la *Ing Cup* en 1989, 1990 y 1991, *Go Intellect* de Ken Chen ganador en 1992 y 1994, y *Handtalk* que luego apareció como *Goemate* escrito por el profesor Zhixing Chen ganador en 1993, 1995, 1996 y 1997. Estos tres últimos años *Handtalk* se adjudicó también la *FOST Cup*.

Desde finales de los noventa hasta ahora distintos programas se han ido alternando en las primeras posiciones de los distintos torneos sin que se volviera a repetir una situación hegemónica como las anteriormente comentadas. Entre otros: *Many Faces of Go*, *Go4++*, *Wulu* y *KCC Igo*

Después de la 8.ª Olimpiada de Computer Games celebrada en noviembre de 2003 en Graz (Austria), *GnuGo 3.4* se ha erigido como el programa más fuerte del momento acabando la competición invicto. En esta competición participaron 11 programas bajo un sistema de todos contra todos. *GoAhead* y *Go Intellect* ocuparon respectivamente la segunda y tercera plaza. Algunos de los destacados ausentes fueron *Many Faces of Go* de David Fotland o *Go4++* de Michael Reiss. En la competición de Go 9x9 los tres primeros clasificados fueron: *Aya*, *NeuroGo* y *Go Intellect*.

Además de las competiciones con presencia física de programadores, computadoras y programas, hay una competición de Computer Go permanente a través

de Internet: la *Computer Go Ladder*. [1] En ella los programas están ordenados según su fuerza y el número de piedras de handicap (cf. sección A.2.5) que hay entre ellos. Cada vez que un programa presenta mejoras puede desafíar a los programas por encima suyo para mejorar su posición en la escalera.

Distintos servidores de Go por Internet (NNGS[2], IGS [3] y recientemente KGS[4]) permiten la conexión de programas al servidor a través del protocolo GTP (cf. sección 5.1) de manera que pueden jugar contra oponentes humanos. *GnuGo* es el programa que más a menudo está conectado y tiene un ránking estable en NNGS de 8 *kyu*.

[1] http://www.cgl.ucsf.edu/go/ladder.html
[2] http://nngs.cosmic.org
[3] http://igs.joyjoy.net
[4] http://kgs.kiseido.com

Capítulo 3

Problema

A partir de cierto punto no hay retorno.
Ese es el punto que hay que alcanzar.
Franz Kafka

3.1 Definición y objetivos

Golois[1] es el programa de Go desarrollado por el Dr. Cazenave durante los últimos diez años. Como la mayoría de programas de Go, *Golois* tiene sus puntos fuertes y sus puntos débiles. Entre sus puntos débiles destaca la resolución de *semeai*. *Golois* tiene a nivel estratégico, un módulo encargado de detectar la aparición de un *semeai* durante la partida, pero la resolución táctica del mismo estaba muy poco desarrollada.

Objetivo El objetivo del proyecto consiste en desarrollar un módulo autónomo encargado de la resolución táctica de *semeai* para su posterior inclusión en la arquitectura de *Golois*. De esta forma, cuando *Golois* detecte a nivel estratégico la aparición de un semeai llamará directamente al módulo de resolución de *semeai* desarrollado (en adelante nos referiremos al mismo como módulo *Semeai-01ES*) para obtener el estatus y el movimiento a jugar.

3.2 ¿Qué es un *semeai*?

En la terminología de Go un *semeai* (o carrera de libertades), define la situación que se da entre dos grupos enemigos que no pueden vivir simultáneamente. La

[1] http://www.ai.univ-paris8.fr/ cazenave/Gogol.html

única opción que tienen para vivir es matando al grupo contrario.

Como todos los conceptos en Go, tras una definición simple, se esconde un concepto extremadamente complejo y difícil de dominar incluso para un jugador humano. Prueba de ello es el ensayo recientemente publicado por Richard Hunter [22] de 200 páginas dedicado enteramente al estudio de *semeai*. Al lector no iniciado en el Go se le recomienda en este punto una lectura del apéndice A y en especial de la sección A.3.

3.2.1 Importancia de un *semeai*

La primera pregunta que se nos plantea es, ¿tiene suficiente importancia el concepto de *semeai* como para merecer un módulo exclusivo?

Cabe recordar que en Computer Go no disponemos de una función de evaluación global. Más bien se trata de un conjunto de pequeñas funciones de evaluación asociadas a objetivos locales concretos: captura, conexión, vida y muerte,... Un *semeai* es un objetivo local crucial que puede desequilibrar por completo la balanza de una partida y por consiguiente tan o más importante que cualquiera de los otros objetivos locales anteriormente mencionados.

Por otro lado sería bueno saber con qué frecuencia aparece un *semeai* en una partida para poder valorar hasta qué punto merece la pena el desarrollo de un módulo dedicado específicamente al estudio de los *semeais*.

Si nos fijamos en las partidas de profesionales nos llevaremos la ingrata sorpresa de ver que raramente aparecen. Sin embargo ello no quiere decir que no tengan importancia. La secuencia de movimientos de una partida entre profesionales es, como dice Hunter, «la punta de un iceberg formado por los cientos de secuencias calculadas por los jugadores que no llegan a concretarse en el tablero»[2]. La gran habilidad de un jugador profesional hace que rara vez inicie una carrera de libertades que no va a ganar. Es por ello que en las partidas de profesionales, los *semeais* quedan sumergidos en el 90 % del iceberg que queda bajo el agua.

Si nos fijamos, en cambio, en las partidas entre jugadores amateurs, nos daremos cuenta de que los *semeais* no sólo son significativamente frecuentes sino que además suelen determinar un final precipitado de la partida, decantando dramáticamente la balanza de la partida en el sentido del jugador que vence el *semeai*. Esto se debe a que los jugadores aficionados realizan muchos de sus movimientos intuitivamente, para «ver qué pasa», sin saber con certeza a priori el resultado final de la lucha. Habida cuenta de que el nivel de un programa de Go esta significativamente por debajo de un jugador medio de club no es difícil deducir cuán

[2] Página 158 [22]

importante puede llegar a ser para una máquina una buena gestión de las carreras de libertades.

3.2.2 Resolución de un *semeai*

Ya hemos visto y argumentado que el conocimiento sobre *semeai* es fundamental para un programa. Pero, ¿en qué consiste la resolución de un *semeai*?

Básicamente podemos definir tres fases en el proceso. En el capítulo 4 se detallarán los pormenores de cada fase.

- **Detección**: El programa debe decidir en qué momento de la partida se ha entrado en una situación de semeai. Una decisión prematura, como decir que dos grupos están en semeai cuando alguno o los dos pueden aún conectarse con otro grupo que está vivo, iniciará el módulo de *semeai* cuando en realidad son otras las alternativas a contemplar (llamar al módulo de conexión). Todo ello resultará en un consumo de tiempo innecesario para acabar obteniendo con toda probabilidad una respuesta errónea. Una decisión tardía, detectar el *semeai* con un turno de retraso, ofrece al adversario un movimiento de ventaja en la carrera. Llamar entonces al módulo nos conducirá a recibir un resultado de *derrota* la mayor parte de las veces. La correcta detección del *semeai* es un paso previo fundamental para garantizar el éxito en el combate.

- **Análisis**: Una vez detectados los dos grupos enemigos en *semeai*, el módulo procederá al análisis de la situación. Una gran cantidad de variables debe ser cotejada: el turno de juego, las libertades de cada grupo, las libertades comunes, los ojos y su estatus, los posibles *kos* (internos o externos), la posibilidad de captura de bloques no esenciales, las opciones de escapar y conectarse a un grupo vivo (si el *semeai* no está completamente cerrado), los *tesujis* para reducir o aumentar el número de libertades efectivas... La fase de análisis es el núcleo del módulo.

- **Retorno de resultado**: Realizado el análisis el módulo deberá retornar el resultado del *semeai*. El resultado de un *semeai* se compone de:

  a) Estatus: victoria, derrota, empate (ambos viven en *seki*) o *ko*.

  b) Movimiento: intersección del tablero donde jugar para garantizar el estatus o pasar.

3.2.3 Dificultades de un *semeai*

Ahora bien, ¿cuánto conocimiento requiere la resolución de *semeai*? y, aún más, ¿cuánto cuesta transmitirle ese conocimiento al programa? Mucho y mucho son las dos respuestas que me vienen de inmediato a la cabeza pero intentaremos ser un poco más concretos.

La principal dificultad de un *semeai* radica en ser parte del Go y en la imposibilidad de desligarlo del resto de los conceptos del juego. Es imposible enseñar a un humano a resolver problemas de *semeai* sin antes enseñarle a jugar a Go. Del mismo modo, una máquina que intente resolver problemas de *semeai* sin conocer nada de captura, conexión, vida y muerte, *ko* o *tesujis* difícilmente tendrá éxito. Por tanto, a priori, parece que un buen módulo de resolución de *semeai* no podrá conformarse sólo con conocimientos específicos de las carreras de libertades, sino que deberá contemplar también el resto de aspectos tácticos del juego.

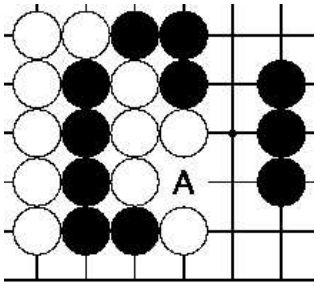

Figura 3.1: El *tesuji* en A permite a negro ganar el *semeai*[3].

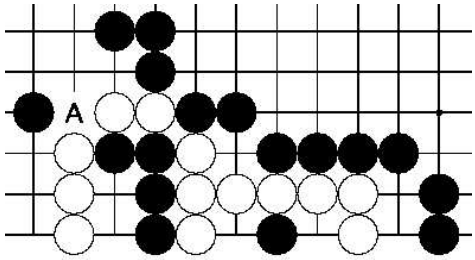

Figura 3.2: Cortar en A permite a negro conectar con el exterior y ganar el *semeai*.

En la figura 3.1 el *throw-in* en A es un *tesuji* que reduce en una las libertades de blanco permitiendo a negro ganar la carrera. En la figura 3.2 el corte en A y la posterior captura de dos piedras transforma lo que hubiera sido un *semeai* completamente perdido para negro en una situación en la que ni necesita jugar para ganar. Estos ejemplos muestran la importancia de la mayoría de los aspectos tácticos del Go para gestionar de manera correcta las carreras de libertades.

Por otro lado son muchas y muy variadas las situaciones que caben dentro de la definición de *semeai*. Con lo que sólo el conocimiento específico para las carreras de libertades, ya es de por sí considerable. En la figura 3.3 presentamos un ejemplo de *semeai* con ojos.

Otra de las dificultades que aparecen es la dimensión del espacio de posibles movimientos. Podemos encontrar ejemplos de *semeai* con más de veinte posibles movimientos a considerar en el nodo raíz y con una secuencia de resolución

[3]Problema 64 página 23 [11]

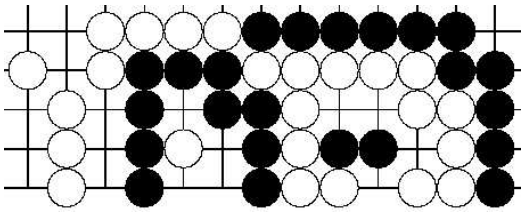

Figura 3.3: *Semeai* con ojos.

de profundidad diez. Estamos hablando de un árbol de búsqueda de aproximadamente $6{,}7 \cdot 10^{12}$ nodos. Semejante tamaño requiere una solución mucho más sofisticada que la simple fuerza bruta de un algoritmo Alpha-Beta. En la figura 3.4 vemos un ejemplo de *semeai* con un árbol de búsqueda inmenso. En el nodo raíz hay 21 movimientos posibles para blanco.

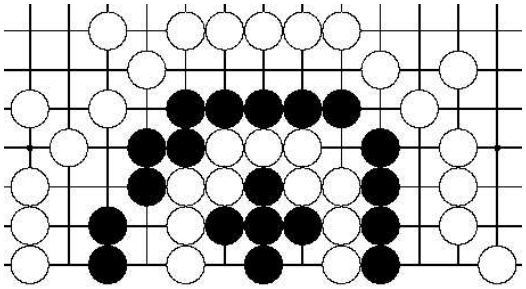

Figura 3.4: En el nodo raíz hay 21 movimientos posibles para blanco.

Finalmente, cabe destacar algunos ejemplos significativos de lo que podríamos calificar como «situaciones extrañas».

En la figura 3.5 se muestra un sorprendente ejemplo de carrera de libertades extrema. En esta composición de Intetsu Akaboshi , blanco inicia el problema con una captura de 16 piedras negras; durante toda la secuencia blanco llega a capturar 88 piedras negras y aún así acaba muriendo con un sólo ojo. La jugada 1 para blanco inicia la increíble secuencia.

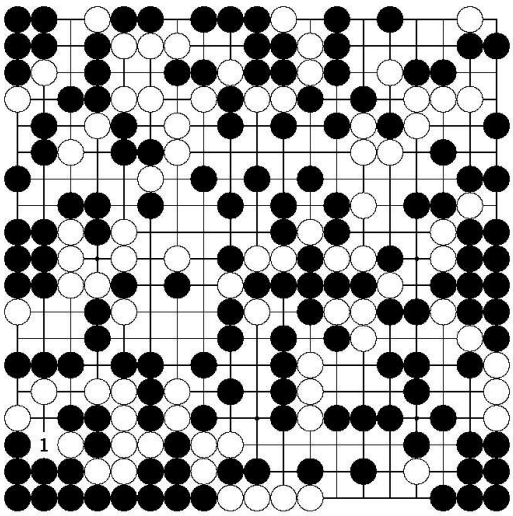

Figura 3.5: Composición del siglo XVIII de Intetsu Akaboshi.

Otra curiosa composición es la de Genan Inseki . En la figura 3.6 se muestra la situación inicial, negro juega e intenta capturar el grupo blanco señalado con triángulos. La lucha que se había iniciado en la esquina superior derecha se estabiliza como *seki* local. Dada la gran cantidad de cortes y de grupos sin un espacio de ojos bien definido, la carrera iniciada se expande y serpentea por todo el tablero. Finalmente tras 150 movimientos (en la figura 3.7 se indican los 50 primeros) la situación se estabiliza globalmente en todo el tablero. La gran sorpresa es que todo el tablero se ha convertido en un gigantesco *seki*.

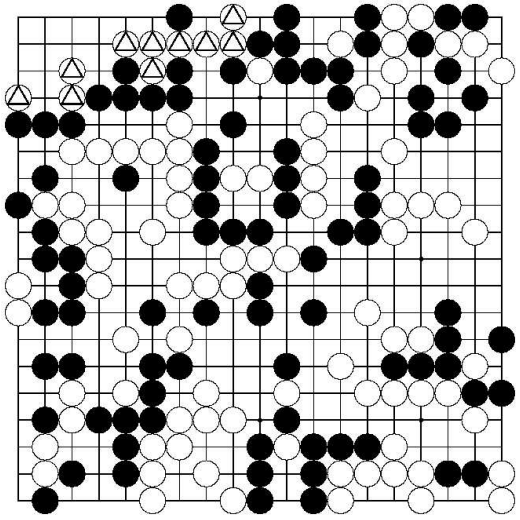

Figura 3.6: Composición de Genan Inseki

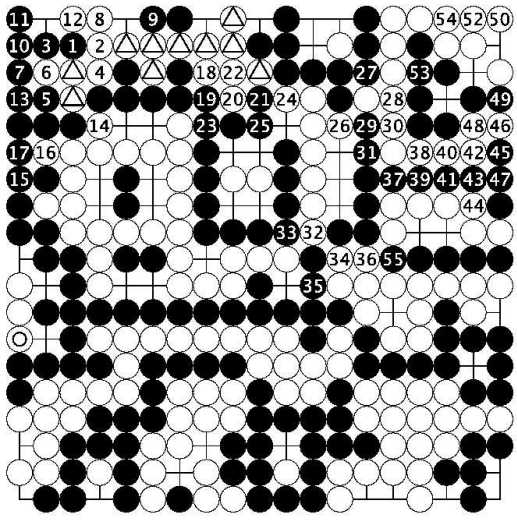

Figura 3.7: Situación final tras 150 movimientos

Otros ejemplos sorprendentes como la situación de *hane-seki* mostrada en la figura A.18 se enmarcan dentro de la amplia definición de *semeai*.

Por todo lo visto hasta ahora: la imposibilidad de desligarlo del resto de aspectos tácticos del Go, el tamaño descomunal de los árboles de búsqueda asociados y la amplitud de situaciones, extrañas o no, que engloba la definición de *semeai*, podemos calificar la dificultad del problema como «extremadamente alta».

Parte II

Solución y Resultados

Capítulo 4

Solución

> La frase mas excitante que se puede oír en ciencia,
> la que anuncia nuevos descubrimientos,
> no es "Eureka!" sino "Es extraño ...".
> I. Asimov

En el presente capítulo desarrollaremos en detalle la solución que proponemos al problema expuesto en el capítulo 3. En la sección 4.1 explicamos las herramientas de las que disponíamos para iniciar nuestro trabajo. En el apartado 4.2 analizaremos los pormenores de la clase Eye, paso previo fundamental para poder entender la estructura detallada del módulo *Semeai-01ES* , presentada en la sección 4.3. Por último, incluimos las declaraciones de las clases que conforman el módulo en la sección 4.4.

4.1 Punto de partida

Cuando nos dispusimos a diseñar el módulo *Semeai-01ES* , por suerte, no tuvimos que empezar de cero. Por un lado, en el plano teórico, el trabajo realizado por Martin Müller sobre *semeai* fue de gran utilidad en un principio y una fuente continua de inspiración durante todo el proceso. Por otro lado, la biblioteca de AI desarrollada por Tristan Cazenave supuso, además de un modelo para la escritura de nuevo código, una base inagotable de funciones que se revelaron altamente útiles en el desarrollo del proyecto.

4.1.1 El trabajo de Müller

En *Race to Capture: Analyzing Semeai in Go* [30], Müller realiza un interesante estudio de las carreras de libertades. En primer lugar define con el máximo rigor posible los distintos elementos que conforman un *semeai*, conceptos que tomaremos prestados y reutilizaremos en este trabajo como «libertad *plain*», «bloque esencial»,...

Define un ojo *plain* como un «área completamente rodeada por un único bloque esencial donde todas las intersecciones libres son adyacentes a dicho bloque». Observa a su vez la diferencia entre ojos pequeños (de tamaño inferior a 4) y grandes tanto en términos de libertades como en la ventaja que proporcionan a la hora de luchar un *semeai*. Modela este comportamiento con un estatus que mostramos en la tabla 4.1 y al que nos referiremos en adelante como: `MullerStatus`. Desarrollaremos más a fondo este concepto en la sección 4.2.2.

TAMAÑO DE OJO	0	1	2	3	4	5	6	7
MULLERSTATUS	0	1	1	1	4	5	6	7
LIBERTADES	0	1	2	3	5	8	12	17

Tabla 4.1: `MullerStatus` y libertades para los distintos tamaños de ojo[1].

A continuación, describe los pasos a seguir para identificar la presencia de un *semeai*. Este proceso corresponde al programa general que es quien tiene la información completa sobre la partición del tablero, los grupos seguros,... y no al módulo de resolución de *semeai* en sí.

Sin embargo, la clasificación de los distintos tipos de *semeai* en ocho grupos diferenciados ha marcado la guía por donde han transcurrido nuestros primeros pasos. En la sección 4.3 presentaremos nuestra visión de la clasificación, claramente inspirada en el trabajo de Müller, pero a la vez atrevida a la hora de incluir tantas modificaciones como creíamos oportuno.

Müller detalla su análisis estático de las clases 0 y 1. Nos dice que ha implementado también un análisis estático para la clase 2 y un método basado en la búsqueda arborescente para las clases 3 y superior; sin embargo, los detalles se omiten por falta de espacio.

Por último, propone un método aplicable a un alto número de situaciones para calcular el valor combinatorio exacto de cada *semeai*, muestra algunas situaciones extrañas que sobrepasan la teoría presentada y acaba comentando algunos de los tests que resuelve de manera satisfactoria.

[1] [30] página 3.

4.1.2 TAIL

TAIL, «Template Library for Artificial Intelligence», es una biblioteca desarrollada principalmente y bajo la dirección de Tristan Cazenave por el grupo de Teoría de Juegos de la Universidad de París VIII.

TAIL está escrita en *C++* y utiliza a fondo todas las potencialidades de herencia, plantillas (*templates*) y polimorfismo que ofrece el lenguaje.

El corazón de TAIL se encuentra en `/tail` y contiene los algoritmos genéricos de búsqueda agrupados fundamentalmente en dos *templates*:

1. `GenericAlphaBeta.h` implementa el algoritmo de búsqueda Alpha-Beta con la posibilidad de añadir las siguientes mejoras: búsqueda de variación principal (PVS), tabla de transposiciones, *null move*, *killer move* y ordenación de movimientos según *history heuristic*.

2. `ABGT.h` implementa las ideas sobre «Generalized Threats Search» (GTS) presentadas en [13]. Por lo tanto, disponemos de un algoritmo Alpha-Beta con GTS de base y la posibilidad de añadir, además de las mejoras anteriormente mencionadas, *iterarive deepening*, *iterative widening* y búsqueda de quiescencia.

Estas plantillas requieren tres parámetros: una clase `Move`, una clase `Board` y una clase `Game`. De este modo, el código del algoritmo de búsqueda se reutiliza para cada nuevo juego que se desee implementar y sólo cabe preocuparse de definir de manera adecuada las tres clases que actúan de parámetro.

TAIL está orientada principalmente a los aspectos tácticos del Go. Aún así hay espacio para otros juegos como las damas francesas (*draughts*), las Líneas de Acción (*Lines of Action, LOA*) o el fútbol de los filósofos de Conway (*Phutball*).

Dada su clara vocación goística, la librería TAIL implementa varias clases que han sido de gran utilidad y exhaustiva utilización en el desarrollo del módulo *Semeai-01ES*. ºNo entraremos a describirlas en detalle, pero sí al menos, citarlas y dar algunas pinceladas sobre su funcionalidad:

- `RectangularBoard`, `GoBoard` e `IncrementalGoBoard`.

  Con estas tres clases se describe por completo todo lo necesario para gestionar eficientemente el tablero de Go. `RectangularBoard` es la clase más básica y define el tamaño y el vector de elementos que componen el tablero. Se puede utilizar como clase base para cualquier juego que utilice un tablero rectangular (en particular cuadrado). `GoBoard` es una clase derivada de la primera y se encarga de la gestión de los *flags* para el control del *ko*, de controlar el turno de juego y del *hashing* de posiciones. Por último `IncrementalGoBoard` se deriva de la anterior y se ocupa del control incremental de los grupos y de la determinación de la legalidad de las jugadas.

- `Rzone` y `StructureGoBoard`.

  `Rzone` almacena de manera eficiente (las operaciones de inclusión, exclusión y consulta se realizan a nivel de bit) un conjunto de intersecciones que tengan relevancia por algún motivo. El conjunto de intersecciones que recorre un *shisho* (cf. figura C.2) o el conjunto de intersecciones que forman un gran ojo (se utilizará en la implementación de la clase `Eye` cf. sección 4.2.2) son algunos ejemplos.

  `StructureGoBoard` nos ofrece la posibilidad de obtener información relativa a una intersección según la estructura del tablero. Si se trata de una intersección de esquina, lateral o centro; si está en primera, segunda o en tercera línea; qué intersecciones son vecinas y cuáles son diagonales a una intersección dada,...

- `String` y `StringArray`

  La clase `String` gestiona todo lo relativo a un grupo de piedras conectadas sobre el tablero. La piedras que lo componen, sus libertades,... y ofrece algunos métodos para determinar los grupos adyacentes, si tiene libertades comunes con otro grupo dado y otras cuestiones que se plantean útiles durante el desarrollo de la partida.

  La clase `StringArray` define una pila de objetos de la clase `String`.

- `GoMove`

  La clase `GoMove` define el movimiento y el color de una jugada.

- `GoHelpers`

  La clase `GoHelpers` contiene como único miembro una referencia a un objeto de la clase `IncrementalGoBoard` y define un gran número de métodos que facilitan la programación en múltitud de situaciones como: tests sobre los grupos, las libertades actuales, las libertades en caso de hacer una jugada determinada, las segundas y terceras libertades, las libertades protegidas, la existencia de *snapback* o *uttegaeshi*,...

Basándose en las clases anteriormente comentadas, se implementan distintos módulos tácticos del Go dando lugar a distintas clases `Game` como `CaptureGame`, `ConnectGame`, `LifeGame` y por supuesto la que aquí nos ocupa: `SemeaiGame`.

Por último, mencionar que todos los juegos implementados son exhaustivamente testeados, utilizando las potencialidades del protocolo GTP (cf. sección 5.1), mediante múltiples tests de regresión.

4.2 La Clase Eye

Hemos reservado un apartado propio para la clase *Eye*, en primer lugar, porque se ocupa de un aspecto totalmente diferente del problema principal de *semeai* y por otro lado, porque la solución hallada tiene suficiente entidad como para merecer una sección propia. Los resultados obtenidos para la manipulación eficiente de los ojos han sido presentados a la «10th Advances in Computer Games Conference», celebrada en noviembre de 2003 en Graz, Austria.

En la sección 4.2.1 presentamos un resumen del artículo publicado [35] donde se detallan los fundamentos teóricos de nuestra aproximación al problema, a continuación en la sección 4.2.2 describimos la manera en que hemos implementado dicha teoría. Hemos optado por no traducir los términos clave puesto que es así como figuran en el artículo en que se presentaron.

4.2.1 *Neighbour Classification* y *Life Property*

I Introducción

Es sabido, tanto por jugadores como por programadores de Go, que cuando un grupo tiene dos ojos está vivo. Aunque es una condición suficiente no es necesaria, en ocasiones un gran ojo es suficiente para vivir, sea porque es posible hacer dos ojos en cualquier momento sea porque está vivo en *seki*.

Este artículo trata sobre la clasificación de ojos grandes y sobre cuándo un gran ojo es suficiente para vivir. Proponemos un algoritmo que responde a esta cuestión estáticamente. Es fácil de programar y muy rápido. Presentamos la *neighbour classification*, un concepto completamente nuevo que permite agrupar las distintas formas de ojo según clases con propiedades comunes. Introducimos también el concepto de *life property* que permite decidir cuando una forma de ojo está viva independientemente del número de piedras enemigas jugadas en su interior. Esta propiedad se basa únicamente en la forma y, allí donde es aplicable, es realmente útil. Es una herramienta completamente segura al no haber heurística alguna implicada.

II Trabajos Previos

La caracterización de ojos y la vida y muerte son dos aspectos del Go que han sido abordados con gran éxito por distintos investigadores.

Landman [26] aplica la teoría combinatoria de juegos para determinar un valor numérico para una forma de ojo determinada.

Fotland [19] describe cómo su programa, «The Many Faces of Go», analiza los ojos. Fotland representa los ojos según su árbol de juego con cuatro valores

distintos; un límite superior e inferior al número de ojos finales y dos valores intermedios para incluir efectos de incertidumbre y *ko*. Su trabajo abarca una gran variedad de ojos y combina análisis estático con una pequeña búsqueda.

Chen & Chen [14] proponen un método general para evaluar heurísticamente la vida de distintas clases de grupos.

Müller [30] extiende el algoritmo de Benson [5] describiendo la seguridad de los grupos en el juego alternado.

Los ojos grandes son de vital importancia en una gran variedad de problemas de *semeai*. Aunque no son una pieza clave para la mayoría de los problemas comunes de *tsumego* cuando aparecen es fundamental gestionarlos de la manera adecuada. La mayoría de las técnicas existentes los tratan de manera insatisfactoria, bien porque lo hacen de manera heurística con lo que pueden aparecer situaciones inesperadas que provoquen una respuesta equivocada, bien porque simplemente dejan que el algoritmo de búsqueda continúe hasta que se convierta en un ojo pequeño con los consiguientes problemas de ineficiencia computacional. En este artículo proponemos una teoría y un algoritmo para la gestión estática de este problema. Es rápido, fácil de programar, libre de consideraciones heurísticas y por tanto completamente seguro. Puede reemplazar, en un gran número de situaciones, un árbol de búsqueda considerablemente profundo por una rápida evaluación estática y puede ser de gran ayuda para mejorar las técnicas existentes y reducir el grado de inexactitud.

III Definiciones

Ojo En el presente capítulo un ojo[2] es un área completamente rodeada por un solo grupo. En el área rodeada aceptaremos tanto piedras del enemigo como propias así como intersecciones vacías no adyacentes al bloque exterior. Esto es una generalización del concepto de *plain eye* definido por Müller [30].

Clasificaremos los ojos según su posición en el tablero:

Corner Eye — El ojo contiene un punto de esquina y sus dos vecinos (cf. figura 4.1).

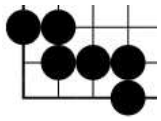

Figura 4.1: Un *corner eye* de clase [1122][3]no *plain*.

Side Eye — El ojo no es un *corner eye* y contiene al menos tres intersecciones laterales (nota: una intersección de esquina es un caso particular de intersección lateral cf. figura 4.2).

[2]En la literatura existente el término ojo se utiliza para referirse a un ojo de un solo espacio, mientras que para ojos mayores se utiliza *X-enclosed region* [5] o *big eye* [19]. En este artículo básicamente nos ocupamos de grandes ojos por tanto, al no haber confusión posible, mantendremos el término *ojo* según la definición dada.

[3]La notación [1122] se explica en la sección IV

Centre Eye — Todo ojo que no sea *corner* ni *side* (cf. figura 4.3, nota: la mayoría de formas de ojo grandes sólo pueden ser *centre eyes* [31]).

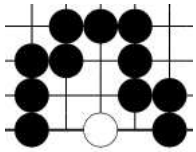

Figura 4.2: Un *side eye* de clase [112234] *plain*.

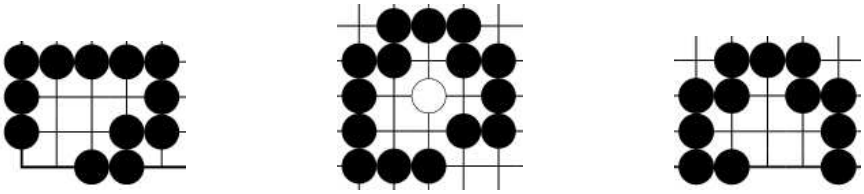

Figura 4.3: Un *centre eye* [1122233] eye, *plain* (izda.); un *centre eye* [112224], *plain* (centro), y un *centre eye* [112224], no *plain* (dcha.).

Forma de ojo Es el conjunto de intersecciones del ojo. Las intersecciones pueden estar vacías u ocupadas por piedras propias o enemigas. Utilizaremos el término *forma Nakade* para referirnos al conjunto de intersecciones que, en caso de ser una forma de ojo, tienen uno o cero *puntos vitales*.

Estatus del ojo Definiremos cuatro posibles estatus para un ojo central: *Nakade, Unsettled, Alive y AliveInAtari*

Nakade — el ojo acabará siendo un único ojo que no será suficiente para vivir. Un ojo nakade puede ser el resultado de: un ojo con un conjunto vacío de *puntos vitales* (cf. figura 4.5), o bien, un ojo con todo su conjunto de *puntos vitales* ocupados por piedras enemigas (cf. figura 4.4).

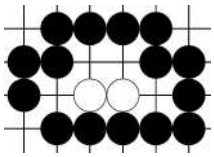

Figura 4.4: Estatus nakade para un ojo de clase [112233]-α. Los dos *puntos vitales* están ocupados por el adversario.

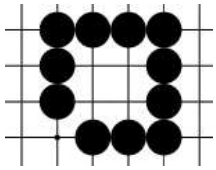

Figura 4.5: Estatus nakade para un ojo de clase [2222]. Este ojo tiene un conjunto vacío de *puntos vitales*.

Unsettled — el ojo acabará teniendo un estatus *nakade* o *alive* dependiendo del turno de juego. Un ojo *unsettled* aparece como resultado de tener una y sólo una intersección vacía en el conjunto de *puntos vitales* (cf. figura 4.6). Un estatus unsettled es lo que Landman define en [26] como $1\frac{1}{2}\varepsilon$.

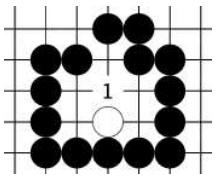

Figura 4.6: Estatus unsettled para un ojo de clase [1222234]. Uno de los dos *puntos vitales* (1) está vacío.

Alive — el grupo poseedor del ojo está vivo independientemente del turno de juego y de las condiciones externas del grupo. Un estatus *alive* puede ser el resultado de: un ojo con dos o más intersecciones vacías en el conjunto de *puntos vitales* (cf. figura 4.7) o que el ojo sea una *n*–forma que no puede ser llenado por el advesario con una $(n-1)$–forma *nakade* (cf. figura 4.8). No haremos distinción alguna entre estar vivo y estar vivo en *seki* como en la figura 4.8, ya que, como dice Landman, «en la mayoría de situaciones estar vivo en *seki* es casi tan bueno como estar vivo con dos ojos»[4].

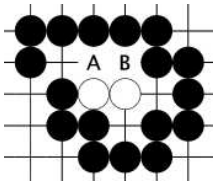

Figura 4.7: Estatus alive para un ojo de clase [1112234]-β. Aunque esta forma puede ser llenada con un *rabbity six*[5], A y B pertenecen al conjunto de *puntos vitales* con lo que hay *miai* de vida.

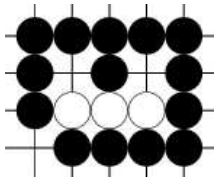

Figura 4.8: Estatus alive para un ojo de clase [11222]. Independientemente del número de turnos seguidos que juegue blanco, negro está incondicionalmente vivo. Blanco no puede llenar la forma de ojo con una *forma nakade* de tamaño cuatro.

AliveInAtari — es un caso particular en que las condiciones exteriores determinan el estatus del ojo. Diremos que un ojo tiene un estatus AliveInAtari si sólo hay una o cero intersecciones vacías adyacentes al grupo pero capturar las piedras del oponente en el interior del ojo garantiza un estatus *alive*. Sólo cuando se han ocupado las libertades exteriores del grupo poseedor del ojo es necesario capturar las piedras del interior (cf. figura 4.9 y 4.10).

[4] Página 242 [26]

[5] Forma de ojo de tamaño seis que se asocia con la cara de un conejo (ver figura 4.13). El término se acuñó por primera vez en [16].

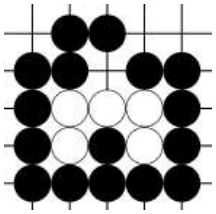

Figura 4.9: Estatus *AliveInAtari* para un ojo de clase [111223]. Capturar garantiza la vida.

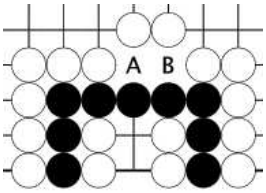

Figura 4.10: Estatus *AliveInAtari* para un ojo de clase [222233]. Cuando A sea ocupado el estatus cambia a *unsettled*, y si se ocupan A y B el estatus es *nakade*. Sin embargo capturar las piedras del interior garantiza un estatus *alive*.

Puntos vitales Conjunto minimal de intersecciones dentro de la forma de ojo que deben ser ocupados por el adversario para garantizar un estatus *nakade* (cf. figura 4.11 y 4.12).

Puntos finales Conjunto minimal de intersecciones dentro de la forma de ojo que no deben ser ocupados por el oponente hasta el final para garantizar un estatus *nakade* en el proceso de matar al grupo (cf. figura 4.11 y 4.12).

Los *puntos finales* no deben confundirse con el *number of ends* de Fotland descritos en [19], mientras que el concepto de Fotland hace referencia a la forma, nuestro concepto hace referencia al orden en que las intersecciones del ojo deben ser ocupadas por el adversario. En la figura 4.11 hay un *punto final* pero tres *ends* de Fotland. Cabe destacar que en la mayoría de los casos lo que es un *punto final* desde el punto de vista de este artículo es también un *end* de Fotland.

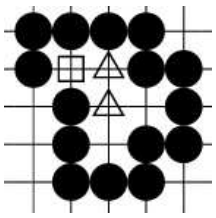

Figura 4.11: *Vital* (△) y *end* (□) *points* para un ojo [11123].

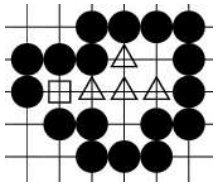

Figura 4.12: *Vital* (△) y *end* (□) *points* para un ojo [1122224].

Life Property Diremos que una forma de ojo tiene la *life property* si los únicos posibles estados para el ojo son *alive* o *aliveInAtari*. Por lo tanto, cuando un ojo tiene la *life property*, sólo cabe comprobar si hay que capturar las piedras del interior del ojo a causa de un estatus *aliveInAtari*.

Por ejemplo una 3–forma en línea puede tener un estatus *nakade*, *unsettled* o *aliveInAtari* dependiendo de las piedras del adversario jugadas en su interior. Por ello esta 3–forma no tiene la *life property* ya que los estatus *nakade* y *unsettled* son posibles. En cambio la forma [11222] que aparece en la figura 4.8 **sólo** puede tener como estatus *alive* o *aliveInAtari* (cuando cuatro de las cinco intersecciones han sido jugadas por el adversario), esta forma tiene la *life property*. Por tanto, mientras que todas las formas de ojo que tienen la *life property* están vivas, no todas las formas que puedan tener un estatus *alive* tienen la *life property*.

La *life property* debe entenderse como una propiedad ligeramente por debajo de la *unconditional life* de Benson [5], ya que si tenemos un *aliveInAtari* estatus puede ser necesario responder dentro del ojo, pero con la ventaja de que detectar la *life property* es una simple cuestión de contar vecinos como se muestra a continuación en la sección IV.

IV Neighbour Classification

Sea $\mathcal{E}_i$ el conjunto de posibles formas de ojo de tamaño i. Nótese que para $i = 1.,6$ existe un isomorfismo entre $\mathcal{E}_i$ y $\mathcal{P}_i$ siendo $\mathcal{P}_i$ el conjunto de i–poliominos libres [31]. Un n–poliomino (o "n–omino") se define como una colección de n cuadrados de igual tamaño colocados con lados coincidentes. Los poliominos libres pueden rotarse de manera que piezas simétricas se consideran idénticas. Para tamaño siete hay que descartar el poliomino agujereado para mantener el isomorfismo.

Sea $e \in \mathcal{E}_i$ definiremos la *Neighbour Classification* de e, $NC(e)$, como un número de i dígitos ordenados de menor a mayor; a cada intersección en la forma de ojo se le asocia un dígito que indica el número de vecinos (intersecciones adyacentes) de esa intersección que pertenecen al espacio de ojo (cf. figura 4.13).

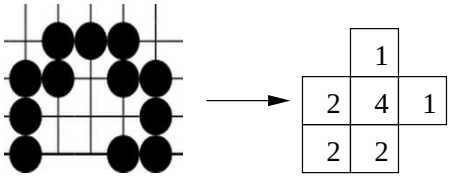

Figura 4.13: Para el *rabbity six* $NC(e) = 112224$

Sea $\sim$ la siguiente clase de equivalencia: sea $e_1, e_2 \in \mathcal{E}_i$ entonces

$$e_1 \sim e_2 \iff NC(e_1) = NC(e_2)$$

Por tanto $\sim$ genera una partición de $\mathcal{E}_i$ definida por las clases de equivalencia en $\mathcal{E}_i / \sim$ (cf. apéndice D).

Ejemplo Dado $\mathcal{E}_5$ podemos encontrar cuatro distintas *Neighbour Classifications* para sus elementos (nótese que $\|\mathcal{E}_5\| = \|\mathcal{P}_5\| = 12$) [31].

$$NC(e) \in \{11222, 11123, 11114, 12223\}, \quad \forall e \in \mathcal{E}_5$$

Theorem 1 (de la neighbour classification) *Sea e un* center eye, $e \in \mathcal{E}_i$ *y* $[e] \in \mathcal{E}_i / \sim$ *la clase de equivalencia de e, para* $i = 1..,7$ *si e tiene la* life property *entonces* $\forall f \in [e]$, f *tiene la* life property. *Inversamente, si e no tiene la* life property *entonces* $\forall f \in [e]$, f *no tiene la* life property.

Demostración: Para $i \in \{1,2,3,4\}$ no hay ninguna forma de ojo que tenga la *life property* por tanto el teorema es correcto. Las formas de tamaño 1 y 2 son siempre *nakade*, las dos 3–formas existentes tienen un punto vital con lo que su estatus puede ser *nakade* o *unsettled* (dependiendo de si el adversario ha jugado o no el punto vital). Hay cinco 4–formas con cero, uno o dos puntos vitales. Todas pueden tener un estatus *nakade* si el adversario juega todos los puntos vitales. La parte interesante aparece con las formas de tamaño mayor.

Bajo las condiciones del teorema, tener la *life property* es simplemente una cuestión de forma. Si y sólo si una *i*–forma no puede ser llenada por el oponente con una $(i-1)$–forma que tenga uno o cero puntos vitales (*forma nakade*), entonces esa *i*–forma tiene la *life property*.

Center eyes de tamaño menor que siete no pueden tener *ko* en su interior. Sólo hay dos 7–formas que pueden tener un estatus de ko en el centro (cf. figura 4.14. Estas formas no tienen la *life property* ya que pueden ser llenadas por un *rabbity six* por lo que el *ko* no interfiere en el resultado del teorema.

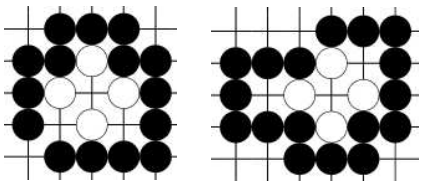

Figura 4.14: Formas de clase [1222234] y [1122224] pueden tener un estatus *ko* en el centro.

5–formas Hay dos *formas nakade* de tamaño cuatro (el cuadrado y la pirámide) por tanto todas las 5–formas que no contengan un cuadrado o una pirámide tendrán la *life property*. Todas las formas pertenecientes a la clase [11222] tienen la *life property* mientras que el resto no la tiene. Las únicas *formas nakade* de tamaño cinco son el coche [12223] y la estrella [11114] (cf. figura C.1).

6–formas Como antes, todas las 6–formas que no tengan un coche o una estrella dentro tienen la *life property*. Son las 26 formas pertenecientes a las clases $\{[112222],[111223],[111133]\}$. La única *forma nakade* de tamaño seis es el *rabitty six*.

$\mathcal{E}_i$	$\mathcal{E}_i/\sim$	$\lvert\cdot\rvert$	*Life Property*
$\mathcal{E}_1$		1	
	$[0]$	1	No
$\mathcal{E}_2$		1	
	$[11]$	1	No
$\mathcal{E}_3$		2	
	$[121]$	2	No
$\mathcal{E}_4$		5	
	$[1122]$	3	No
	$[1113]$	1	No
	$[2222]$	1	No
$\mathcal{E}_5$		12	
	$[11222]$	7	Sí
	$[11123]$	3	No
	$[11114]$	1	No
	$[12223]$	1	No

$\mathcal{E}_i$	$\mathcal{E}_i/\sim$	$\lvert\cdot\rvert$	*Life Property*
$\mathcal{E}_6$		35	
	$[112222]$	13	Sí
	$[111223]$	12	Sí
	$[111133]$	1	Sí
	$[112233]$	4	No
	$[122223]$	2	No
	$[112224]$	1	No
	$[111124]$	1	No
	$[222233]$	1	No
$\mathcal{E}_7$		107	
	$[1122222]$	30	Sí
	$[1112223]$	40	Sí
	$[1122233]$	11	Sí
	$[1111233]$	8	Sí
	$[1222223]$	5	Sí
	$[1111224]$	4	Sí
	$[1112333]$	2	Sí
	$[1222333]$	2	Sí
	$[1112234]$	2	No
	$[1222234]$	1	No
	$[1122224]$	1	No
	$[2222224]$	1	No

Tabla 4.2: *Neighbour Classifi cation* para $\mathcal{E}_i$, $i = 1.,7$

7–formas Ahora sólo cabe preocuparse de las formas conteniendo un *rabbity six*, hay cinco formas distribuidas en cuatro clases de equivalencia. El resto de 7–formas tienen la *life property*. No hay ninguna *forma nakade* de tamaño siete. Esto concluye la prueba del teorema 1. □

La clasificación exhaustiva de todas las formas de ojo de tamaño menor que ocho se muestra en la tabla 4.2.

El aportación principal del teorema reside en el hecho de que la *life property* sólo depende de la forma del ojo. Por lo tanto, dada una forma de ojo sólo hay

que encontrar su *neighbour classification*. Si la clase tiene la *life property*, independientemente del número de piedras del oponente, sabemos que el grupo que posee el ojo está vivo, únicamente hay que comprobar si es necesario capturar las piedras del adversario en el interior debido a un estatus *aliveInAtari*. Si la clase no posee la *life property* es necesario un estudio adicional para decidir el estatus (cf. sección V).

No es posible extender el teorema para *centre eyes* de tamaño mayor que siete porque aparecen situaciones más complejas como *kos* o la posibilidad de que el oponente haga un ojo en el interior del ojo en cuestión. Sin embargo, veremos que no es un problema ya que la *life property* es una condición demasiado restrictiva para ese tipo de ojos.

V Identificación de *vital points* y *end points*

Otra propiedad interesante de la *neighbour classification* es que permite encontrar fácilmente los *vital* y *end points* para una forma de ojo central dada simplemente observando su código. Vamos a mostrar a modo de ejemplo la identificación para todas las formas de tamaño seis. La identificación para las formas de tamaños de uno a cinco es sencilla de deducir y para tamaño siete se requiere un proceso análogo al que nos disponemos a mostrar.

De las 35 posibles formas de tamaño seis sólo necesitamos analizar 10 formas repartidas en cinco clases de equivalencia. Las otras 25 poseen la *life property* y se resuelven de manera inmediata sin necesidad de mayor estudio.

Veamos como proceder con las cinco clases que no poseen la *life property*:

- **[112224]** El *rabbity six* es la única forma *nakade* de tamaño seis. El punto vital es el 4-punto[6]. El 2-punto no adyacente al punto vital puede considerarse como punto final (cf. figura 4.15). Aunque no es necesario jugarlo al final de la secuencia, sólo si está jugado es necesario comprobar que no hay una forma *nakade* del adversario dentro del ojo.

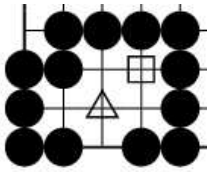

Figura 4.15: *Vital y end points* para la clase [112224].

[6]Utilizaremos esta notación para referirnos al punto que tiene 4 vecinos o que aporta el dígito 4 al código de la clase.

- **[111124]** Los puntos vitales son los {2,4}-puntos y el punto final es el 1-punto adyacente al 2-punto (cf. figura 4.16).

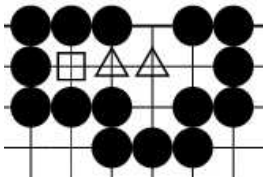

Figura 4.16: *Vital y end points* para la clase [111124].

- **[222233]** Puntos vitales son los dos 3-puntos (cf. figura 4.17). No hay forma eficiente de definir los puntos finales así que habrá que contemplar siempre la posibilidad de una forma *nakade* (zigzag de tamaño 4) en el interior.

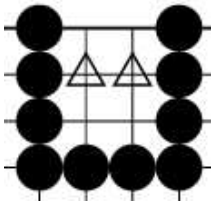

Figura 4.17: *Vital y end points* para la clase [222233].

- **[112233]** Necesitamos crear dos subclases para realizar la identificación. Definiremos la subclase [112233]-α como el subconjunto de dos elementos en que los dos 3-puntos son vecinos, y la clase [112233]-β como el subconjunto de dos elementos en que los dos 3-puntos no son vecinos. De este modo para los α-elementos hay dos puntos vitales correspondientes a los dos 3-puntos y dos puntos finales correspondientes a los dos 1-puntos. Para los β-elementos los puntos finales son los mismos pero las otras cuatro intersecciones son puntos vitales (cf. figura 4.18 y 4.19).

- **[122223]** El punto final es el único 1-punto. Como puntos vitales hay que considerar el 3-punto y sus tres vecinos (cf. figura 4.20).

Hasta el momento no hemos encontrado una manera única de identificar los puntos vitales y finales para cualquier forma de ojo dada, por lo que es necesario una implementación particular para cada clase. Sin embargo la *neighbour classification* permite realizar esa implementación de manera eficiente.

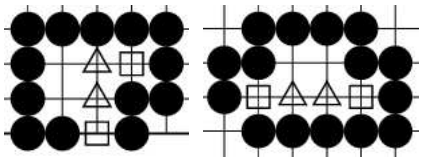

Figura 4.18: *Vital y end points* para la clase [112233]-α.

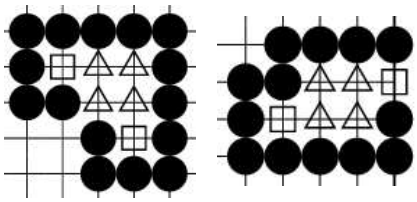

Figura 4.19: *Vital y end points* para la clase [112233]-β.

VI Ojos en el lateral y en la esquina

Para los ojos *corner* y *side* se cumple la siguiente implicación (NoLP = No *Life Property*):

$$NoLP\ Center \Rightarrow NoLP\ Side \Rightarrow NoLP\ Corner$$

Por tanto, una vez hecho el estudio para ojos centrales, sólo las clases con la *Life property* en el centro deben ser comprobados en el lateral y en la esquina.

Para ojos en el lateral, el teorema 1 sigue siendo cierto para tamaños de uno a cuatro. Para tamaños cinco y seis, sólo aparecen situaciones de *ko* en las clases que no tienen la *life property* en el centro por lo que el teorema continua siendo válido. Para tamaño siete hay dos clases ([1222333] y [1112333]) que tienen la *life property* en el centro pero dejan de tenerla en el lateral debido al *ko*. Desgraciadamente, para la clase [1122233] siete de sus once miembros no tienen la *life property* debido al *ko* mientras que los otros cuatro sí que la tienen en el lateral (cf. figuras 4.21 y 4.22). Por lo que el teorema ya no es aplicable para ojos en el lateral de tamaño siete.

Aunque el teorema deja de ser cierto para ojos en el lateral, hay aún mucho conocimiento derivado de la *neighbour classification* que puede ser utilizado para una implementación que resuelva ojos en el lateral. Simplemente necesitaremos considerar más casos y las formas que no tienen la *life property* deberán ser tratadas con mayor cuidado. Por ejemplo, hemos visto que, si el ojo es central, una forma [112233]-α tiene dos puntos vitales y, por tanto, si el oponente no ha jugado en ninguno de los dos puntos vitales tiene un estatus *alive*. Sin embargo esto ya no se cumple en el lateral como muestra la figura 4.23. Para ojos en el lateral

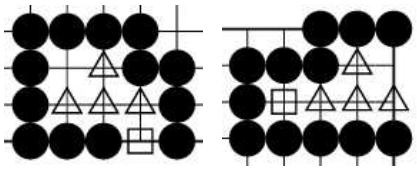

Figura 4.20: *Vital y end points* para la clase [122223].

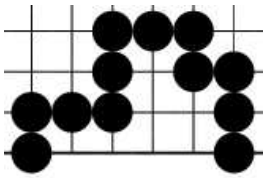

Figura 4.21: Este elemento de la clase [1122233] tiene la *life property* en el lateral.

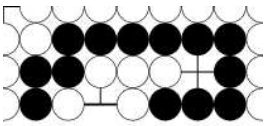

Figura 4.22: Este elemento de la clase [1122233] no tiene la *life property* en el lateral.

aparece lo que podríamos denominar un *Unsettled-Ko* estatus.

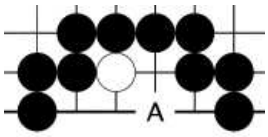

Figura 4.23: Si blanco juega en A aparece un *ko*. Si negro gana el *ko* el estatus será *alive* si lo pierde será *nakade*

En la esquina la situación es aún peor; *bent four*, *ko* y la posibilidad para el oponente de hacer fácilmente un ojo dentro del gran ojo dificulta enormemente la aplicación del teorema.

Cabe destacar cuan fuerte es la condición de tener la *life property*: es fácil encontrar situaciones extrañas de formas de ojo que no tienen la *life property* (ya que puede aparece un *ko*) pero que son prácticamente imposibles de matar en una partida real (cf. figura 4.24).

Sin embargo, el hecho de que el teorema no sea aplicable en la esquina no significa que la *neighbour classification* sea inútil en estos casos. Puede utilizarse para clasificar formas de ojo de una manera eficiente. Por ejemplo, sólo 12 de las 35 formas de ojo de tamaño seis pueden ser *corner eyes*. Seis de esas doce formas son de clase [112222] y [111223], decidir el estatus de estas formas en función de las piedras jugadas en el interior es sencillo. Para las otras seis formas podemos simplemente retornar un estatus *unknown* y dejar que el algoritmo de búsqueda

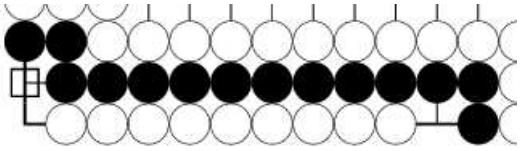

Figura 4.24: Un corner eye de tamaño 12 sin la *life property*. Si blanco juega en □ aparece un estatus de *ko*.

continúe hasta que el ojo se convierta en una forma de tamaño cinco, más sencillo de decidir a través de una implementación caso por caso.

VII Aplicación a problemas de *semeai*

La *neighbour classification* se ha utilizado con éxito para la resolución de problemas de *semeai*. Siguiendo la clasificación de *semeais* de Müller [30] y utilizando la *neighbour classification* hemos conseguido resolver estáticamente las clases de 0 a 2 pero también todas las carreras con ojos centrales o laterales que están por encima de la clase 2 bien sea porque el ojo no es *plain* o bien porque hay más de un bloque no esencial dentro del ojo. Esto constituye una mejora significativa respecto a los resultados alcanzados estáticamente por Müller en [30].

En `www.ai.univ-paris8.fr/~ritx/semeai.zip` puede encontrarse un subconjunto representativo de los problemas de *semeai* resueltos utilizando la *neighbour classification*.

VIII Conclusión

Hemos presentado tres nuevas ideas acerca de los ojos: el concepto de *punto final*, la definición de *life property* y la *neighbour classification*.

La *neighbour classification* junto con la *life property* conforman una herramienta completamente segura para decidir estáticamente el estatus de un ojo bajo ciertas condiciones. El método es sencillo de programar y puede, en muchas situaciones, reemplazar un árbol de búsqueda profundo por una rápida y fiable evaluación estática.

Para formas de ojo que no tienen las condiciones iniciales (ojos en el lateral y en la esquina) hemos mostrado que, a pesar de que no se puede aplicar el teorema, se puede extraer una gran cantidad de información útil gracias a la *neighbour classification*.

Ha sido probado con problemas de *semeai* y se ha revelado como una potente herramienta.

4.2.2 Implementación de la Clase Eye

Como veremos en la sección 4.3, la clase `Eye` juega un papel determinante en el desarrollo del módulo *Semeai-01ES* . La clase `Eye` es el resultado de implementar en código ejecutable las ideas presentadas en la sección 4.2.1.

Consideraciones Previas

La presencia de un ojo en un *semeai* altera por completo el análisis estático del mismo, sobre todo en los casos en que el ojo es de tamaño superior a tres (al ser el número de libertades reales superior al número de libertades físicas). Es por ello por lo que es necesaria la capacidad de detección y análisis de ojos en el estudio de *semeai*. Ahora bien, debe tenerse presente que un problema de *semeai* no es un problema de vida y muerte (donde los ojos juegan un papel si cabe más importante), y por tanto hay que determinar primero qué cantidad de conocimiento nos será necesario.

Considero que no tendría demasiado sentido, por ejemplo, incluir en el módulo *Semeai-01ES* todo el conocimiento que proponen Chen y Chen en [14] ya que el porcentaje de situaciones de en el que se utilizaría sería mínimo. Es innegable, sin embargo, que ante un ojo grande cabe la posibilidad de hacer dos y por tanto unos mínimos conocimientos de vida y muerte deben contemplarse ante el desarrollo de un *semeai*.

Es por todo esto que la teoría presentada en la sección 4.2.1 está lejos de ser un estudio general de ojos. Más bien es un estudio adaptado a las situaciones que se presentan relevantes en los problemas de *semeai* en lo referente a ojos.

Tras estas consideraciones me he regido fundamentalmente por dos ideas a la hora de diseñar la clase `Eye`:

- Es preferible no conocer el valor de una variable que asignarle un valor erróneo.
- Lo que no se conoce a profundidad p en el árbol de búsqueda puede descubrirse a nivel $p+1$.

Funcionalidad

Dado un grupo (clase `String` cf. sección 4.1.2), la clase `Eye` se encarga de determinar si tiene un ojo en el sentido de la definición dada en la sección 4.2.1.III y de decidir el estatus del ojo y el punto vital.

La primera fase, la detección del ojo, se lleva a cabo a manos de la función `Init`, y es la que más tiempo de procesador consume no sólo en cuanto a la clase

Eye se refiere sino a todo el módulo *Semeai-01ES* . Aquí radica el punto crítico a optimizar de todo el módulo.

La segunda fase, la decisión del estatus y el punto vital, se ejecuta de manera prácticamente inmediata, siguiendo las ideas mostradas en la sección 4.2.1. Sin embargo es la que concentra el mayor número de líneas de código de todo el módulo debido a la gran casuística a contemplar especialmente para los ojos que no son de tipo central.

Limitaciones

La limitación más importante que presenta la clase Eye es la necesidad de que el ojo este completamente cerrado por el grupo propietario. En la figura 4.25 el grupo blanco tiene $1\frac{1}{2}$ ojos según la terminología de Landman [26]. Sin embargo el análisis estático de esta situación dará como resultado ningún ojo. Cuando se inicie la búsqueda a profundidad 1 se detectará correctamente que la jugada A da a blanco un ojo de estatus *Alive* o, lo que es lo mismo, dos ojos y por tanto la vida del grupo y la victoria en el *semeai*.

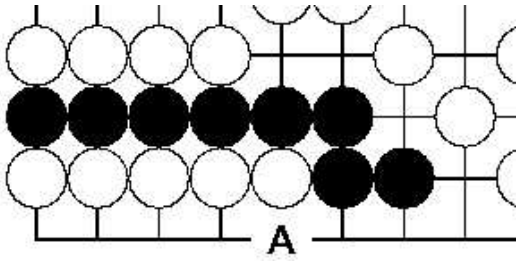

Figura 4.25: Ojo no detectado al no estar la región completamente cerrada.

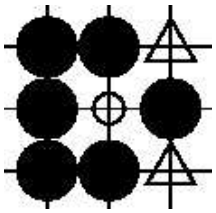

Figura 4.26: Ojo no detectado, compartido por dos grupos.

Otra limitación es la incapacidad para detectar ojos compartidos por más de un grupo. En la figura 4.26 el grupo negro tiene un ojo seguro, blanco necesita jugar a la vez en las dos intersecciones marcadas con triángulos para poder convertirlo en un ojo falso. Sin embargo al estar compartido por dos grupos la clase Eye no lo detectaría. Al avanzar el algoritmo de búsqueda y jugarse una de las dos intersecciones marcadas con triángulo, el ojo pasaría a estar rodeado por un sólo grupo y sería detectado de manera correcta por la clase Eye.

Funciones

La clase Eye tiene un total de 23 métodos aunque la mayoría son subsidiarios de los dos métodos principales: el de inicialización y el de decisión del estatus y el punto vital.

Inicialización El método `init` se encarga de realizar la inicialización completa de las variables de la clase. En primer lugar se encarga de detectar el espacio de ojo, a continuación debe decidir la posición del mismo (*corner*, *side* o *centre*) y, con esta información, decidir el estatus y el punto vital. En la figura 4.27 se muestra el diagrama conceptual de este método.

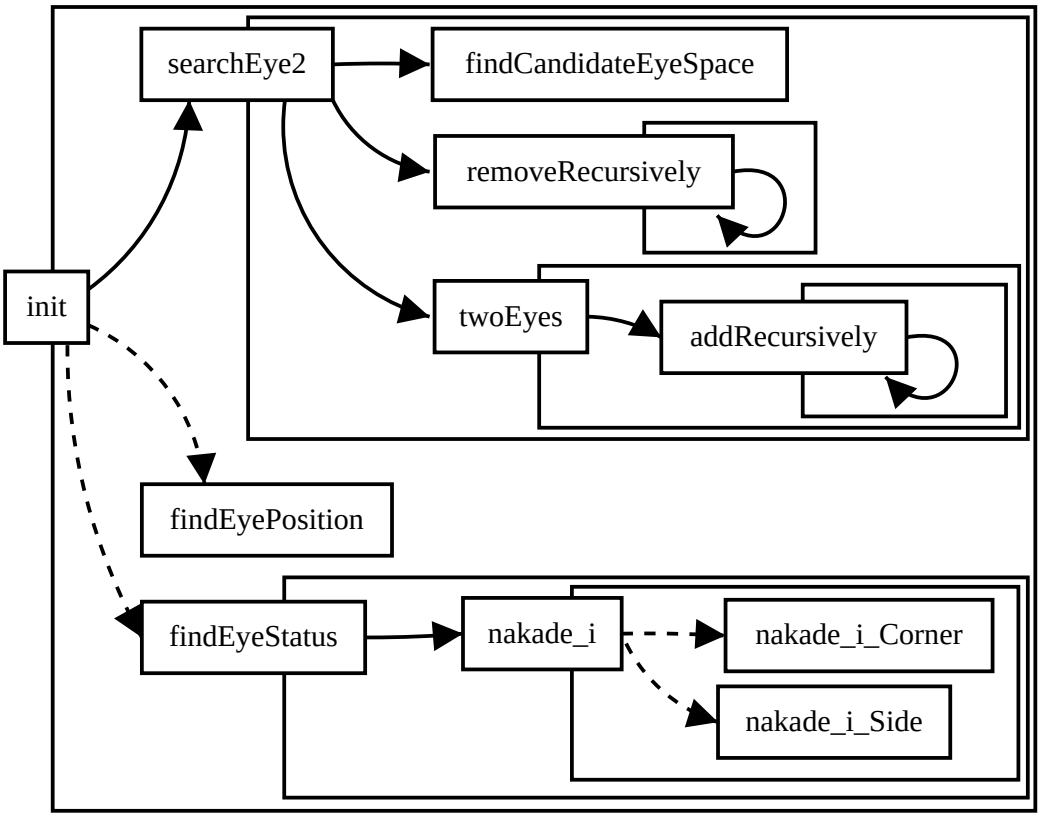

Figura 4.27: Diagrama de flujo de ejecución de la función init de la clase Eye.

Existe también un método `initmin` que sólo se ocupa de determinar el espacio de ojo. Este método permite un importante ahorro computacional a la clase p (cf. sección 4.3.5).

Estatus y punto vital El método `findEyeStatus` deriva, en función del tamaño del ojo y de su posición en el tablero, a la correspondiente función `nakade_i`. Estas funciones determinan en primer lugar la *Neighbour Classification* del ojo y, si no tiene la *life property* determinan, caso por caso, el estatus y el punto vital. Entendemos por punto vital la intersección a jugar en el interior del ojo independientemente del estatus del mismo.

4.3 El módulo *Semeai-01ES*

Semeai-01ES está escrito en *C++* como parte integrada de la librería TAIL. Lo hemos bautizado así porque el nombre describe el proceso de desarrollo que lo originó. Sobre la *clase 0* se diseñó la *clase 1*. Esta, pronto fue sutituida por la *clase e* conformando así la función de evaluación. La *clase s* se encarga de implementar la búsqueda (*search*) utilizando la *clase e* como función de evaluación.

A continuación procedemos a describir la arquitectura general del módulo y los pormenores de cada una de las clases que lo integran.

4.3.1 Arquitectura general

La detección de un *semeai* requiere un conocimiento profundo de la situación global del tablero; qué grupos están incondicionalmente vivos, qué grupos pueden conectarse hacia una zona segura, qué grupos están encerrados,... Por ello implementar la detección es un trabajo que no dista demasiado de la implementación de un programa completo de Go. *Golois* dispone de un pequeño módulo que se encarga, utilizando todo el conocimiento que tiene de una posición dada, de detectar la existencia de una carrera de libertades entre dos grupos. Sin embargo, adolece de un análisis pobre de dichas situaciones.

Con estas condiciones decidimos obviar en nuestro diseño el trabajo de detección y ponernos a trabajar presuponiendo que nuestro punto de partida para realizar el análisis de un *semeai* serían dos intersecciones del tablero, correspondientes a los dos grupos implicados en la carrera. A partir de estos valores iniciales *Semeai-01ES* pone en marcha todo su conocimiento para devolver el resultado adecuado; el estatus de la carrera y el movimiento a jugar.

En la figura 4.28 presentamos un esquema del funcionamiento del módulo. *Semeai-01ES* recibe como argumento de entrada `(i1,i2)` que indican los dos grupos en *semeai*, siendo `i1` del color del turno de juego. A continuación se pone en marcha el algoritmo de búsqueda Alpha-Beta que utiliza como función de evaluación el binomio `Clase p` y `Clase e`. La `clase p` se encarga de decidir si la posición dada pertenece al conjunto de posiciones que la `clase e` sabe resolver de manera correcta. En caso afirmativo el nodo en cuestión es terminal para el árbol de búsqueda y la `clase e` devuelve el status y el movimiento a jugar. De lo contrario se devuelve un resultado *Unknown* y la búsqueda continúa hasta que la posición evolucione a un estado evaluable.

El estatus de la carrera en realidad son dos; el estatus de la carrera en caso de que realicemos una jugada y el estatus en caso de pasar. Esta información combinada con el valor combinatorio de la carrera puede revelarse muy útil para el nivel superior.

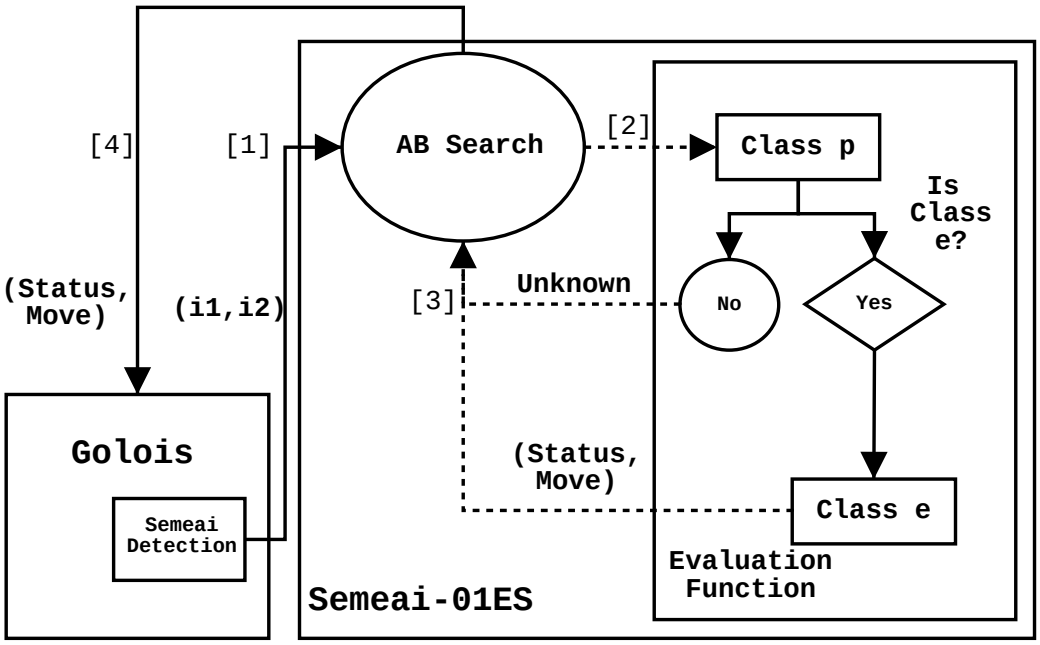

Figura 4.28: Arquitectura general del módulo *Semeai-01ES*

La `clase p` constituye el punto crítico de este esquema. Sea $\mathcal{P}$ el conjunto de posiciones evaluables de manera correcta por la `clase e` y p_i una posición correspondiente a un nodo dado del árbol de búsqueda, si $p_i \in \mathcal{P}$ y no se detecta como tal estaremos desaprovechando la potencialidad de la función de evaluación e incrementando el tamaño del árbol de manera innecesaria. Por el contrario, si $p_i \notin \mathcal{P}$, un error en la `clase p` nos llevará a evaluar precipitadamente y de manera incorrecta la posición p_i.

Para implementar la `clase p` minimizando la posibilidad de error, es fundamental poder definir $\mathcal{P}$, de manera inequívoca, con un conjunto reducido de reglas. Por ello en el diseño de la `clase e` existe un compromiso entre el tamaño del conjunto $\mathcal{P}$ y la sencillez de las reglas que lo definen. Así cada vez que queramos ampliar el conjunto $\mathcal{P}$ mejorando la `clase e` deberemos sopesar primero cómo afectará dicha mejora a la `clase p`.

Estatus y Movimiento

El resultado que devuelve el módulo es el estatus de la carrera y el movimiento a jugar aunque, en realidad, es un poco más complejo. *Semeai-01ES* no devuelve un estatus sino dos; el estatus en caso de jugar `statusIfPlay` y el estatus en caso de pasar `statusIfPass`. En el ejemplo de la figura 4.29 vemos porqué es

necesaria esta aparente redundancia.

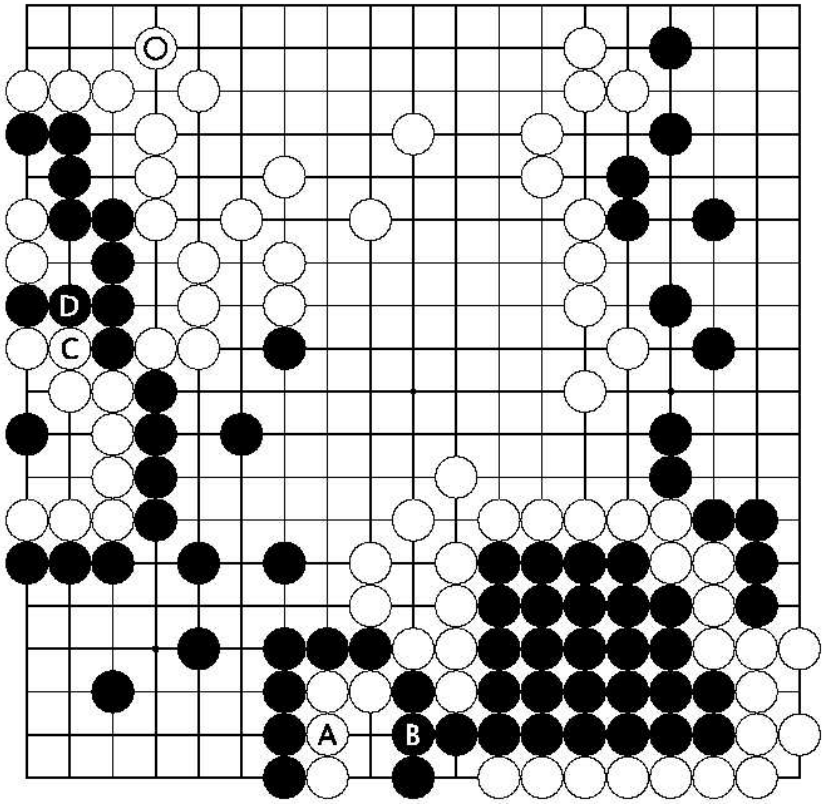

Figura 4.29: ¿Cuál de los dos *semeais* es más grande?

Definimos el valor de un *semeai* como (SWV≡SemeaiWinningValue):

$$SWV(Colour) = ScoreWinning(Colour) - ScoreLoosing(Colour)$$

Así para la carrera (A,B):

$$SWV(Black) = (4 \cdot 2) - (-30 \cdot 2) = 8 - (-60) = 68$$

Para la carrera (C,D):

$$SWV(Black) = ((2 \cdot 9 + 4) + (2 \cdot 2 + 2)) - (-((2 \cdot 10 + 2) + (2 \cdot 1 + 4))) = 54$$

Si la única información que transmitimos al nivel superior es el valor de victoria y el estatus en caso de jugar, en el ejemplo de la figura 4.29 se escogería erróneamente jugar en el *semeai* entre A y B. Definamos ahora el valor de *seki* de un *semeai* (SemeaiSekiValue) como:

$$SSV(Colour) = ScoreSeki(Colour) - ScoreLoosing(Colour)$$

Para la carrera (A,B):

$$SSV(Black) = 0 - (-2 \cdot 30) = 60$$

En realidad para la carrera entre A y B el estatus es *Winner* en caso de jugar, pero *Seki* en caso de pasar y la diferencia entre un resultado y otro es de tan sólo 8 puntos. Para la carrera entre C y D los estatus son *Winner* y *Looser* respectivamente con una diferencia de 54 puntos. Por ello jugar en (C,D) es 46 puntos más grande que en (A,B) y esta diferencia sólo es detectable a alto nivel si mantenemos hasta el final la información sobre los dos estatus.

Los posibles estatus a nivel de función de evaluación para un *semeai* dado son: `{Winner, Looser, Seki, KoSemiai, NotKnown}` Por el momento la gestión de situaciones de *ko* es muy rudimentaria y casi siempre se traducen a estatus desconocido.

El estatus *seki* se ha entendido siempre en la bibliografía sobre Go según la definición dada en la sección A.2.3, por ejemplo la carrera de la figura 4.30 es *seki*. Sin embargo como estatus de *semeai* hemos decidido adoptar la convención de ampliar el significado del término a situaciones en que ambos grupos están vivos por tener dos ojos. Como ya hemos mencionado la línea que separa lo que es un problema de vida y muerte de un problema de *semeai* en ocasiones es difusa. En la carrera de la figura 4.32 negro gana si juega primero. Si juega blanco el estatus es *seki* y puede obtenerlo de dos manera diferentes: jugando en el ojo de negro llegamos a un *seki* convencional, o bien, puede hacer dos ojos, permitiendo a negro vivir a su vez con dos ojos y la situación es igualmente de empate. Por ello entenderemos a partir de ahora el término *seki* aplicado a *semeais* bien como vida compartida o bien como vida independiente (cada uno con sus dos ojos).

En cuanto al movimiento a jugar se trata de la intersección que garantiza el estatus obtenido. Sin embargo a nivel interno manejamos dos variables, `move` y `intersectionToPlay`. La primera se decide junto con el estatus y solo indica si hay que jugar o pasar. Será jugar si $statusIfPlay \neq statusIfPass$ y pasar en caso contrario. La segunda se calcula a posteriori y será una intersección del tablero sólo si $move \neq Pass$, y es el valor que retornamos al nivel superior.

4.3.2 Semeai Clase 0

Un *semeai* de clase 0 según la definición de Müller en [30] es una carrera completamente cerrada, sin grupos no esenciales, sin ojos y donde todas las libertades son *plain*.

[7] STS-RV (cf. sección 5.2.2); fi chero semeais_0.tst, tests 5 y 6.

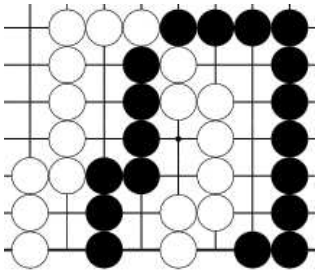

Figura 4.30: *Semeai* de clase 0^7.

Una libertad es *plain* si puede ser ocupada por el adversario sin necesidad de movimientos de aproximación. En la figura 4.30 tenemos un ejemplo de *semeai* de clase 0; un único grupo por bando, sin ojos y todas las libertades son *plain*.

La implementación de la clase 0 es relativamente sencilla. El cuerpo principal de la clase está constituido por tres métodos: `init`, `decideStatus` y `playMove`.

La función `init` inicializa las variables de la clase y se ocupa del cálculo de las libertades compartidas (almacenadas en una clase `Rzone`), fundamentales para decidir el estatus y el movimiento a jugar. También calcula el valor combinatorio de la carrera.

K	SEMEAI STATUS / MOVE
≥ 1	Winner / Pass
0	Winner / Play
≤ -1	Looser / Pass

Tabla 4.3: *Semeai* status $sl = \{0, 1\}$

La función `decideStatus` implementa la casuística mostrada en las tablas 4.3 y 4.4. Sea *k* la diferencia entre las libertades exteriores del grupo que tiene el turno y las libertades exteriores del otro grupo, y sea *sl* el número de libertades compartidas debemos distinguir dos casos. Si las libertades compartidas son 0 ó 1 no hay posibilidad de *seki* y el estatus del *semeai* depende de *k* según la relación mostrada en la tabla 4.3. Si el número de libertades compartidas es mayor o igual que 2 es posible el estatus *seki* y este depende de *k* según la tabla 4.4.

De este modo la función determina el estatus de la carrera y si es necesario jugar para obtenerlo o es posible pasar.

Por último, la función `playMove` determina la intersección a jugar. Si no es posible pasar `playMove` elegirá una libertad exterior y, sólo si no quedan, jugará una libertad compartida. Cuando hay más de una libertad exterior para elegir

K	SEMEAI STATUS / MOVE
$\geq \quad sl$	Winner / Pass
$sl-1$	Winner / Play
$sl-2$	Seki / Pass
$\cdots$	
$-sl+2$	
$-sl+1$	Seki / Play
$\leq \quad -sl$	Looser / Pass

Tabla 4.4: *Semeai* status $sl \geq 2$

escogemos siempre la que maximiza[8] las libertades del grupo en *semeai* [3].

4.3.3 Semeai Clase 1

La clase 1 de Müller admite para ambos grupos la existencia de ojos *plain*[9] que pueden contener un grupo no esencial del adversario en forma *nakade*. La clase 2 es como la clase 1 incluyendo formas *unsettled*.

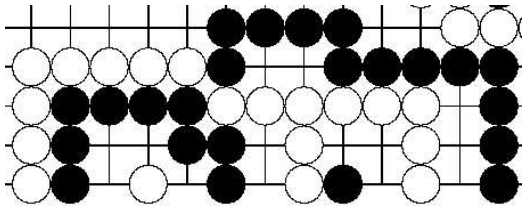

Figura 4.31: *Semeai* de clase 1[10].

En el proceso de matar un grupo en *semeai* con un gran ojo, la carrera puede pasar de ser de clase 1 a clase 2 en más de una ocasión. No queda claro en el artículo cuál es la ventaja de mantener una separación conceptual entre estas dos clases. Es por ello que nuestra clase 1 engloba las clases 1 y 2 de Müller. La única contrapartida de esta aproximación es que deberemos contemplar una casuística mayor para englobar las situaciones *unsettled*.

[8] Aunque es una heurística innecesaria para carreras estrictamente de clase 0 (decideStatus no comete errores) puede ser de utilidad en situaciones en que se detecte erróneamente un *semeai* de clase 0 sin serlo.

[9] Un ojo es *plain* si está rodeado por un sólo grupo y todas las intersecciones vacías son adyacentes a ese grupo [30].

[10] STS-RV; fi chero semeais_1.tst, tests 5 y 6.

[11] STS-RV; fi chero semeais_1.tst, tests 43 y 44.

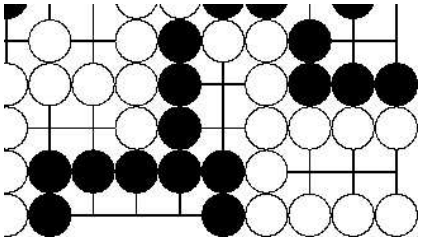

Figura 4.32: *Semeai* de clase 1 (clase 2 de Müller)[11].

En el desarrollo del módulo *Semeai-01ES* el ciclo de vida de la clase 1 fue extremadamente corto. Tras la implementación de la clase 1 rápidamente desarrollamos las ideas presentadas en la sección 4.2 que dieron lugar a la clase e. La clase e cubre un conjunto de casos mucho mayor, que incluyen todas las situaciones que se resolvían con la clase 1. Por ello procederemos directamente a la discusión de la clase e.

4.3.4 Semeai Clase e

$e = 2{,}71828182845904523536028 7\ldots$ es un número ligeramente mayor que 2 y por ello da nombre a la presente clase; por cubrir un conjunto de casos ligeramente mayor que los *semeais* de clase 2 de Müller. Entendemos por *semeai* de clase e todos los de clase 1 y 2 más todos aquellos que tengan un gran ojo según la definición dada en la sección 4.2.1.III. Un ojo rodeado por un sólo grupo, pudiendo tener más de un grupo no esencial del oponente en el interior, un grupo no esencial propio o intersecciones vacías no adyacentes al grupo que rodea al ojo. Es una generalización del concepto de *plain eye* definido por Müller en [30].

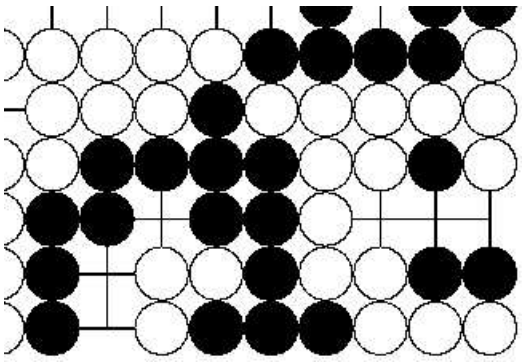

Figura 4.33: *Semeai* de clase e[12].

La clase e se implementa como clase derivada de la clase 0 (cf. sección 4.4) añadiendo tan sólo un nuevo contador de libertades para cada grupo (cuando hay ojos las libertades reales, número de turnos que el oponenete necesita para matar al grupo, son inferiores a las libertades físicas, intersecciones adyacentes vacías) y un objeto de la clase Eye para cada grupo.

La clase Eye juega un papel determinante en la resolución de *semeais* de clase e y ambas están íntimamente relacionadas. Las ideas presentadas en la sección 4.2 se utilizarán intensivamente en el desarrollo de este apartado.

Al igual que con la clase 0, el núcleo central de la clase e lo conforman los métodos `init`, `decideStatus` y `playMove`.

El método `init`, se encarga de la inicialización de las variables de clase y, muy especialmente, de inicializar los dos objetos de clase Eye. Existe también un método `InitMinor` que se encarga de incializar tan sólo aquellas variables requeridas para poder ejecutar la clase p (cf. sección 4.3.5).

`decideStatus` utiliza toda la información proporcionada por la clase Eye para determinar el estatus de la carrera. En la tabla 4.5 se muestra, en función de los cuatro posibles estatus de ojo de cada jugador, el movimiento y los dos estatus de la carrera. El movimiento puede ser pasar (P) o jugar (Y), y el estatus puede ser ganado (W), perdido (L) o *seki* (S).

PLAYER EYE STATUS	OTHER EYE STATUS	(MOVE,STATUSIFPLAY,STATUSIFPASS)
Nakade	Nakade	`libertyStudy`
	Unsettled	`libertyStudy`
	Alive	(P,L,L)
	AliveInAtari	(P,L,L) / (Y,W,L)
Unsettled	Nakade	`libertyStudy`
	Unsettled	`libertyStudy`
	Alive	(Y,S,L)
	AliveInAtari	(Y,S,L) / (Y,W,S)
Alive	Nakade	(P,W,W)
	Unsettled	(Y,W,S)
	Alive	(P,S,S)
	AliveInAtari	(P,S,S) / (Y,W,S)
AliveInAtari	Nakade	(P,W,W) / (Y,W,L)
	Unsettled	(Y,W,S) / (Y,S,L)
	Alive	(P,S,S) / (Y,S,L)
	AliveInAtari	(P,S,S) / (Y,W,S) / (Y,S,L)

Tabla 4.5: *Semeai* status en función de los ojos de cada grupo

Si el estatus de ojo es `AliveInAtari` la primera terna indica el resultado si el

[12] STS-RV; fi chero semeais_e.tst, tests 1 y 2.

grupo que posee el ojo con dicho estatus no está en *atari* y la segunda terna el caso contrario. En la última fila, la primera terna supone que ninguno está en *atari*, la segunda supone que *other* lo está, y la tercera que *other* no lo está y *player* sí.

Hay cuatro combinaciones de estatus que no tienen una terna como resultado. El estatus en esas situaciones necesita un análisis pormenorizado de las libertades y el tamaño de los ojos para cada grupo. Por ello se llama al método `libertyStudy` que se encarga de realizar esa tarea. El funcionamiento de este método se resume en las tablas 4.6 y 4.7.

(Move, StatusIfPlay, StatusIfPass)					
$sl = 0$		$sl = 1$		$sl \geq 2$	
		$k > 1$	(P,W,W)	$k > sl$	(P,W,W)
$k \geq 1$	(P,W,W)	$k = 1$	(Y,W,S)	$k = sl$	(Y,W,S)
$k = 0$	(Y,W,L)	$k = 0$	(P,L[13],S)	$sl > k > -sl$	(P,S,S)
$k \leq 1$	(P,L,L)	$k = -1$	(Y,S,L)	$k = -sl$	(Y,S,L)
		$k < -1$	(P,L,L)	$k < -sl$	(P,L,L)

Tabla 4.6: Status si ambos ojos tienen el mismo MüellerStatus

Si ambos ojos tienen el mismo `MullerStatus` el movimiento y estatus de la carrera vienen dados por la tabla 4.6. Al igual que para la clase 0, *sl* (las libertades compartidas) y $k = playerlibs - otherlibs$ determinan por completo el resultado. Nótese que ahora las libertades de un grupo son: sus libertades exteriores sin contar las compartidas más las que proporciona el ojo (si lo tienen) según la tabla 4.1.

(Move, StatusIfPlay, StatusIfPass)				
$LibsAtt = LibsDef$				(Y,W,L)
$LibsAtt > LibsDef$	Att=CToPlay[14]	$eyeOther = Uns.$	$eyePlayer = Nakade$	(Y,W,L)
			$eyePlayer \neq Nakade$	(Y,W,S)
		$eyeOther \neq Uns.$		(P,W,W)
	Def=CToPlay			(P,L,L)
$LibsAtt < LibsDef$	Att=CToPlay			(P,L,L)
	Def=CToPlay	$eyeOther = Uns.$	$eyePlayer = Nakade$	(Y,W,L)
			$eyePlayer \neq Nakade$	(Y,W,S)
		$eyeOther \neq Uns.$		(P,W,W)

Tabla 4.7: Status si los ojos tienen distinto `MullerStatus`.

[13] Si `eyePlayer==Unsettled` a considerar a parte.

[14] CToPlay indica el color que tiene el turno de juego.

Si el `MullerStatus` es diferente el que lo tenga menor es el *Attacker* y el otro el *Defender*. Las libertades de *Attacker* son las exteriores sin contar las compartidas más las que proporciona el ojo. Para *Defender* se añaden además las libertades compartidas por tener un `MullerStatus` mayor[15]. A partir de aquí la tabla 4.7 nos muestra el movimiento y el estatus para cada posible situación. Si leemos la tabla desde la esquina superior derecha hasta la inferior izquierda moviéndonos de derecha a izquierda y de arriba abajo obtendremos la secuencia `if-else` que implementa la función `libertyStatus`.

PlayMove			
PLAYEREYE	OTHEREYE	INTERSECCIÓN	
		$other.nbLib = 1$	capturar
		$player.nbLib = 1$	playerEye.VP
Unsettled	Nakade		playerEye.VP
	Unsettled	$statusIfPlay = W$	otherEye.VP
		$statusIfPlay \neq W$	playerEye.VP
	AliveInA		playerEye.VP
	Alive		playerEye.VP
Alive	Unsettled		otherEye.VP
AliveInA	Unsettled		otherEye.VP
Nakade	Unsettled		otherEye.VP
	Nakade	Algoritmo Nakade vs Nakade	

Tabla 4.8: Intersección a jugar según estatus de ojos

Una vez determinado el movimiento y el estatus, el método `playMove` calculará la intersección a jugar. En la tabla 4.8 se muestra el algoritmo implementado por este método. Leyendo la tabla de arriba a abajo obtendremos la secuencia `if-else` implementada. En la tabla VP indica el punto vital del ojo. Si estamos en un caso *Nakade vs Nakade* el movimiento a jugar será por este orden: una libertad exterior the *other*, si no hay se deberá rellenar una libertad dentro del ojo de *other* y si todas menos una están jugadas se deberá jugar una libertad compartida. Nótese que todo esto se realizará sólo si el método *decideStatus* nos ha devuelto un movimiento `Play`.

4.3.5 Semeai Clase p

La clase p constituye el punto crítico de todo el módulo *Semeai-01ES* . Una vez construida la clase e, llamemos $\mathcal{P}$ al conjunto de posiciones que dicha clase evalúa correctamente. Cuando utilicemos un algoritmo de búsqueda para evaluar una posición $p \notin \mathcal{P}$ deberemos saber, en cada nodo del árbol, si hemos llegado ya a

[15]Método propuesto por Richard Hunter en [22]

una posición $p' \in \mathcal{P}$ para llamar a la función de evaluación (la clase e) y etiquetar dicho nodo. Ese es el cometido de la presente clase.

Hasta ahora todo el código implementado se ha basado en un estudio teórico profundo y está libre de toda heurística. Es completamente seguro en cuanto a que no deja sin contemplar ningún caso. Sin embargo ahora se va a introducir cierta heurística.

El conjunto $\mathcal{P}$ además de ser inmenso es muy heterógeneo. Conseguir caracterizar todos sus elementos con una lista de reglas concreta y reducida es sumamente complicado. Sea $\mathcal{Q}$ el conjunto de posiciones que la clase p identifica como evaluables por la clase e, nuestro objetivo será:

1. $\mathcal{Q} \cap \mathcal{P} = \mathcal{Q}$
2. $\mathcal{P} - \mathcal{Q}$ mínimo.

La primera condición es fundamental, en lenguaje coloquial nos dice que "más vale quedarse corto que pasarse". Esto es así porque no identificar una posición evaluable como tal es relativamente grave; el árbol continuará creciendo por una rama que ya podría haberse etiquetado. Sin embargo identificar como evaluable una posición que no lo es tiene consecuencias demoledoras; la posición se evaluará de manera errónea casi con total seguridad y el análisis de la carrerá caerá como un castillo de naipes. La segunda condición simplemente nos anima a afinar al máximo nuestra caracterización del conjunto.

Y, ¿qué es lo que hace tan difícil caracterizar al conjunto $\mathcal{P}$? Básicamente el concepto de "completamente rodeado" que caracteriza a los *semeais* de clase e, es el más complejo de determinar. No sólo se trata de ver que no hay salida al exterior, sino también de ver que ninguno de los grupos que rodean es capturable.

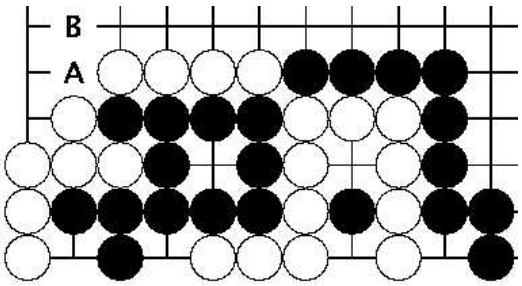

Figura 4.34: Este *semeai* no es de clase e.

Hay otros aspectos que dificultan esta caracterización. En la carrera de la figura 4.34 nada parece indicar a primera vista que no sea un *semeai* de clase e. Si evaluamos esta posición con la clase e obtendremos el resultado negro juega y

consigue *seki* y blanco juega y gana. Sin embargo negro jugando en A consigue una libertad extra y el verdadero estatus es, tanto para negro como para blanco, pasar y *seki*. Así, en el ejemplo, si blanco tuviera una piedra en A ó B se trataria de una posición evaluable mientras que tal y como está la posición no pertenece a $\mathcal{P}$; un detalle completamente externo a la localidad de la carrera nos determina la respuesta de la clase p.

La clase p contempla las siguientes situaciones:

1. MÁS DE UN GRUPO ESENCIAL Si alguno de los dos bloques puede conectar en una sola jugada con otro grupo del tablero $p \notin \mathcal{P}$

2. EXISTENCIA DE LIBERTADES NO PLAIN Si hay libertades que requieren más de un movimiento por parte del adversario para ser ocupadas $p \notin \mathcal{P}$.

3. GRUPOS ADYACENTES EN ATARI La captura de dichos grupos puede alterar de manera abrumadora el estatus de la carrera. Es necesario implementar búsqueda arborescente.

4. EXISTENCIA DE UN "PASILLO" POR SEGUNDA LÍNEA Si hay más de dos piedras por segunda línea con sus correspondientes intersecciones por primera línea vacías, el orden de los movimientos puede ser importante y el estatus puede verse alterado.

5. EXISTENCIA DE UN GRUPO DÉBIL RODEANDO AL GRUPO EN SEMEAI Permite la posibilidad de ganar libertades jugando fuera de la zona local de la carrera.

6. NO COMPLETAMENTE RODEADO Si hay posibilidad de ganar libertades exténdiendose del grupo en *semeai* o bien es posible conectar con el exterior en más de una jugada $p \notin \mathcal{P}$.

Este listado de reglas, sin ser completo ni ser producto de un estudio teórico definitivo, cumple de manera satisfactoria los dos objetivos propuestos para la clase p.

La clase p, además de determinar si una posición es evaluable se encarga de generar los movimientos posibles para una posición dada, como parte integrada del algoritmo de búsqueda.

La implementación de la clase p se realiza, de nuevo, como clase derivada de la clase e. Se añaden las variables correspondientes para poder llevar recuento de las seis condiciones mencionadas y cada una de ellas se computa con un método propio.

El método `init` inicializa todas las variables relativas al *semeai*: bloques esenciales, libertades compartidas, turno de juego, ... En cuanto a los ojos `init` llama

al método `initmin` de la clase Eye que sólo inicializa un conjunto imprescindible de variables relativa al ojo (espacio de ojos, bloques no esenciales en el interior,...) pero no se preocupa de calcular el estatus. Esto último se reserva para el caso en que la posición sea evaluable, de esta manera se ahorra tiempo de ejecución en cada nodo que resulta no pertenecer al conjunto $\mathcal{P}$. A continuación se llama, de manera sucesiva, a cada uno de los seis métodos que comprueban el cumplimiento de las reglas que definen a $\mathcal{P}$.

`findLinked` comprueba la existencia de grupos conectables al bloque esencial en una jugada. Para ello recorre todas las libertades del grupo esencial, para cada una de ellas busca vecinos que sean del mismo color. Si tal vecino existe y no es una piedra del propio grupo esencial ni se trata de una intersección del ojo el flag de más de un bloque esencial se activa.

En la figura 4.35, si consideramos como bloques esenciales A y B, `findLinked` se activaría tanto para A y C como para B y D. La carrera no es de clase e.

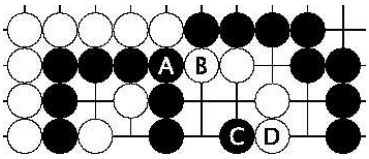

Figura 4.35: Más de un grupo esencial para ambos.

`findLibsNotPlain` es un método más complicado de lo que parece. Una libertad es *plain* si puede ser ocupada por el adversario en una sola jugada. Como es habitual en Go, tras una definición sencilla, se esconden multitud de situaciones complicadas. Parece que una manera fácil de implementar este método sería comprobar que cada una de las libertades del grupo esencial puede ser jugada por el adversario sin que ninguno de sus grupos quede en *atari*.

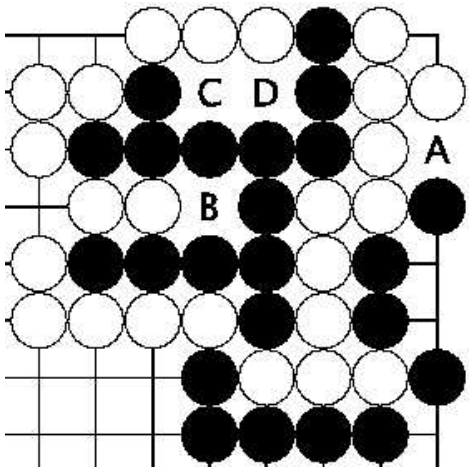

Figura 4.36: Libertades no *plain*.

Sin embargo la figura 4.36 nos muestra que tendremos que sofisticar un poco más nuestro algoritmo. A y B son libertades no *plain* y serían detectadas de manera satisfactoria por el algoritmo propuesto. En cambio, tanto C como D pueden ser individualmente ocupadas por blanco sin ningún problema, pero no simultáneamente.

El algoritmo constará de una doble pasada; en la primera se detectarán las libertades no *plain* individuales. A continuación agruparemos las libertades en grupos conexos y, tras ocuparlas todas de golpe, miraremos si estamos o no en *atari*.

`findAdjacentsAtari` simplemente comprueba para cada uno de los grupos

adyacentes a los bloques esenciales, que tengan más de una libertad.

Un pasillo en segunda línea altera en gran medida la clase de un *semeai*. Las carreras que aparecen en las figuras 4.37 y 4.38 presentan las mismas características en cuanto a ojos (no hay), libertades exteriores (cuatro para cada grupo) y libertades compartidas (no hay). Si sólo atendieramos a estas características cometeríamos el error de clasificarlas como clase 0. Sin embargo la carrera de la figura 4.38 transcurre en terreno pantanoso: la segunda línea. Por ello esta carrera no es de clase 0, ni tan siquiera de clase e.

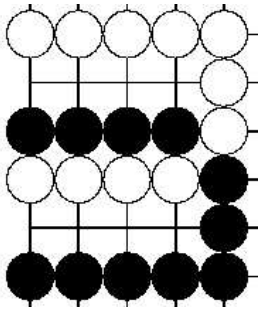

Figura 4.37: *Semeai* de clase 0.

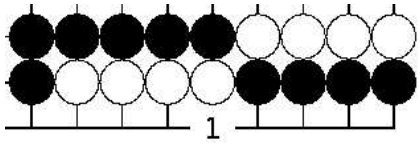

Figura 4.38: *Semeai* no es de clase e.

Por un lado existe la posibilidad de crear un ojo con una sola jugada, y por otro, un *hane* en primera línea puede llevar en muchas ocasiones a una situación de *ko*. Así mientras que en el primer *semeai* blanco puede jugar cualquiera de las libertades de negro para ganar y puede resolverse estáticamente, en el segundo blanco debe jugar en 1 para ganar y este movimiento sólo podrá encontrarse realizando una búsqueda en árbol.

`findSecondLineCorridors` activa el correspondiente flag si encuentra un pasillo en segunda línea de dos o más piedras. El procedimiento es el siguiente: en primer lugar almacena en una `Rzone` todas las piedras del bloque esencial que esten en segunda línea, a continuación elimina las que no corresponden a pasillos y por último busca si quedan dos o más piedras, de las candidatas iniciales conectadas. Las piedras que son descartadas responden a tres motivos: i) en primera línea hay una piedra del mismo color ii) en primera línea hay una piedra de color opuesto que pertenece a un bloque seguro o al otro grupo en *semeai* iii) en primera línea hay una intersección vacía que pertenece a un ojo.

Entendemos por grupo débil aquel que, por su estructura (sin ojos y sus libertades en segunda o primera línea), permite al adversario ganar un tiempo en la carrera. En la figura 4.39 blanco tiene tres libertades y sólo necesita dos tiempos para matar a negro. Parece que la carrera está decidida independientemente del turno de juego. Sin embargo, blanco tiene un grupo débil rodeando a negro. Si negro juega en A blanco necesita ahora cuatro tiempos para ganar la carrera al no poder dar *atari* a negro hasta no haber capturado la piedra negra en A. Negro gana la ca-

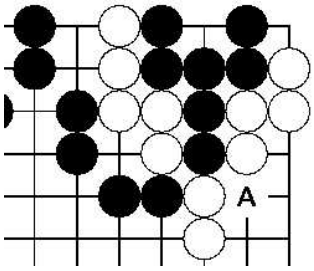

Figura 4.39: Negro en A gana la carrera[16].

[16]Problema 173 pág. 59 [11]. STS-RV fi chero semeais_GSAT.tst problemas 53 y 54.

rrera. La carrera presentada no es de clase e debido a la existencia del movimiento en A que modifica la cuenta de libertades efectivas.

`findWeakSurroundingGroups` se encarga de detectar este tipo de situaciones y evitar que, en estos casos, se llame a la función de evaluación. La caracterización heurística de estos grupos se basa en los siguientes aspectos: ser un grupo adyacente al grupo en *semeai*, no tener ojos, tener todas sus libertades en primera o segunda línea, tener un máximo de dos libertades no compartidas con el grupo en *semeai* y que una de ellas no esté protegida (que al colocar una piedra enemiga no pueda ser capturada en un movimiento).

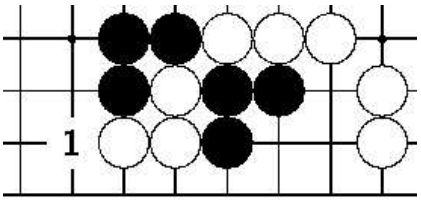

Figura 4.40: Blanco en 1 escapa.

El último punto de la lista es determinar si la carrera está completamente rodeada y es a su vez el más complejo de determinar. `isCompSurrounded` implementa dos ideas y ambas encierran cierto grado de heurística. La primera activará el flag si es possible extenderse del grupo en *semeai* de manera que la piedra añadida tenga tres libertades. En la figura 4.40 si no detectamos la posibilidad de escape y evaluamos directamente la carrera obtendríamos que negro gana sin necesidad de jugar. Al detectar que no está completamente rodeado la búsqueda se inicia y se detecta 1 como movimiento ganador para el que primero juegue.

La segunda idea consiste en determinar si hay grupos conectables, según lo estipulado en el método `findLinked`, tras una jugada; es decir, si hay algún grupo conectable en dos jugadas. En la figura 4.41 pasa lo mismo que en el caso anterior. Si obviamos la posibilidad de conexión y evaluamos directamente obtendríamos, de manera errónea, que blanco gana sin necesidad de jugar. Tras detectar la posiblidad de conexión y activarse el flag correspondiente, la búsqueda se inicia y determina A como movimiento ganador para el que primero juegue.

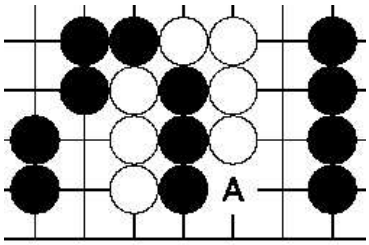

Figura 4.41: Negro en A conecta con el grupo amigo.

Por último la clase p se encarga, en cada nodo, de generar todos los posibles movimientos a jugar en el siguiente nivel. La generación de movimientos debe ser por un lado altamente selectiva (para evitar explorar ramas que no llevan a ningún sitio) y por el otro suficientemente exhaustiva para estar seguros de que el movimiento ganador es generado (si no es así dificilmente lo encontraremos!!). Estamos de nuevo ante un compromiso de difícil solución.

ORDEN	MOVIMIENTO
1	Si `playerEye==Unsettled` playerEye.VP
2	Si `otherEye==Unsettled` otherEye.VP
3	Libertades del grupo oponente
4	Si `AdjacentAtariPlayer` Libertades de los grupos adyacentes a Player con 2 ó menos libertades y libertad de los grupos adyacentes en atari a los adyacentes de Player
5	Si `NPlainLibsOther` Movimientos de aproximación a las libertades no plain de Other
6	Segundas libertades[17] de Other
7	4 para Other
8	5 para Player
9	Libertades de Player
10	Segundas libertades de Player
11	4 Si no se cumple la condición
12	5 Si no se cumple la condición
13	7 Si no se cumple la condición
14	8 Si no se cumple la condición

Tabla 4.9: Orden de los movimientos generados.

`calculateMoves` genera los movimientos contemplados en la tabla 4.9 en el orden en el que aparecen. El orden en que se prueban los movimientos es un factor muy importante para conseguir un nivel de poda elevado en el algoritmo Alpha-Beta. La tabla refleja la prioridad que hemos dado a cada movimiento.

Los resultados obtenidos por la clase p alteran de forma acorde el orden de los movimientos. Si se han detectado grupos adyacentes en atari, se probará primero las libertades de esos grupos adyacentes, si se han detectado libertades no *plain* daremos prioridad a los movimientos de aproximación. Si no es así, los movimientos generados en 4,5,7 y 8 pasarán respectivamente a las posiciones de 10 a 14.

En la posición 4 (o la 11 si no se cumple la condición) la decisión de tomar las libertades de los adyacentes que tengan dos o menos libertades es heurística. ¿Por qué no tres? De hecho el no tomar tres nos imposibilita resolver el problema 14 de [22] pág. 191, ya que la solución pasa por jugar un libertad de un grupo adyacente con tres libertades. Aún así hemos estimado que dos libertades es un buen límite ya que tres incrementa demasiado el árbol de búsqueda y una nos lleva a descartar la jugada correcta en numerosas situaciones.

En la figura 4.42 se muestran, numerados según el orden en que han sido generados y van a ser ejecutados, los posibles movimientos para negro (el movimiento 2 se descarta a posteriori por ser ilegal).

[17] Las segundas libertades de un grupo son las libertades de sus libertades.

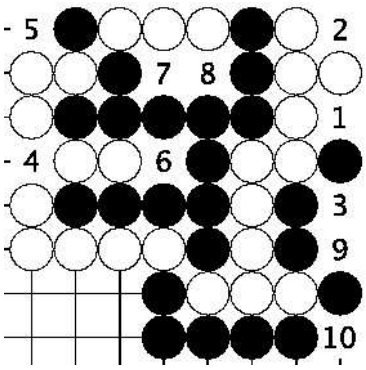

Figura 4.42: Movimientos generados numerados según su orden de prioridad[18].

Por último se ejecuta la función `calculateMovesRemove` que se encarga de detectar y suprimir movimientos innecesarios. En primer lugar se encarga de suprimir todas las jugadas realizadas en el interior de un ojo grande y de añadir únicamente el punto vital. En la sección 4.2.2 veremos cómo la clase Eye nos indica, independientemente del número de piedras (amigas o enemigas) jugadas en su interior, cuál es la intersección a jugar. Por otro lado se encarga de detectar pasillos de libertades equivalentes y suprimir todas las libertades equivalentes menos una.

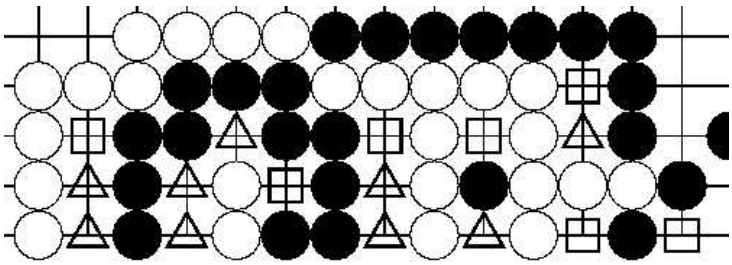

Figura 4.43: Movimientos generados □ y △, movimientos suprimidos △.

En la figura 4.43 se muestran los movimientos generados por la función `calculateMoves` y los movimientos suprimidos a posteriori por la función `calculateMovesRemove`. Los movimientos suprimidos responden tanto a intersecciones interiores a un gran ojo como a pasillos de libertades equivalentes.

[18]STS-RV; fi chero semeais_RH.tst, tests 11 y 12. Corresponde al problema 7 [22] pág. 56.

4.3.6 Semeai Clase s

La clase s es la encargada de implementar el algoritmo de búsqueda. Al igual que las anteriores se deriva de su predecesora, la clase p. Gracias a la integración del módulo con la librería TAIL, solamente es necesario definir el método `evaluation` y hacer pequeñas modificaciones en la plantilla base `Game` (cf. sección 4.1.2). A continuación, la activación de ciertos flags nos permitirá tener en marcha un completo algoritmo de búsqueda alpha-beta con tabla de transposiciones, *killing move* y ordenación de movimientos según *history heuristic*.

Los dos métodos principales de esta clase son `init` y `evaluation`. `Init` simplemente incializa la clase p y algunas variables de control de la clase s. El grueso de esta clase se centra en la función `evaluation`. En primer lugar este método llama a la clase p para decidir si se encuentra ante una posición evaluable. En caso afirmativo la clase e decide el estatus y el movimiento a jugar.

VALOR	15003	15002	15001	3	2	1	-15001	-15002	-15003
STATUS	WP	W	KW	SP	S	KS	U	KL	L

Tabla 4.10: Posibles valores de retorno de la función de evaluación.

La función de evaluación devuelve al algoritmo de búsqueda uno de los resultados que aparecen en la tabla 4.10: ganar pasando, ganar jugando, ganar en *ko*, *seki* pasando, *seki* jugando, *seki* en *ko*, *unknown*, perder en *ko* o perder.

Si la posición no es de clase e se intenta decidir de manera heurística el estatus de la carrera y el movimiento a jugar. Algunas de estas heurísticas son fruto de información procedente de la clase p. En la tabla 4.11 se resumen las heurísticas implementadas.

- **H0** A pesar de la simplicidad de esta heurística, es una de las más intensamente utilizadas junto con H1 y permite ahorrar un nivel de profundidad en la obtención del resultado de la carrera. Si al color al que le toca jugar encuentra a su oponente en *atari* gana la carrera jugando en la última libertad.

- **H1** H1 invierte la idea de H0 y permite ganar dos niveles de profundidad. Si el color que juega está en *atari*, no puede ganar libertades (i.e. no puede capturar ningún grupo adyacente y jugando en su última libertad no gana libertades adicionales) y su oponente no está en atari la carrera está perdida para él.

- **H2** H2 corrige un exceso de celo de la clase p. Si un grupo tiene una unica libertad no *plain* y ni tiene ojos ni comparte libertades con el oponente la cuenta de libertades reales no se ve afectada y el algoritmo de la clase e

obtiene el resultado correcto. Simplemente hay que preocuparse de que la libertad no *plain* sea la última en ser jugada.

- **H3** H3 utiliza la información contenida en `TwoEyes`, un flag de la clase Eye que se activa cuando el gran ojo tiene una piedra amiga en su interior que le proporciona dos ojos seguros. Sabiendo que uno de los dos grupos está incondicionalmente vivo decidir el estatus de la carrera se convirtie en una tarea trivial.

- **H4** H4 y H5 no son propiamente heurísticas sino contadores y están relacionados con el método `calculateMovesRemove` de la clase p. En cada gran ojo hay tantos movimientos posibles como intersecciones vacías pero sólo uno es el punto vital. Si la clase Eye lo ha detectado todos los demás se suprimen habiendo así menos jugadas posibles a ser exploradas en el algortimo de búsqueda. H4 se incrementa por cada jugada descartada.

- **H5** Equivalentemente, H5 hace lo mismo que H4 pero con los pasillos de libertades equivalentes.

H0	si el oponente está en *atari* y tenemos el turno
H1	turno del oponente, éste en *atari* y nosotros no
H2	solo una libertad no *plain*, no hay ojos y no hay libertades compartidas
H3	información en `TwoEyes`
H4	reducción de movimientos en el ojo
H5	reducción de movimientos por equivalencia de pasillo

Tabla 4.11: Relación de heurísticas utilizadas por la clase s.

En el apéndice E se puede consultar la intensidad de utilización de estas heurísticas en cada uno de los tests de la STS-RV. En la sección 5.3.1 se comparan los resultados obtenidos por *Semeai-01ES* con y sin heurísticas y a la luz de esos resultados podemos afirmar que el nivel del módulo mejora significativamente gracias a ellas.

Por último comentar que hemos catalogado el estatus *seki* como «no terminal». Esto significa que el algortimo de búsqueda a pesar de haber detectado un movimiento que garantice *seki* continuará buscando hasta agotar el límite de nodos explorables o la profundidad de búsqueda máxima. Esto se debe a que es frecuente que en la exploración de una posición ganable, se encuentre antes un movimiento que derive en *seki*. Si este estatus fuera terminal obtendríamos respuestas subóptimas en muchos casos. La contrapartida de esta decisión es que todos los tests con resultado *seki* presentan una pésima eficiencia temporal.

4.4 Código de las definiciones de Clase

A continuación presentamos las cabeceras de las clases comentadas en el presente capítulo.

```
class SemiaiClass_0 : public BasicGoGame
{
 protected:
  String *essentialBlockPlayer, *essentialBlockOther;
  Rzone sharedLiberties;
  SemiaiStatus status;
  int nbOutsideLibsEBP, nbOutsideLibsEBO;
  int localSemiaiWinningValue;  /* Deiri Counting, Counts the value of winning vs loosing it*/
  int localSemiaiSekiValueP;
  int localSemiaiSekiValueO;* How much is worth achieving a seki than dying for each player */
  int colorToPlay;          /* not very accurate in class 1 if one of them makes two eyes.   */
  toPlay_t move;

  GoMove intersectionToPlay;
  GoMove intersectionToPlay2; /* When using search to have an alternative to nplain liberty */

public:

  SemiaiClass_0(IncrementalGoBoard *b=NULL);

  /* Init all variables except status and move*/
  void init( int stoneA=0, int stoneB=1, int turn=White, IncrementalGoBoard *b=NULL );

  /* Returns the status for the color to Play and stores in move if it is necessary to play
     to achieve the status or Pass is possible */
  void decideStatus();

  /* If move is Play then stores and returns the intersection number to be played.
     If not it returns Pass=400*/
  GoMove playMove();

  /* Print debugging info */
  void debugSemiai();

  int getLocalSemiaiWinningValue() { return( localSemiaiWinningValue ); }
  int getLocalSemiaiSekiValueP() { return( localSemiaiSekiValueP ); }
  int getLocalSemiaiSekiValueO() { return( localSemiaiSekiValueO ); }
  SemiaiStatus getStatus() { return( status ); }
  GoMove getIntersectionToPlay() { return( intersectionToPlay ); }
  void setBoard( IncrementalGoBoard *b ) { board = b; return;}

};
```

Figura 4.44: La clase semeai 0

```
class SemiaiClass_1 : public SemiaiClass_0
{
  int nbTotalLibsEBW, nbTotalLibsEBB; /* Outside + Eye Libs*/
  Eye whiteEye, blackEye;
  String *nonEssentialBlockWhite, *nonEssentialBlockBlack;

public:
  SemiaiClass_1(IncrementalGoBoard *b=NULL);

  /* Init all variables except status and move*/
  void init( int stoneA=0, int stoneB=1, int turn=White, IncrementalGoBoard *b=NULL );

  /* Returns the status for the color to Play and stores in move if it is necessary
     to play to achieve the status or Pass is possible */
  void decideStatus();

  /* To be called by decideStatus when necessary.
     Performs the liberty study to decide the status and move. */
  void libertyStudy( eyeStatus_t eyePlayer, eyeStatus_t eyeOther );

  /* If move is Play then stores and returns the intersection number to be played.
     If not it returns Pass=400*/
  int playMove();

  /* Print debugging info */
  void debugSemiai();

  /* Get private variables */
  Eye getWhiteEye() { return( whiteEye ); }
  Eye getBlackEye() { return( blackEye ); }

};
```

Figura 4.45: La clase semeai 1

```
class SemiaiClass_e : public SemiaiClass_0
{
 protected:
  int nbTotalLibsEBP, nbTotalLibsEBO; /* Outside + Eye Libs*/
  Eye playerEye, otherEye;

public:

  SemiaiClass_e(IncrementalGoBoard *b=NULL);

  /* Init all variables except status and move*/
  returnStatus_t init( int stoneA=0, int stoneB=1, int turn=White, IncrementalGoBoard *b=NULL );

  /* If init in higher class p has been called only necessary to init those who
     hasnt been initialized yet */
  returnStatus_t initMinor();

  /* Returns the status for the color to Play and stores in move if it is necessary to play
     to achieve the status or Pass is possible */
  returnStatus_t decideStatus();

  /* To be called by decideStatus when necessary.
     Performs the liberty study to decide the status and move. */
  returnStatus_t libertyStudy( eyeStatus_t eyePlayer, eyeStatus_t eyeOther );

  /* If move is Play then stores and returns the intersection number to be played.
     If not it returns Pass=400*/
  returnStatus_t playMove();

  /* Print debugging info */
  void debugSemiai();

  /* Get private variables */
  Eye getPlayerEye() { return( playerEye ); }
  Eye getOtherEye() { return( otherEye ); }

};
```

Figura 4.46: La clase semeai e

```
class SemiaiClass_p : public SemiaiClass_e
{
 protected:
  StringArray playerLinked;                   /* linked  string for player */
  StringArray otherLinked;                    /* linked  string for other */
  Rzone playerLibsNotPlain, otherLibsNotPlain; /* not plain libs for player and other all of them mixed */
  Rzone playerLibsINotPlain, otherLibsINotPlain; /* not plain libs individual. */
  int playerNbSLibsNP, otherNbSLibsNP;        /* nb of strings of libs not plain */
  int playerNbILibsNP, otherNbILibsNP;        /* nb of single libs not plain */
  int playerNbAdjAtari, otherNbAdjAtari;      /* nb of adjacent strings in  atari for both */
  Rzone secondLCPlayer, secondLCOther;        /* second line corridors for player & other */
  Rzone winLibsPlayer, winLibsOther;          /* moves that let win libs to player & other */
  Rzone movesSemiai;               /* empty intersections candidate moves. decided heuristically */

  myBool_t isCompletelySurrounded;
  Rzone player3LibWinningMoves, other3LibWinningMoves;
  StringArray playerLinkedInOne, otherLinkedInOne;

  isClassE_t isClassE;

public:

  /*Constructor call init functions*/
  SemiaiClass_p(IncrementalGoBoard *b=NULL);

  /* Init all variables except isClassE*/
  returnStatus_t init( int stoneA=0, int stoneB=1, int turn=White, IncrementalGoBoard *b=NULL );

  /* Check for adjacent strings in atari */
  returnStatus_t findAdjacentsAtari( String *essentialBlock, String *essentialBlock2 );

  /* find linked Strings */
  returnStatus_t findLinked( String *essentialBlock, StringArray &Linked );

  /* find libs not plain */
  returnStatus_t findLibsNotPlain( String *essentialBlock );

  /*find second line corridors */
  returnStatus_t findSecondLineCorridors( String *essentialBlock);

  /*find weak surrounding groups on the second line*/
  returnStatus_t findWeakSurroundingGroups( String *essentialBlock);

  /* calculate all possible moves for semiai */
  returnStatus_t calculateMovesSemiai( int col=-1 );

  /* remove equivalent libs for moves for semiai */
  returnStatus_t calculateMovesSemiaiRemove( const String *Block );

  /* Decide if it is completely sorrounded. 3 lib winning move + linked in one move */
  returnStatus_t isCompSurrounded();

  /* Print debugging info */
  void debugSemiai();

  /* Get private variables */
  StringArray getPlayerLinked() { return( playerLinked ); }
  StringArray getOtherLinked() { return( otherLinked ); }
  Rzone getPlayerLibsNotPlain() { return( playerLibsNotPlain ); }
  Rzone getOtherLibsNotPlain() { return( otherLibsNotPlain ); }
  Rzone getMovesSemiai() { return( movesSemiai); }
  isClassE_t getIsClassE() { return( isClassE ); }
};
```

Figura 4.47: La clase semeai p

```
class SemiaiClass_s : public SemiaiClass_p{
  int inter1, inter2;
  int MaxColor;  //not necessary equivalent to color in BasicGoGame
 public:
  Rzone m_trace;

  SemiaiClass_s( IncrementalGoBoard *b, int interplayer=-1, int interother=-1 );
  void init( IncrementalGoBoard *b, int interplayer, int interother );

  int tryMove (const GoMove & _move, Rzone *trace = NULL);
  void popMove ();

  short evaluation (int colorToPlay, GoMove & bestMove,
    Rzone *trace = NULL, Rzone *fail_trace = NULL);
  short evalKo (short int eval);

  int minMoves (MoveSet <GoMove> & moves, int order = 0,
Rzone * z = &rzoneTemp, Rzone * fail_z = &rzoneTemp);
  int maxMoves (MoveSet <GoMove> & moves, int order = 0,
Rzone * z = &rzoneTemp, Rzone * fail_z = &rzoneTemp);

  int minMovesKill (MoveSet <GoMove> & moves, int order = 0,
    Rzone * z = &rzoneTemp, Rzone * fail_z = &rzoneTemp);
  int maxMovesKill (MoveSet <GoMove> & moves, int order = 0,
    Rzone * z = &rzoneTemp, Rzone * fail_z = &rzoneTemp);

  short minEval ();
  short minKEval();
  short maxEval ();
  short oppositeResult (short eval);
  int isTerminal (short int eval);
  int modifiable (short int eval);

  int maxColor() { return MaxColor;}
  int minColor() { return board->other(MaxColor);}

  inline static void makeSgfTrace (short int res, short int eval, int depth,
   int alpha, int beta,
   unsigned long long nbMoves, int nbNodes,
   const string& game_id);
  inline void endNode (short int res, short int eval, int depth,
       int alpha, int beta,
       unsigned long long nbMoves, int nbNodes = 0);
  void debugSemiaiInSgf();
};
```

Figura 4.48: La clase semeai s

```
class Eye : public BasicGoGame
{
  char Color;
  int Turn;
  String *owner;
  Rzone EyeSpace,EyeFilledSpace, EyeFilledFriendSpace;
  int nbNEBopp, nbNEBfri;
  Rzone NotPlain;
  myBool_t TwoEyes;
  eyeStatus_t EyeStatus;
  eyePosition_t EyePosition;
  uint MuellerStatus;
  uint EyeLibs, EyeSize, EyeShape;
  int VitalPoint;
  gainingLiberties_t gLPlay;

 public:

  Eye( IncrementalGoBoard *b = NULL );
  char        getColor() { return(Color); }
  Rzone       getEyeSpace() { return (EyeSpace); }
  Rzone&      getEyeFilledSpaceA() { return ( (Rzone&) EyeFilledSpace ); }
  Rzone       getEyeFilledSpace() { return (EyeFilledSpace); }
  int         getnbNEBopp() { return (nbNEBopp); }
  int         getnbNEBfri() { return (nbNEBfri); }
  eyeStatus_t getEyeStatus() { return (EyeStatus); }
  uint        getMuellerStatus() { return (MuellerStatus); }
  uint        getEyeLibs() { return (EyeLibs); }
  uint        getEyeSize() { return (EyeSize); }
  uint        getEyeShape() { return( EyeShape); }
  int         getVitalPoint() { return (VitalPoint); }
  eyePosition_t getEyePosition() { return (EyePosition); }
  myBool_t    getGLexist() { return ( gLPlay.exist ); }
  gainingLiberties_t getgLPlay() { return( gLPlay ); }
  myBool_t    getTwoEyes() {return TwoEyes;};
  void        setEyeStatus( eyeStatus_t s ) { EyeStatus = s; }
  void        setVitalPoint( int vp ) { VitalPoint = vp; }
  void        resetgLPlay() { gLPlay.exist = FALSE; }

  returnStatus_t init( String *s, String *sopp, Rzone &sharedLibs, int turn,
                       IncrementalGoBoard *b );
  returnStatus_t initmin( String *s, String *sopp, Rzone &sharedLibs, int turn,
                          IncrementalGoBoard *b );
  returnStatus_t initmincont( String *s, String *sopp, Rzone &sharedLibs, int turn,
                              IncrementalGoBoard *b );
  returnStatus_t searchEye2( String *s, String *sopp, Rzone &sharedLibs=voidRzone );
  void removeRecursively( int inter, Rzone &nonEyeSpace );
  Rzone findCandidateEyeSpace( String *s );
  returnStatus_t twoEyes();
  returnStatus_t addRecursively( int inter, Rzone &temp );

  void findEyePosition();
  returnStatus_t findEyeStatus();

  returnStatus_t nakade3();
  returnStatus_t nakade4();
  returnStatus_t nakade5();
  returnStatus_t nakade6();
  returnStatus_t nakade7();
  returnStatus_t nakade4_Corner( Rzone &, Rzone &, int a[4], int b[4] );
  returnStatus_t nakade5_Corner( Rzone &, Rzone &, Rzone &, Rzone &, int a[5], int b[5] );
  returnStatus_t nakade6_Corner( Rzone &, Rzone &, Rzone &, Rzone &, int a[6], int b[6] );
  returnStatus_t nakade7_Corner( Rzone &, Rzone &, Rzone &, Rzone &, int a[7], int b[7] );
  returnStatus_t nakade6_Side( Rzone &, Rzone &, Rzone &, Rzone &, int a[6], int b[6] );
  returnStatus_t nakade7_Side( Rzone &, Rzone &, Rzone &, Rzone &, int a[7], int b[7] );

  void debugEye();
  void debugGL()
};
```

Figura 4.49: La clase Eye

```
#define FALSE 0
#define TRUE (!FALSE)
#define END_OK 0
#define END_ERROR 1
#define reason_MAX 100
#define function_MAX 50

typedef struct{
  int             status;   /* 0 ok, 1 fatal error */
  char            reason[reason_MAX];  /* if status=1 reason contains the reason of the failure */
  char            function[function_MAX];/* if status=1 the error arrived in function,file,line. */
  char            file[function_MAX];
  int             line;
}returnStatus_t;

typedef int myBool_t;
typedef enum semiaiStatus_t { Winner, Looser, Seki, KoSemiai, NotKnown };
typedef enum toPlay_t { Pass2, Play }; /* PN: Pass2 to not to interfere with Pass in GoMove.h */

#define possible_reasons 9
enum { MORE_1_EB_PLAYER, MORE_1_EB_OTHER, N_PLAIN_LIBS_PLAYER, N_PLAIN_LIBS_OTHER,
 N_COMPLETELY_SURROUNDED, ADJACENT_IN_ATARI_PLAYER, ADJACENT_IN_ATARI_OTHER,
 SECOND_LINE_CORRIDOR_PLAYER, SECOND_LINE_CORRIDOR_OTHER };

const char reason[possible_reasons][reason_MAX]={"more than one EB for player",
 "more than one EB for other", "non plain liberties for player", "non plain liberties for other",
 "not completely surrounded","player has adjacent string(s) in atari",
 "other has adjacent string(s) in atari", "player has a second line corridor",
 "other has a second line corridor"};

typedef struct{
  myBool_t answer;      /* only if we are sure is class_e will be true */
  myBool_t reasons[possible_reasons];/* if not class e the positions for the reasons are true.*/
  myBool_t onlyOneNPLibertyPlayer;   /* if TRUE will be considered as class e */
  myBool_t onlyOneNPLibertyOther;    /* but alternative moves needed */
}isClassE_t;

class SemiaiStatus
{
  semiaiStatus_t statusIfPlay;
  semiaiStatus_t statusIfPass;
 public:
  SemiaiStatus()
    {
      statusIfPlay=statusIfPass=NotKnown;
    }
  void setStatusIfPlay( semiaiStatus_t status ){ statusIfPlay = status; }
  void setStatusIfPass( semiaiStatus_t status ){ statusIfPass = status; }
  void setStatus( semiaiStatus_t status ){ statusIfPlay = statusIfPass = status; }

  semiaiStatus_t getStatusIfPlay( ){ return(statusIfPlay); }
  semiaiStatus_t getStatusIfPass( ){ return(statusIfPass); }
};

/* Semiai Helpers class aims to be more generally useful.
   Functions of common interest to other classes */
class SemiaiHelpers2
{
  IncrementalGoBoard *board;
 public:
  SemiaiHelpers2 (IncrementalGoBoard * b) { board = b; }
  /* Converts a string to an Rzone */
  Rzone stringToRzone( String *s );
  /* How many strings in a given Rzone? */
  int nbStringsInRzone( Rzone &z );
  /*how many common elements?*/
  int intersectionCardinalRzone( Rzone &z1, Rzone &z2);
};
```

Figura 4.50: La clase SemeaiHelpers2

Capítulo 5

Resultados Experimentales

No supe comprender nada entonces. [...]
¡Las flores son tan contradictorias!
A. Saint-Exupéry "El Principito"

5.1 Go Text Protocol

GnuGo es hasta ahora, el único programa de Go de código abierto. Está desarrollado bajo el proyecto GNU y se distribuye según licencia GPL. *GnuGo* participa regularmente en las competiciones internacionales de Computer Go y tiene un ránking estable de 8 *kyu* en el servidor de Go por Internet NNGS. En noviembre de 2003 ganó con 10 victorias en 10 partidas la VIII Olimpiada de Computer Games.

Una de las herramientas que incorpora la versión 3.0 de *GnuGo* es Go Text Protocol (GTP). GTP aparece como alternativa al antiguo Go Modem Protocol y con el objetivo de ser un interfaz más potente y flexible para la comunicación máquina-máquina. GTP tiene dos aplicaciones principales; en primer lugar comunicar *GnuGo* con 'gnugoclient' para conectar el programa al servidor NNGS y por otro lado facilitar la ejecución de colecciones de problemas de test (tests de regresión).

Es en esta segunda aplicación en la que estamos interesados. El objetivo principal de los tests de regresión es controlar los errores del programa. Un test típico consiste en especificar una posición de tablero, obtener el movimiento generado por el programa y contrastarlo con la/s posible/s respuesta/s.

Por otro lado GTP permite especificar en los tests si esperamos o no que el programa responda correctamente. Así cuando un test que esperamos fallar es superado podemos detectar la mejora efectiva del programa. Por el contrario, cuando

un test que esperamos pasar falla sabemos que los últimos cambios introducidos han generado efectos laterales negativos en la estructura del mismo.

5.1.1 Sintaxis GTP

Las colecciones de test se almacenan en ficheros de extensión tst. Una parte de un fichero de test se muestra en la figura 5.1.

Las líneas que empiezan por #, o en general, cualquier carácter siguiendo a un símbolo # son interpretados como comentarios en modo GTP y por tanto ignorados por la máquina. Los comandos GTP se ejecutan en orden lineal pero sólo aquellos que aparecen en líneas numeradas se utilizan para la realización de test.

Las líneas empezadas por #? son mágicas para los tests de regresión e indican el resultado correcto del test junto con el estatus esperado (superado o no superado). La cadena de caracteres entre corchetes se compara como expresión regular con la respuesta del comando GTP numerado inmediatamente anterior.

Una característica útil de las expresiones regulares es que mediante el carácter `|` podemos especificar alternativas. Así en el test 90 `[B14|D17]` significa que tanto si el movimiento generado es `B14` como si es `D17` el test se marcará como superado. Además si la cadena de caracteres empieza con `!` estamos indicando que se aceptarán como resultados válidos todos aquellos que no figuren en la expresión regular. En el test 95 cualquier resultado excepto `P13` permitirá superar el test.

```
# Connecting with ko at B14 looks best. Cutting at D17 might be
# considered. B17 (game move) is inferior.
loadsgf games/strategy25.sgf 61
90 gg_genmove black
#? [B14|D17]

# The game move at P13 is a suicidal blunder.
loadsgf games/strategy25.sgf 249
95 gg_genmove black
#? [!P13]

loadsgf games/strategy26.sgf 257
100 gg_genmove black
#? [M16]*
```

Figura 5.1: Ejemplo de fi chero de test[1]

Por último podemos especificar nuestras expectativas sobre el comportamiento del programa. Por defecto los test se espera que sean superados. Si por algún motivo queremos indicar que esperamos que el test falle añadiremos un asterisco tras los corchetes como en el test 100.

[1] Ejemplo obtenido del manual de documentación de *GnuGo* [20]

Con la distribución de *GnuGo* vienen definidos una serie de comandos GTP, algunos de interés general y la mayoría especificos para la realización de tests de regresión para *GnuGo*. Para poder realizar los tests de regresión sobre el módulo *Semeai-01ES* tendremos que definir un comando GTP que se adecúe a nuestras necesidades (cf. sección 5.2.1). Por ejemplo, `loadsgf` es un comando GTP que se encarga de cargar el fichero sgf (cf. apéndice B) hasta una posición determinada (opcional) para ser tratado por el programa y `gg_genmove` pide el mejor movimiento a nivel global para el color especificado.

Con todo esto ya podemos interpretar el ejemplo de la figura 5.1: cargar el fichero `strategy25.sgf` hasta la posición 61 y obtener el mejor movimiento de negro. Si el movimiento generado por el programa es `B14` o `D17` el test es superado (como se esperaba). Cargar el mismo fichero hasta la posición 249 y obtener el mejor movimiento para negro. Si el programa responde `P13` test no superado, en otro caso test superado. Por último cargar el fichero `strategy26.sgf` hasta la posición 257 y obtener el mejor movimiento de negro. Si el programa responde M16 el test será inesperadamente superado.

5.1.2 Diagrama de ejecución

Una vez realizado el fichero de test podemos invocar al programa en modo GTP y redireccionar la entrada al fichero de test. La salida la redireccionamos a su vez a una pequeña utilidad escrita en `awk` que se encarga de comparar los resultados dados por el programa con los especificados en el fichero de test.

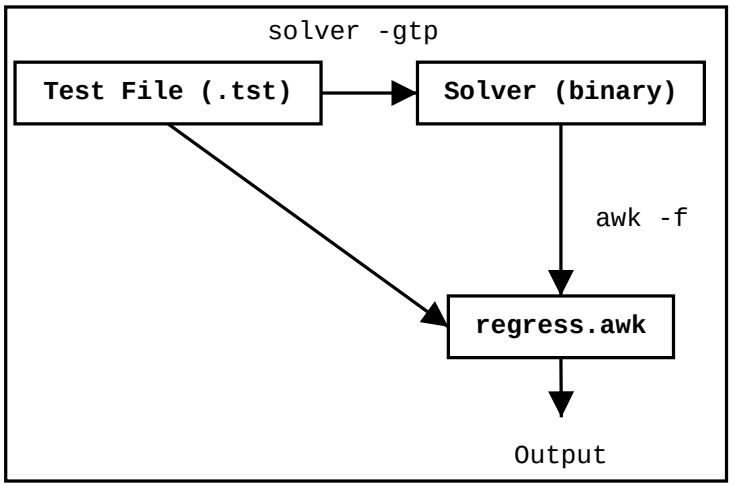

Figura 5.2: Diagrama de ejecución de una colección de tests de regresión

En la figura 5.2 podemos ver el diagrama de ejecución de una colección de

tests cualquiera. Se llama al programa a testear en modo gtp enviándole como entrada el fichero de test. La salida de este junto con el mismo fichero de test sirve como entrada a la ejecución del script `regress.awk` que compara los resultados obtenidos por el programa con los especificados en el fichero `fichero.tst`. Por último expulsa por la salida estándar los resultados del test con el formato que se muestra en la figura 5.3.

5.1.3 Presentación de los resultados

Tras el proceso de los datos obtendremos en la salida estándar el resultado de cada uno de los tests. Para cada test hay cuatro posibles resultados:

```
1 failed: Correct '!E5', got 'E5'
2 failed: Correct 'C9|H9', got 'F9'
3 PASSED
4 failed: Correct 'B5|C5|C4|D4|E4|E3|F3', got 'B7'
5 PASSED
6 failed: Correct 'D4', got 'E4'
7 PASSED
8 failed: Correct 'B4', got 'A3'
9 failed: Correct 'G8|G9|H8', got 'D9'
10 failed: Correct 'G9|F9|C7', got 'J9'
11 failed: Correct 'D4|E4|E5|F4|C6', got 'B3'
12 failed: Correct 'D4', got 'C6'
13 failed: Correct 'D4|E4|E5|F4', got 'C6'
```

Figura 5.3: Resultado de un test de regresión

- **passed** Test superado como se esperaba. Este es el resultado ideal.
- **PASSED** Test superado inesperadamente. Es el resultado deseado cuando estamos deseando corregir un error, un antiguo test que no se superaba ahora se supera.
- **failed** Test no superado, como esperábamos. Estos son los tests que habitualmente reflejan las debilidades de la máquina.
- **FAILED** Test no superado de manera inesperada. Este resultado suele aparecer cuando añadimos cambios al programa que sin darnos cuenta generan efectos laterales no deseados. También puede ser motivado por un test que antiguamente se pasaba gracias a varios errores mutuamente compensados, cuando uno de los errores se arregla dejan de estar compensados y los otros salen a la luz.

Un ejemplo de posible salida tras efectuarse un test de regresión puede verse en la figura 5.3.

5.2 Semeai Test Suite

En todo proyecto de desarrollo de software la fase de test constituye una etapa de máxima importancia. Cuando el proyecto en cuestión es un programa de Go la importancia de la comprobación del código es si cabe mayor. Se dice que las partidas de Go son como los copos de nieve; no hay dos iguales.

Al desarrollar un programa de Go existe un compromiso entre el grado de especificidad que damos a una determinada función y la cantidad de posibles situaciones de tablero que estamos contemplando. Cuanto más general intentamos que sea una función más casos posibles estamos contemplando y, por tanto, más fácil es descuidar alguna excepción. Esa excepción no contemplada es la que acaba haciendo perder a tu programa la partida decisiva en el momento más inesperado.

Durante el desarrollo de *Semeai-01ES* la fase de test ha sido continuada, cada nuevo submódulo añadido ha sido exhaustivamente comprobado con una colección de test específica. Sin embargo nos encontramos ante la paradoja de la autoevaluación: ¿cómo se cuestiona uno mismo lo que no sabe si no lo sabe? O, lo que es lo mismo, ¿cómo estar seguros de que pasar los tests garantiza la corrección del código diseñado?

Para maximizar la seguridad de que el código es correcto y operativo nos hemos basado en dos principios: (1) Exhaustividad. Cuanto mayor sea el número y más variados sean los tests mejor. (2) Asegurarnos de que, al menos, lo que el programa debe resolver correctamente así lo hace.

Para la realización de los tests se han utilizado las potencialidades de GTP descritas en la sección 5.1. Para ello hemos tenido que definir una nueva función gtp que se amoldara a nuestras necesidades: `solve_semeaiS` .

5.2.1 El comando gtp solve_semeaiS

El comando GTP `solve_semeaiS` recibe como parámetros dos intersecciones que determinan los dos grupos en *semeai*. Primero se encarga de comprobar que las intersecciones son válidas, es decir, que las intersecciones pertenecen al tablero, que no están vacías y que indican posiciones de colores opuestos.

A continuación asume que la primera intersección indica el color con el turno de juego y llama a la función `semeaiResult` que se encarga de llamar a la clase s e iniciar la búsqueda del estatus y el movimiento a jugar óptimo para dicha situación.

Por último devuelve el estatus y el movimiento de manera compatible con la sintaxis interna del protocolo GTP.

```
#####     Semeai test suite for class 0 semeai    #####

# * Function:  Decide Semeai Status and move to play.
# * Arguments: vertex for essential white/black block,
#              vertex for essential black/white block.
#              First color is assumed to play.
# * Fails:     invalid vertex, empty vertex,
#              vertices of same colors
# * Returns:   Semeai status (if play) followed by move to play.
#              0=Looser, 1=Winner, 2=Seki, 3=Unknown, 4=Ko

# gtp command: solve_semeaiS

loadsgf Sgf_test_files/_semeai_C0_001.sgf
1 solve_semeaiS A10 A11
#? [0 PASS]

2 solve_semeaiS A11 A10
#? [1 PASS]

loadsgf Sgf_test_files/_semeai_C0_002.sgf
3 solve_semeaiS L1 N1
#? [2 PASS]

4 solve_semeaiS N1 L1
#? [2 PASS]

loadsgf Sgf_test_files/_semeai_C0_003.sgf
5 solve_semeaiS H1 K1
#? [2 PASS]

6 solve_semeaiS K1 H1
#? [2 PASS]
```

Figura 5.4: Inicio del fi chero de test semeais_0.tst

5.2.2 STS-RV

Uno de nuestros objetivos al realizar la colección de tests es que pudieran servir a otros programadores. Disponer de una amplia y completa colección de tests para cada uno de los posibles objetivos tácticos del juego de Go (*tsumego*, *semeai*, *yose*, conexión, captura,...) es algo muy positivo. En primer lugar permite ahorrar la tediosa tarea de realizar los tests, en segundo lugar resuelve la paradoja de la autoevaluación puesto que es con los tests de otra persona con los que se pone a prueba el propio programa y, por último, si se convierten en una referencia utilizada por la mayoría de la comunidad, sirven como patrón de comparación entre las capacidades de los distintos programas[2].

[2]La colección de tests de *tsumego* de T. Wolf es un ejemplo

Por todo ello hemos decidido poner orden en el cajón de sastre de nuestros tests, documentar el funcionamiento y la estructura de la colección y distribuirla según licencia GPL. Hemos bautizado la colección como `STS-RV 1.0` y está disponible en Internet[3] desde abril de 2004.

Hasta el momento nos consta que el grupo de desarrollo de *GnuGo* ha puesto a prueba la versión 3.5.5 de la máquina con la `STS-RV` (cf. sección 5.3.2).

Estructura de la colección

La colección STS-RV se estructura en 6 ficheros de test que suman un total de 722 tests de regresión repartidos en 130 ficheros sgf (cf. apéndice B). Cada test se realiza por partida doble siendo blanco o negro el color a jugar. En muchas ocasiones ambos tests son interesantes puesto que el resultado final depende del turno de juego y el movimiento ganador de blanco y de negro no tienen porqué coincidir. Sin embargo hay ocasiones también en que siendo uno de los dos tests altamente significativo el test inverso es trivial y no tiene mayor interés (por ejemplo en los tests que implican un *tesuji*). Aún así hemos decidido mantener en todos los tests el desdoblamiento por una cuestión de homogenenidad.

En la tabla 5.1 presentamos de manera esquemática la composición de la colección. Hemos eliminado de la versión final los tests que hemos considerado demasiado específicos a nuestro proceso de desarrollo. Por ejemplo los tests de la *clase p* en que nos preguntamos si un *semeai* dado conforma a la especificaciones de la *clase e* difícilmente serán de utilidad para quien haya seguido un proceso de desarrollo distinto al nuestro.

FICHERO TEST	CONTENIDO	FICHEROS SGF	N. F. SGF	N. TESTS
semeais_0.tst	S. Clase 0	_semeai_C0_00i.sgf	13	26
semeais_1.tst	S. Clase 1 y 2	_semeai_C1_00i.sgf	27	208
semeais_e.tst	S. Clase e[4]	_semeai_Ce_00i.sgf	33	252
semeais_RH.tst	RH Problems[5]	pagi_PjPk.sgf	27	94
semeais_GSAT.tst	*Tesuji*[6]	_semeai_GSAT_00i.sgf	11	88
semeais_Misc.tst	S. Clase > e	_semeai_C9_00i.sgf	19	54
TOTAL			130	722

Tabla 5.1: Estructura de la STS-RV

[3] `http://www.ai.univ-paris8.fr/~ritx`

[4] Comprueban la validez de la implementación de la clase Eye

[5] Problemas obtenidos de [22]

[6] Problemas específi cos de *semeai* donde hay algún *tesuji* implicado obtenidos de [11]

5.3 *Semeai-01ES* vs STS-RV

Tras describir la estructura interna del módulo y las características de la STS-RV procedemos a detallar los resultados obtenidos por *Semeai-01ES* . En la tabla 5.2 se muestra el porcentaje de tests superados para cada uno de los seis ficheros de test que conforman la suite. Los presentes resultados se han obtenido con un límite de 10000 nodos por test y todas las heurísticas descritas en la sección 4.3.6 activadas.

FICHERO TEST	N. TESTS	N. T. SUPERADOS	PORCENTAJE
semeais_0.tst	26	26	100.0
semeais_1.tst	208	208	100.0
semeais_e.tst	252	243	96.4
semeais_RH.tst	94	41	43.6
semeais_GSAT.tst	88	76	86.4
semeais_Misc.tst	54	39	72.2
TOTAL	722	633	87.7

Tabla 5.2: Resultados de *Semeai-01ES* frente a la STS-RV

Es necesaria cierta cautela a la hora de valorar estos resultados. Sería tan poco correcto desatar una euforia descontrolada por la casi perfección obtenida en los tres primeros conjuntos de test como proclamar, de manera derrotista, que no se alcanza ni tan siquiera el 50 % de tests superados por los resultados obtenidos para el conjunto de tests semeais_RH.

Los tests para las clases 0, 1 y e son tests adaptados a la estructura de *Semeai-01ES* . Su función principal era comprobar que el módulo realizaba correctamente todo aquello que se le suponía capaz de hacer. Durante el desarrollo de la función de evaluación estos tests fueron comprobados una y otra vez hasta conseguir una función de evaluación prácticamente libre de error. Por ello, es relativamente normal haber obtenido tan excelentes resultados. Cabe añadir que la mayoría de estos tests se resuelven de manera estática a nivel del nodo raíz.

Por otro lado, la dificultad de los tests de Richard Hunter, podría calificarse de extrema. Dificultad para un humano o para una máquina son conceptos entre los que no existe una correlación clara. La gran mayoría de los tests de Hunter, independientemente de la dificultad que tengan para un humano, son altamente complicados para una máquina por dos motivos: i) el tamaño del conjunto de posibles movimientos en el nodo raíz ii) la longitud de la secuencia ganadora. Este cóctel garantiza un árbol de búsqueda de dimensiones desmesuradas. Los tests de Hunter se han incluido en la STS-RV más como muestra de la enorme dificultad que puede encerrar el concepto de *semeai* y como punto de referencia al que sería

deseable llegar en un futuro, que como conjunto de tests que es imperativo superar para poder afirmar que se tiene una máquina válida para decidir un *semeai*.

Los dos últimos conjuntos de test son altamente significativos. `semeais_GSAT` es un conjunto de tests con un nivel de dificultad adecuado; el espacio de movimientos posibles no es excesivamente grande, la secuencia de resolución en el peor de los casos alcanza los 12 movimientos y todos ellos requieren la detección de un *tesuji* para ser resueltos. Además es un conjunto de tests totalmente independiente de la estructura de *Semeai-01ES* y por todo ello los resultados obtenidos tienen un alto valor descriptivo de las capacidades del módulo.

`semeais_Misc` a pesar de haber sido desarrollado a base de composiciones propias, y en algún caso de situaciones aparecidas en mis propias partidas, tiene como objetivo detectar las carencias del módulo y no señalar sus bondades. Además todos los tests son de *semeais* de clase superior a e y por tanto ponen a prueba tanto la función de evaluación como la corrección del algoritmo de búsqueda. Por ello los resultados obtenidos para este conjunto de tests son también altamente significativos.

En el apéndice E incluimos información detallada de cada test: número de nodos explorados, profundidad a la que se detecta la solución, utilización de las heurísticas, ...

Con todo esto podemos concluir que los resultados mostrados en la tabla 5.2 ponen de manifiesto el alto nivel conseguido por *Semeai-01ES* en la resolución de situaciones de *semeai*.

Tipos de error

Al realizar tests de regresión es tan importante la información que aportan los tests superados como la que se puede extraer de los tests errados. De hecho, no todos los errores son iguales, ni nos indican lo mismo acerca de nuestro programa. Por ello presentamos a continuación una clasificación de los distintos tipos de errores y describimos detalladamente a qué tipo de error pertenecen los tests no superados por *Semeai-01ES* .

1. **Error muy grave (MG)** Indican un error, estructural o localizado, en el código del programa. Catalogaremos como error MG a aquellos tests que indiquen o bien un estatus diferente o bien un movimiento a jugar diferente del apropiado.

2. **Error moderado (M)** Indican una carencia del programa para acometer un problema con un amplio árbol de búsqueda. Catalogaremos como error M aquellos test que devuelvan resultado `Unknown` tras haber agotado los 10000 nodos disponibles. Un error M indica una necesidad de mayor selectividad

a la hora de escoger los movimientos posibles y a la hora de cortar las ramas del árbol que no contienen la solución al problema.

3. **Error leve (L)** Un error leve se produce cuando el programa detecta correctamente el estatus pero no se da cuenta de que es posible pasar para conseguirlo y juega un movimiento innecesario. Es leve puesto que el *semeai* se resuelve correctamente y el único perjuicio es perder la *sente*. Los errores leves que comete *Semeai-01ES* se deben al hecho de que no se genera en el nodo raíz un movimiento pasar como parte del conjunto de posibles movimientos.

TST	MG	T	M	T	L	T	**T**
0		0		0		0	0
1		0		0		0	0
e	85, 94, 96, 98, 106, 110, 117, 156, 183	9		0		0	9
RH	27, 38, 45, 47, 51, 61-63, 75, 93-94	11	13-15, 17-18, 24, 29, 33, 35-37, 39-40, 49-50, 53-54, 57-58, 60, 65-68, 71, 73, 74, 77-84, 86, 89, 91-92	39	16, 55, 64	3	53
GSAT	63, 64, 67, 88	4	9, 10, 25, 59, 71, 81, 83, 87	8		0	12
Misc	1-4, 6, 35, 37	7	33, 40, 42, 44, 45, 52-54	8		0	15
TOTAL		31		55		3	89

Tabla 5.3: Clasificación de los errores de *Semeai-01ES* frente a la STS-RV

En la tabla 5.3 aparecen detallados todos los tests no superados por el módulo *Semeai-01ES* agrupados según la clasificación de errores propuesta. El grueso de los tests no superados corresponde a errores moderados y leves y tan sólo 31 (un 4.2 % de toda la suite) son errores muy graves.

Estos resultados sugieren dos caminos a seguir. En primer lugar continuar mejorando la estructura del módulo para reducir ese pequeño porcentaje de errores muy graves y, en segundo lugar, esos 55 errores moderados indican que aún queda espacio para seguir perfeccionando la calidad del módulo y llaman a mejorar la selectividad del algoritmo de búsqueda.

5.3.1 Variación de los resultados respecto a las heurísticas y al número de nodos explorados

Estudiar la variabilidad de los resultados obtenidos por *Semeai-01ES* respecto al número de nodos explorados es un experimento interesante para obtener conclusiones sobre las características del módulo, tanto acerca de la calidad de la función de evaluación como del algoritmo de búsqueda. Por otro lado para determinar la validez y calidad de las heurísticas utilizadas en el algoritmo de búsqueda estudiar el efecto de su desactivación constituye otro experimento crucial.

FICHERO TEST	N. TESTS	N. T. SUPERADOS	PORCENTAJE
semeais_0.tst	26	26	100.0
semeais_1.tst	208	208	100.0
semeais_e.tst	252	243	96.4
semeais_RH.tst	94	29	30.8
semeais_GSAT.tst	88	65	73.9
semeais_Misc.tst	54	36	66.7
TOTAL	722	607	84.1

Tabla 5.4: Resultados de *Semeai-01ES* explorando 1000 nodos

Número de nodos. Hemos realizado dos ejecuciones de toda la suite con 1000 y 100000 nodos respectivamente. En ambas hemos mantenido todas las heurísticas activadas igual que en la ejecución mostrada en la tabla 5.2.

En primer lugar observamos que no hay variación alguna en los resultados de los tests 0, 1 y e. Este resultado era previsible puesto que, como ya habíamos explicado, la mayoría de estos tests se resuelven de manera estática.

FICHERO TEST	N. TESTS	N. T. SUPERADOS	PORCENTAJE
semeais_0.tst	26	26	100.0
semeais_1.tst	208	208	100.0
semeais_e.tst	252	243	96.4
semeais_RH.tst	94	45	47.9
semeais_GSAT.tst	88	80	90.9
semeais_Misc.tst	54	42	77.8
TOTAL	722	644	89.2

Tabla 5.5: Resultados de *Semeai-01ES* explorando 100000 nodos

Por otro lado, para los tests RH, GSAT y Misc, observamos una clara correlación entre el aumento del número de nodos explorados y el aumento del número de

tests superados. Una función de evaluación teóricamente perfecta cumple la condición de que, a mayor profundidad de exploración mayor, probabilidad de dar con el movimiento correcto. Una función de evaluación poco rigurosa, con parches y heurísticas abusivas, tiene un comportamiento impredecible y, cuanto más se utiliza, más se manifiestan sus errores. La correlación observada demuestra la robustez del diseño realizado para la función de evaluación del módulo *Semeai-01ES* .

Heurísticas. La realización de la colección de tests STS-RV con todas las heurísticas desactivadas y explorando 10000 nodos ha producido los resultados mostrados en la tabla 5.6.

Observamos que el número de tests superados es similar al obtenido con la exploración de 1000 nodos y las heurísticas activadas. Sólo este dato prueba la validez de las heurísticas utilizadas así como la mejora que producen en el comportamiento del módulo.

Sin embargo, cabe notar además que, en la mayoría de los tests superados, la respuesta correcta se obtiene a mayor profundidad que con las heurísticas activadas. Sin necesidad de entrar en una comparativa exhaustiva de las profundidades de búsqueda alcanzadas en uno y otro caso, podemos contrastar los tiempos utilizados en cada caso. Mientras que en la ejecución con 10000 nodos y heurísticas activadas se utilizaron un total de 1005s, en la ejecución sin heurísticas se consumieron 1375s.

Por estas razones podemos concluir que el diseño de las heurísticas utilizadas en *Semeai-01ES* es altamente satisfactorio.

FICHERO TEST	N. TESTS	N. T. SUPERADOS	PORCENTAJE
semeais_0.tst	26	26	100.0
semeais_1.tst	208	208	100.0
semeais_e.tst	252	245	97.2
semeais_RH.tst	94	35	37.2
semeais_GSAT.tst	88	66	75.0
semeais_Misc.tst	54	36	66.6
TOTAL	722	616	85.3

Tabla 5.6: Resultados de *Semeai-01ES* sin heurísticas activadas.

5.3.2 *Semeai-01ES* vs GnuGo 3.5.5

Para poder valorar de manera adecuada los resultados obtenidos por el módulo *Semeai-01ES* es necesaria una referencia con la que poder comparar. *GnuGo* es

el presente campeón de la *Computer Olympiad* y por tanto uno de los mejores programas en la actualidad. Hemos decidido probar la `STS-RV` con la última versión en desarrollo de *GnuGo*, la 3.5.5. *GnuGo* utiliza código específico para la resolución de *semeai* desde antes de la versión 2.0. A partir de enero de 2003 se produjo un cambio sustancial al respecto, la versión 3.3.14 reemplazó el antiguo módulo de análisis estático por un módulo que realizaba búsqueda arborescente para el subproblema de *semeai*. Desde entonces hasta la presente versión se han producido mejoras de manera incremental [18]. Para poner *GnuGo* a prueba hemos contado con la inestimable ayuda de Gunnar Farnebäck, uno de los principales desarrolladores de *GnuGo* que ha adaptado el comando `solve_semeaiS` a las características del programa y ha ejecutado todos los tests de regresión.

FICHERO TEST	N. TESTS	N. T. SUPERADOS	PORCENTAJE
semeais_0.tst	26	24	92.3
semeais_1.tst	208	123	59.1
semeais_e.tst	252	213	84.5
semeais_RH.tst	94	43	45.7
semeais_GSAT.tst	88	79	89.8
semeais_Misc.tst	54	29	53.7
TOTAL	722	511	70.8

Tabla 5.7: Resultados de GnuGo 3.5.5 frente a la STS-RV

En la tabla 5.7 presentamos los resultados de *GnuGo* 3.5.5. Hay un dato sorprendente en la tabla de resultados de *GnuGo* y es que consigue un 84.5 % de aciertos en la serie e cuando teóricamente son más complicados que los tests de la serie 1 donde sólo obtiene un 59.1 %. Esto se debe a que la mayoría de los tests de la suite e consisten en un grupo que tiene dos ojos y ya está vivo, contra otro que tiene un gran ojo; la cuestión es saber si la carrera acabará en victoria del grupo vivo o finalizará como *seki* en el sentido amplio.

En estas situaciones *GnuGo* detecta la vida incondicional de uno de los dos grupos y no llama a su módulo de *semeai* sino a su módulo de vida y muerte [18]. Esto sucede en 210 de los 252 tests de la serie (7-84, 87-96, 99-114, 117-164, 191, 192, 197-252). El módulo de vida y muerte de *GnuGo*, por ser una de las partes más importantes de un programa de Go, está muy evolucionado y esa es la razón por la que obtiene tan buenos resultados en la serie e.

Al comparar los resultados de *GnuGo* con la tabla 5.2 podemos inferir dos conclusiones. En los tests adaptados a la estructura de *Semeai-01ES* (0,1 y e), éste es claramente superior a *GnuGo* y en segundo lugar en los tests independientes (RH, GSAT y Misc) los dos programas alcanzan un nivel de juego similar. Por tanto, podemos afirmar, que el módulo *Semeai-01ES* ha alcanzado un nivel de

dominio de los *semeais* ligeramente superior a uno de los más potentes programas de Go de la actualidad.

5.4 Incorporación de *Semeai-01ES* a Golois

En nuestro planteamiento inicial, la incorporación de *Semeai-01ES* a *Golois*, se perfilaba como uno de los objetivos principales. En primer lugar porque someter al módulo a una situación de juego real, nos permitiría ver cómo influyen los posibles desajustes en la detección del *semeai* provocados por *Golois* en el funcionamiento efectivo del módulo. En segundo lugar porque podríamos someter al módulo a tests de tablero completo pudiendo así valorar la evaluación del valor combinatorio de cada lucha que realiza y que en los tests individuales queda obviada. Y por último por el placer y la satisfacción personal de verle actuar en una situación de juego real.

Por desgracia, un proyecto de tan sólo cuatro meses (en la teoría puesto que acabaron siendo seis más el tiempo de escribir esta memoria) no permite hacer todo lo que uno quisiera en un inicio. Durante el desarrollo del mismo han surgido agradables imprevistos, como la posibilidad de escribir un artículo, que han consumido gran parte del tiempo disponible. Añadiendo a todo esto ciertas imcompatibilidades de calendario acabaron obligándonos a descartar por el momento la integración con el programa global.

Sin embargo no considero que esto implique un fracaso. Los resultados presentados en la sección 5.3 demuestran haber alcanzado un nivel de juego respetable en un aspecto táctico del juego de Go de elevada dificultad. Si todos nuestros objetivos iniciales se hubieran cumplido, qué objetivos tan pobres hubieran sido.

Así que, por el momento, *Semeai-01ES* queda como parte de TAIL a la espera de ser integrado en *Golois* en un futuro próximo en que confluyan las condiciones necesarias para poder llevar a cabo la tarea.

Capítulo 6

Trabajo Futuro

Vértigo que el mundo pare,
que corto se me hace el viaje.
I. Serrano, "Atrapados en Azul"

La existencia de esta sección necesariamente responde a una de estas dos posibles razones: (a) no he acabado mi trabajo, (b) mi trabajo no está acabado. La razón (b), en su sentido coloquial, quiere decir que no es una línea de investigación muerta, que hay nuevas ideas a poner en práctica, que se puede mejorar y perfeccionar, en definitiva que, a medida que explorábamos los caminos que nos habíamos propuesto recorrer, nuevos caminos aparecían.

Seguramente la verdadera razón de ser es una combinación ponderada de (a) y (b). Ejercicio para el lector: determinar qué porcentaje corresponde a cada una de ellas.

A continuación, presentamos algunos de estos caminos:

- **Integración en *Golois*** Como ya hemos explicado en la sección 5.4 la integración de *Semeai-01ES* en *Golois* no se ha completado. Queda por tanto como una tarea a realizar en un futuro próximo.

- **Incrementar el número de tests superados** Gracias a la base de tests que hemos desarrollado, el perfeccionamiento del módulo se puede monitorizar en base al número de tests superados. Es por ello que incrementar este número se perfila como un camino interesante. Para ello las líneas a seguir por orden de importancia son:

    1. *Perfeccionamiento de la Clase p* La mayoría de los tests no superados presentan el mismo problema: la detección precipitada o tardía del momento en que se debe llamar a la función de evaluación. El

perfeccionamiento de la *clase p*, añadiendo nuevos parámetros que estrechen la definición de qué es y qué no es un *semeai* de *clase e*, puede incrementar considerablemente el rendimiento del módulo.

2. *Definición de nuevas heurísticas* Hemos visto en la sección 5.3 cómo la incorporación de heurísticas mejora los resultados del módulo *Semeai-01ES* de manera importante. Detectar nuevas heurísticas a incluir puede continuar esta mejora. Todas las heurísticas hasta ahora incorporadas son «heurísticas en positivo», esto es heurísticas que permiten detectar con antelación la victoria. Quizás un análisis pormenorizado de los árboles de búsqueda de cada test realizado nos permiten vislumbrar «heurísticas en negativo», es decir, heurísticas que nos permitan decidir que una determinada rama del árbol no contiene el movimiento correcto y por tanto puede ser desestimada.
3. *Finalizar la implementación de la clase Eye* La implementación de las clases de equivalencia, según la *neighbour classification*, que todavía no se han escrito puede reducir en algunos casos la profundidad de búsqueda necesaria para hallar la solución.
4. *Mejorar la gestión del* ko Aunque la *clase Eye* se ha implementado teniendo en cuenta las posibles situaciones de *ko* que puedan aparecer, a nivel global del módulo en estos momentos el *ko* se está tratando como un estatus *Unknown* por sencillez y para evitar posibles errores. Una gestión atrevida del *ko* puede mejorar los resultados en aquellos tests que impliquen dicha situación en su árbol de búsqueda.

- ***Semeai-01ES* con GTS** Las características del subobjetivo del Go *semeai* le convierten en un buen candidato a ser tratado con el algoritmo GTS [13]. Comparar las prestaciones del módulo, con su actual algoritmo de búsqueda Alpha-Beta, con otra versión del mismo implementando un algoritmo de búsqueda GTS se perfila como una interesante línea de investigación que seguramente resultará en un mayor número de tests resueltos y un menor número de nodos explorados medio en el global de la colección de test.

- **Traducción al inglés** La lengua dominante en cualquier área de investigación es, por desgracia o por fortuna, el inglés. Aunque en un principio había decidido escribir la memoria en inglés, la falta de tiempo y la preferencia del Dr. Cazenave por el castellano frente al catalán me han llevado a escribirla en castellano. Esto impone un serio handicap al acceso por parte de otros miembros de la comunidad no hispano hablantes a su contenido. Es por ello que esperamos encontrar en un futuro el tiempo necesario para preparar su traducción al inglés.

Después de esta fugaz incursión en el mundo del Computer Go he podido comprender muchos de los aspectos atractivos de esta disciplina. La gran dificultad inherente del juego, la necesidad de una alta capacidad de análisis, la multitud de situaciones en que un enfoque matemático puede dar interesantes frutos, la ineficacia de los métodos estándar que se utilizan en Computer Games, y, sobre todo, la emoción de la competición entre programas procedentes de todos los rincones del mundo hacen de esta disciplina un campo de investigación muy tentador.

Por ello, el verdadero proyecto de futuro que me gustaría llevar a cabo es el desarrollo de mi propio programa de Go. Si este proyecto se llevará a cabo el año que viene o después de mi jubilación depende de demasiados factores que escapan a mi control. El tiempo dirá.

Capítulo 7

Conclusión

> Es una reflexión penosa para un hombre considerar
> lo que ha hecho, comparado con lo que debió hacer.
> Sam Johnson

Llegados a este punto es momento de hacer balance. A lo largo (muy largo) de esta memoria hemos presentado los resultados de seis meses de investigación en el laboratorio de Inteligencia Artificial de la Université Paris VIII.

En primer lugar me gustaría aclarar las dudas que el tema de este trabajo pueda suscitar como proyecto de fin de carrera de Ingeniería de Telecomunicaciones. Es innegable que hubiera sido mucho más «normal» escoger un tema relacionado con el tratamiento de la señal o la codificación de la información. También es cierto que hay muy poco o nada de comunicación a distancia en este trabajo. Sin embargo, y teniendo presente que nos dirigimos a un mundo cada vez más interdisciplinar, la frontera entre las telecomunicaciones, la informática y las matemáticas se vuelve más y más difusa. Por ello considero, que la palabra importante en el binomio “Ingeniería de Telecomunicaciones” es, sin lugar a dudas, la primera. Ingeniería no es más que utilizar el ingenio para resolver problemas que todavía no tienen solución. En este trabajo hemos atacado un problema abierto de elevada dificultad y hemos propuesto una solución original que proporciona unos resultados de alto nivel. Por ello considero que, a pesar de lo poco ortodoxo de la materia tratada, esta memoria constituye un proyecto de fin de carrera de Ingeniería perfectamente válido.

En el capítulo 3 definíamos nuestro objetivo inicial como: “desarrollar un módulo autónomo encargado de la resolución táctica de *semeai* para su posterior inclusión en la arquitectura de *Golois*”. A medida que el proyecto avanzaba se perfilaron dos nuevos subobjetivos: la creación de una base de tests de *semeai* y

el desarrollo de una novedosa teoría de clasificación de formas de ojo.

STS-RV — Hemos diseñado una completa colección de tests de *semeai*, distribuida bajo licencia GPL para el beneficio de la comunidad de Computer Go. La STS-RV (cf. sección 5.2) contiene 722 tests agrupados en 6 ficheros que permiten comprobar de manera exhaustiva las capacidades de cualquier módulo de *semeai*. La colección está disponible en Internet y esperamos que sea utilizada por el mayor número posible de programadores.

Neighbour Classification — En la sección 4.2.1 hemos presentado una teoría completamente nueva sobre cómo clasificar de manera eficiente formas de ojo y hemos definido la *life property* que permite decidir la vida de ciertos grupos con gran eficiencia computacional. Estas ideas [35] fueron presentadas en noviembre de 2003 en la «10th Advances in Computer Games Conference», una de las conferencias más importantes a nivel internacional sobre Computer Games.

Semeai-01ES — El objetivo inicial y principal de este proyecto es el desarrollo de un módulo de resolución de *semeai*. La consecución del mismo es el módulo *Semeai-01ES* y ha sido ampliamente descrito en el capítulo 4. Los resultados obtenidos (cf. capítulo 5) muestran el alto nivel alcanzado y nos permiten afirmar haber obtenido una calidad equiparable a uno de los mejores programas de la actualidad.

La consecución de estos tres objetivos nos permite valorar de manera muy positiva el trabajo realizado durante los seis meses de estancia en el laboratorio de Inteligencia Artificial de la Université Paris VIII.

No quisiera dejar de mencionar la importancia que también ha tenido el aspecto personal de esta experiencia. Son muchos los motivos que la han hecho tan valiosa.

En primer lugar la posibilidad de realizar este proyecto en el extranjero me ha permitido descubrir una nueva cultura con su lengua, sus gentes y sus incontables matices.

Después de cinco años de universidad acostumbrado al aprendizaje guiado, a las preguntas concretas, a resolver exámenes y a saber exactamente que es lo que hay que hacer para llegar al curso siguiente; embarcarme en este proyecto ha supuesto una ruptura radical con esta dinámica. En el mundo real de la investigación no hay guías ni preguntas concretas sino dudas y un enjambre dispar de caminos posibles a seguir (aún recuerdo con aflicción la inseguridad y el miedo de la primera semana). Tomar contacto con la realidad del entorno de un laboratorio de investigación ha sido uno de los aspectos más postitivos de esta experiencia.

Por último, el hecho de enfrentarme a un proyecto de investigación completamente nuevo para mí, para el que no tenía una formación específica previa me ha

obligado a realizar un esfuerzo adicional para poder cubrir las carencias de base con las que me encontré en un principio. Por ello, llegar hasta aquí, hasta la última página de esta memoria, supone una enorme satisfacción.

A modo de deseo final, espero tener la oportunidad en un futuro próximo de llevar a cabo las posibles mejoras detalladas en el capítulo 6 y poder así ver en una competición real al módulo *Semeai-01ES* como parte integrada de un programa de Go.

Parte III

Apéndices

Apéndice A

El Go

El presente apéndice va dirigido a aquellos lectores que desconocen el juego del Go y sus reglas. La información aquí presentada, aunque no convertirá al lector en un experto jugador, es suficiente para permitir la comprensión de los aspectos relacionados con el juego que aparecen en los capítulos 2, 3, 4 y 5.

A.1 Pequeña historia del Go

El Go es el juego más antiguo que se juega hoy en día igual que en sus orígenes. Apareció hace unos 4000 años en China por donde se expandió con rapidez. A lo largo de la historia la consideración que del Go se tenía sufrió grandes variaciones. En la antigüedad se le consideraba uno de los Cuatro Artes junto con la música, la pintura y la poesía, Mao Zedong requirió a sus generales el estudio del juego mientras que la Revolución Cultural lo condenó como un mero «pasatiempo de burgueses».

El Go se introdujo en Corea durante la ocupación china hacia el año 109 A.C. aunque se desarrolló con lentitud.

En Japón fueron unos monjes budistas a su regreso de un viaje a China quienes introdujeron el juego hace unos 1400 años. El juego experimentó un rápido crecimiento y se estudió con avidez. Apreciado hasta en círculos imperiales el juego alcanzó una condición ceremonial. Aparecieron las llamadas «cuatro escuelas de Go» donde se empezaron a desarrollar las técnicas que condujeron al juego a la modernidad.

Los jugadores más relevantes de la historia de Japón son Dosaku (1645-1702), Honinbo Shusaku (1829-1862) a quien se conocía como «El Invencible» (se dice que nunca perdió con negras), y Go Seigen (1914).

En Europa el Go fue durante siglos un desconocido. Matteo Ricci, un misionero jesuita, cita por primera vez en un texto europeo el juego del Go hacia el

año 1600 pero no es hasta 1880 cuando el alemán Otto Korschelt escribe un libro completo sobre el juego. A partir de aquí se empezó a jugar en Alemania pero su expansión europea fue lenta. En 1958 se celebra el primer campeonato de Europa de Go.

A finales del siglo XIX los emigrantes chinos y japoneses introducen el juego en los Estados Unidos y Canadá y en 1934 se funda la *American Go Association*.

En la actualidad hay cinco ligas profesionales: dos en Japón (Nihon Kiin y Kansai Kiin), una en China, una en Corea y una en Estados Unidos donde juegan emigrantes de las otras cuatro ligas.

Los periódicos asiáticos publican a diario columnas sobre el Go, hay un canal de televisión dedicado 24 horas al Go y los torneos profesionales tienen bolsas equiparables a las de los torneos de golf en Occidente.

Corea es el país más potente en la actualidad y dónde el juego es más popular, se dice que uno de cada tres coreanos sabe jugar a Go.

En Internet hay cada vez más servidores de Go donde jugadores de todo el mundo se encuentran a cualquier hora del día.

A.2 Reglas del juego

Hay distintos conjuntos de reglas que definen el juego del Go. A continuación explicaremos las reglas japonesas y en la sección A.2.7 comentaremos otros sistemas y sus diferencias.

El Go es un juego de información perfecta para dos jugadores (blanco y negro). Se juega en un tablero de 19 por 19 líneas que dan lugar a 361 intersecciones. Un movimiento consiste en colocar una piedra en una de las intersecciones que aún queden vacías o bien pasar. Alternativamente blanco y negro realizan un movimiento hasta que el juego acaba. Negro mueve primero.

A.2.1 Grupos y libertades

Un grupo es un conjunto maximal no vacío de piedras adyacentes del mismo color conectado por líneas del tablero. Las libertades de un grupo son las intersecciones vacías adyacentes conectadas por una línea del tablero.

En la figura A.1 podemos ver cómo una piedra aislada tiene dos, tres o cuatro libertades dependiendo de si se encuentra en la esquina, el borde o el centro respectivamente. También vemos cómo las piedras enemigas contribuyen a reducir de tres a uno las libertades de la piedra negra a la derecha.

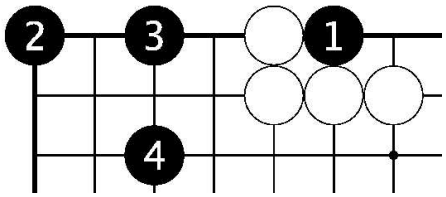

Figura A.1: Distinto número de libertades para las piedras negras dependiendo de su posición y de las piedras enemigas

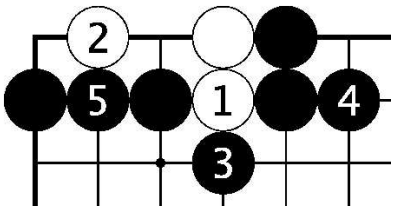

Figura A.2: Dos grupos blancos y tres negros con sus respectivas libertades. Los números indican las libertades de cada grupo.

En la figura A.2 hay dos grupos blancos y tres grupos negros. Nótese cómo la piedra marcada con 3 forma un grupo de una piedra aislado de los otros dos grupos negros por no ser adyacente.

A.2.2 Captura

Las piedras colocadas a lo largo de una partida no pueden moverse pero sí pueden ser capturadas. Cuando un grupo, de una o más piedras, es privado de todas sus libertades el grupo es capturado y se retira del tablero.

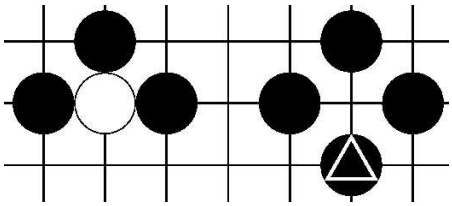

Figura A.3: Antes y después de la captura de una piedra

En la figura A.3 se observa como si negro juega en la última libertad de la piedra blanca esta es capturada y retirada del tablero.

En la figura A.5 tenemos un ejemplo de la captura de un grupo de cinco piedras negras. Si es el turno de blanco éste puede capturar una piedra negra jugando en 1 (figura A.4).

Si una piedra al ser colocada y después de retirar las posibles capturas queda sin libertades se trata de un «suicidio» y es un movimiento ilegal.

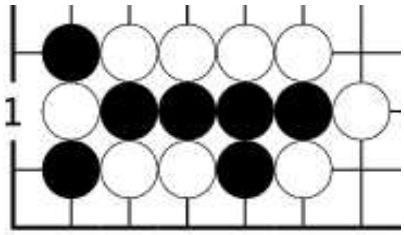

Figura A.4: Las cinco piedras negras están en *atari*.

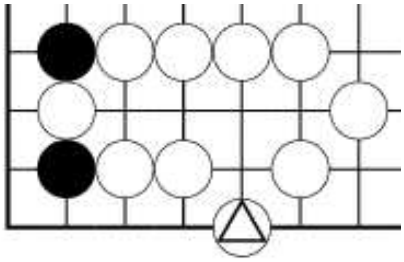

Figura A.5: Blanco en △ captura las cinco piedras negras.

A.2.3 Grupos inmortales y *seki*

Un ojo es una intersección vacía completamente rodeada por un grupo. Un grupo con un sólo ojo puede ser capturado por el adversario jugando primero todas las libertades exteriores y por último jugando en el ojo. Esta última jugada priva al grupo de su última libertad y por tanto queda capturado. No se puede jugar en el ojo hasta el final debido a la regla del suicidio.

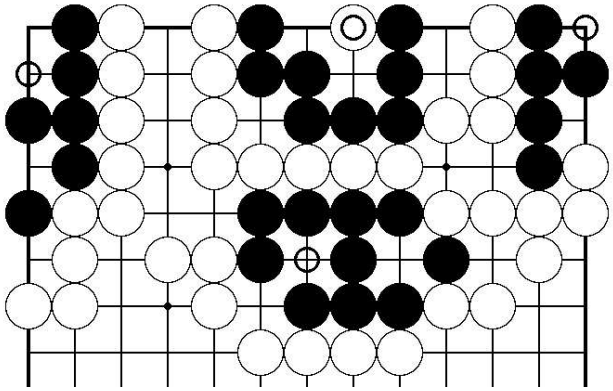

Figura A.6: Grupos negros con un ojo (señalados con un círculo)

Un ojo falso ocurre cuando alguna de sus las intersecciones adyacentes está controlada por un grupo enemigo. En la figura A.6 el grupo negro de la derecha y el de abajo tienen ambos un ojo falso.

Un gran ojo, objeto principal de discusión de la sección 4.2.1, es un ojo formado por más de una intersección. Dependiendo de su tamaño, su forma y las piedras amigas y enemigas en su interior dará lugar a uno o más ojos. El grupo de la zona centro-superior de la figura A.6 contiene un ojo de tres intersecciones que dará lugar a un sólo ojo. El grupo es mortal.

Un grupo es inmortal si tiene dos o más ojos. Debido a la regla del suicidio ninguno de los dos ojos puede devenir un movimiento legal para el adversario. Se debe tener cuidado de no jugar en los ojos de un grupo propio bajo pena de convertirlo en un grupo mortal.

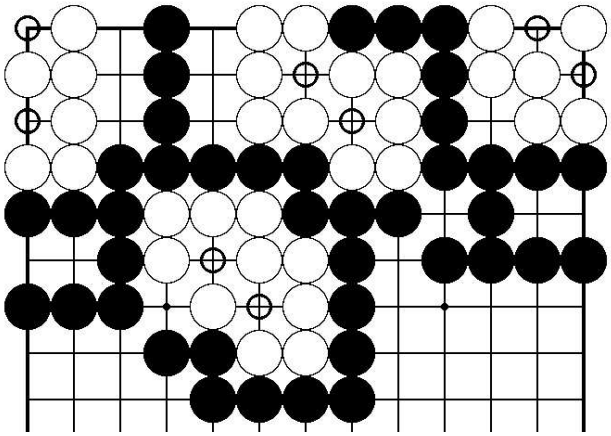

Figura A.7: Grupos blancos con dos ojos (señalados con un círculo)

En la figura A.7 aparecen distintos ejemplos de grupos blancos inmortales con dos ojos. Nótese que se puede obtener la inmortalidad bien con un sólo grupo que tenga dos ojos o bien con dos grupos desconectados que compartan dos libertades no contiguas.

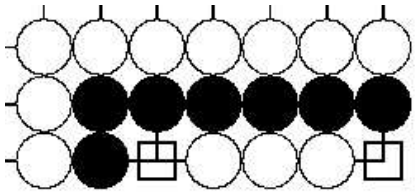

Figura A.8: *Seki*: si negro juega blanco captura, si blanco juega negro captura y tiene dos ojos

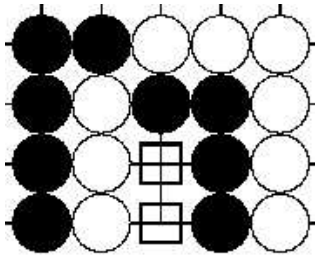

Figura A.9: El *seki* más sencillo

Hay un caso particular en que un grupo puede vivir sin necesidad de hacer dos ojos denominado *seki* . Dos grupos están vivos en *seki* cuando el primero en jugar muere. De esta forma ninguno de los dos juega en la zona y la partida termina dando ambos grupos por vivos.

En la figura A.9 tenemos un ejemplo básico de vida en *seki*, el primer jugador que ocupe una de las dos intersecciones será capturado por el adversario, así ninguno de los dos juega y la partida termina dando ambos grupos por vivos y con 0 puntos localmente para cada uno (cf. sección A.2.6). En A.8 también tenemos un *seki* aunque es un poco más complicado de ver. Si negro juega sera inmediatamente capturado por blanco. Si es blanco el que juega negro capturará cuatro piedras en línea y blanco no podrá evitar que negro consiga dos ojos y sea inmortal. De nuevo ninguno de los dos jugadores moverá en las intersecciones marcadas.

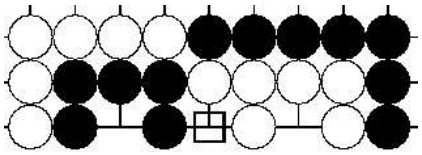

Figura A.10: *Seki* con ojos.

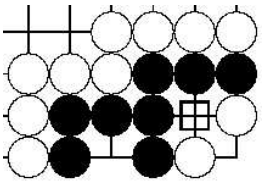

Figura A.11: *Seki* con ojo débil en la esquina

En las figuras A.10 y A.11 tenemos dos ejemplos más de *seki* donde ambos grupos tienen un sólo ojo. De nuevo ninguno de los dos juega y obtienen localmente 0 puntos cada uno.

A.2.4 *Ko*

Ko significa infinito en japonés. La regla del *ko* impide que se den situaciones de bucle infinito en el juego y dice que es ilegal realizar una jugada que lleve a la misma posición en la que estaba el tablero dos turnos atrás. Otros reglamentos como el ING (cf. sección A.2.7) incluyen la regla del *superko* que impiden la repetición de una posición independientemente del número de turnos que haya pasado desde que se dio.

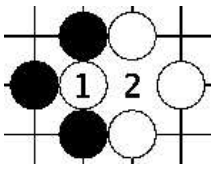

Figura A.12: No es legal recapturar una piedra por la regla del *ko*

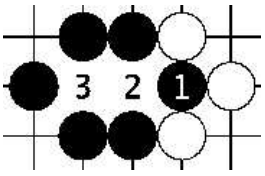

Figura A.13: Sí es legal capturar una piedra tras la captura de dos

En la figura A.12 negro acaba de capturar una piedra blanca en 1. A continuación blanco puede jugar donde quiera en el tablero excepto en 2 que es una jugada ilegal por la regla del *ko*. En cambio, en A.13, negro acaba de capturar dos piedras

blancas con su jugada en 1, a continuación es legal para blanco jugar en 2 puesto que el tablero obtenido será diferente (dos turnos antes había una piedra blanca en 3 y ahora no).

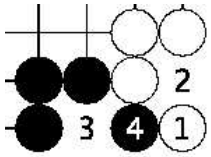

Figura A.14: *Ko* en la esquina

En A.14 tenemos un ejemplo de *ko* en la esquina. Blanco acaba de capturar en 1, para negro es ilegal jugar en 2. Nótese que si blanco capturase en 3 aparece un nuevo *ko* y sería ilegal para negro jugar en 4.

A.2.5 Sistema de categorías y handicap

Una de las peculiaridades del Go es su sistema de handicap, que permite enfrentar a jugadores de muy distinto nivel en condiciones de igualdad. Un principiante tiene un nivel de 30 *kyu* , a medida que mejora su juego va subiendo de nivel (29, 28, . . . , 10, . . . , 5, 4, 3, 2, 1) hasta primer *kyu*. Una mejora en este nivel lleva a la categoría de primer *dan* amateur y así hasta 7 *dan* (un 7 *dan* amateur es ligeramente inferior a un 1 *dan* profesional).

El sistema de handicap de Go permite enfrentar a jugadores de distinta fuerza en condiciones de igualdad colocando en los puntos de handicap tantas piedras (hasta un máximo de nueve) como diferencia de niveles haya. El jugador débil toma las negras y coloca las piedras de handicap, el jugador fuerte, con blancas, realiza la primera jugada. De esta manera un jugador de nivel 10 *kyu* puede enfrentarse en condiciones de igualdad con un amplio abanico de contrincantes que va desde 19 *kyu* (un principiante con un mes de experiencia puede alcanzar ese nivel) hasta 1 *kyu* (que es un amateur de nivel considerable).

A.2.6 Objetivo del juego y fin de partida

El objetivo del Go es rodear el máximo territorio posible, es decir, tener el control del máximo número posible de intersecciones.

Detectar el fin del juego es uno de los aspectos complicados para los no iniciados. El juego acaba cuando los dos jugadores pasan de manera consecutiva. En este punto siempre surge la misma pregunta, ¿y cuándo pasa un jugador? Llega un momento en el juego en que no hay ningún movimiento que pueda aumentar la

puntuación de ningún jugador ya que los territorios están perfectamente definidos; así una piedra en territorio propio hará que se tenga una intersección menos controlada y una piedra en territorio ajeno le dará un prisionero de más al oponente. En ese momento los dos jugadores pasan.

Cuando acaba la partida la puntuación de negro se obtiene sumando las intersecciones vacías de territorio bajo su control más las piedras blancas capturadas durante la partida. La puntuación de blanco se obtiene sumando las intersecciones vacías bajo su control más las piedras negras capturadas más el *komi*.

El *komi* es una puntuación adicional que se da a blanco para compensar la ventaja que tiene negro por comenzar él la partida. En Japón el komi es de $6\frac{1}{2}$ puntos para el tablero de 19×19.

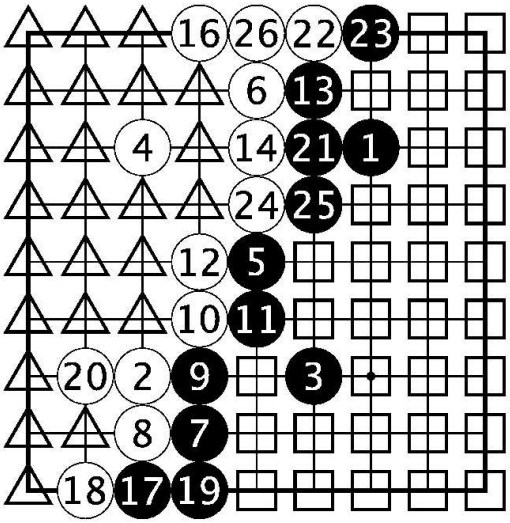

Figura A.15: Una partida completa en 9×9, negro gana de $5\frac{1}{2}$ puntos

En la figura A.15 se muestra el desarrollo de una partida completa de go sobre un tablero de 9×9. La jugada 15 de negro se produce bajo la jugada 26 y posteriormente blanco captura la piedra. Suponiendo un *komi* de $\frac{1}{2}$ puntos la puntuación de negro es de 32 puntos (correspondientes a los 32 cuadrados marcados sobre el tablero) y la de blanco es de $26\frac{1}{2}$ puntos (25 puntos de territorio marcados con triángulos más 1 punto correspondiente a la piedra negra capturada en la jugada 22 más $\frac{1}{2}$ puntos de *komi*). Negro gana la partida de $5\frac{1}{2}$ puntos.

A.2.7 Distintos Reglamentos

Cada liga profesional tiene su propio reglamento; así existen las reglas japonesas (según las que nos hemos regido para este apéndice), las chinas, las reglas ING,

las de Nueva Zelanda o las de la American Go Association entre otras. Estas cinco aparecen definidas en [10].

Las diferencias entre unas y otras son sutiles y carentes de interés para el jugador que se inicia puesto que rara vez implican un cambio en el signo de la partida. Sin embargo hay casos muy especiales en que dependiendo del reglamento utilizado el ganador será blanco o negro ante un mismo final de partida. Berlekamp y Wolfe dedican dos apéndices y casi 60 páginas en [7] a la matematización y comparación de los distintos reglamentos y presentan una curiosa situación de final de partida en que el resultado depende de las reglas elegidas.

A.3 *Semeai*: Definición y Ejemplos

Puesto que es el *semeai* el objeto principal de estudio de este proyecto merece sobradamente una sección propia en esta breve introducción al Go.

Un *semeai* es «una carrera por la captura entre dos grupos opuestos; (1) ambos completamente rodeados, (2) que comparten libertades o frontera, (3) y sin posibilidad de hacer dos ojos» [1]. El resultado final puede ser que un grupo capture al adversario o bien que ambos coexistan en *seki*.

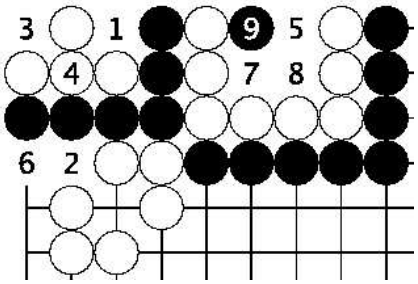

Figura A.16: *Semeai* con un ojo para cada grupo. Se muestra numerada la secuencia ganadora para negro

Dentro de este sencillo concepto caben una infinidad de situaciones dispares, algunas muy rocambolescas, que reciben el calificativo de *semeai*. La habilidad para resolver un *semeai* es crucial para el desenlace de una partida en que una de estas situaciones aparezca. Y ¿qué es resolver un *semeai*? Es determinar, bajo la suposición de juego perfecto por parte de ambos jugadores, cuál es el resultado de la carrera y cuántos movimientos de pasar pueden realizarse sin alterar el estatus de la carrera.

[1] Berlekamp y Wolfe [7] pág. 215

En la figura A.16 observamos un típico *semeai* donde ambos grupos tienen un sólo ojo. En este ejemplo gana el primer jugador que mueva (es un juego «fuzzy» según la terminología de Conway en [15] o «unsettled» según la que hemos utilizado en este trabajo). Blanco en 2 ó 6 ganan el *semeai*, la secuencia ganadora para negro la mostramos numerada en la figura.

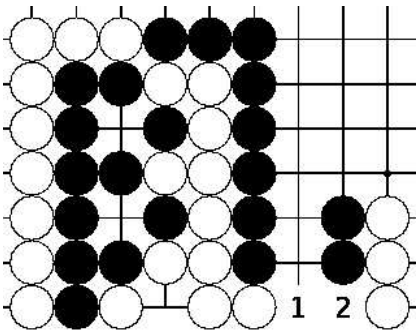

Figura A.17: Negro consigue un *seki* en triple *ko* jugando en 1

En la figura A.17 aparece una situación muy rocambolesca. Blanco gana de manera trivial si juega primero en 2. Si es negro quien tiene el turno el resultado es más complicado de dilucidar. Negro en 1 acaba la secuencia dando lugar a un *seki* en triple *ko*.

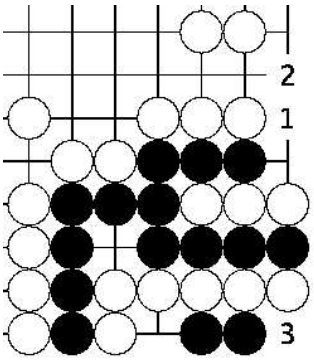

Figura A.18: La secuencia 1-3 permite a negro vivir en *hane-seki*[2]

En la figura A.18 damos una vuelta más de tuerca. De nuevo blanco en 1 gana el *semeai* pero si juega negro primero la sorprendente secuencia de 1 a 3 convierte la carrera en un *seki* o para ser más precisos un *hane-seki*. El lector

[2]Reproducido con permiso de R. Hunter [22] pág. 192

puede comprobar que si negro defiende su piedra en *atari* blanco gana la carrera, inversamente si blanco captura la piedra es él quien pierde.

A.4 Para saber más

La bibliografía sobre aprendizaje, técnicas y perfeccionamiento del juego de Go es extensísima, aunque sólo un pequeño porcentaje se traduce al inglés. En internet hay gran cantidad de recursos para convertirse de manera *on-line* en un buen jugador de Go.

Pueden visitarse los siguientes sitios web:

1. `http://www.gobase.org`
2. `http://kgs.kiseido.com`
3. `http://www.goproblems.com`
4. `http://gofme.net`

En (1) tenemos la mayor base de datos sobre Go en inglés: biografías de jugadores profesionales, partidas, historial de torneos, noticias de actualidad sobre las ligas profesionales, aprendizaje, software relacionado y un largo etcétera. En (2) un servidor de go con cliente basado en Java. Para estudiar y realizar de manera interactiva problemas de Go visitar (3). En (4) la página del club de Go de la *Facultat de Matemàtiques i Estadística* de la UPC (GoFME).

Apéndice B

El formato SGF

El formato SGF, acrónimo de *Smart Game Format*, es un formato de fichero de texto utilizado para almacenar partidas de juegos de tablero de dos jugadores (Go, Othello, Lines of Action,...). El formato se basa en el árbol de variaciones de la partida y por tanto permite, además de visualizar la rama principal que corresponde a la partida jugada, todas las variaciones adicionales que surgen durante la revisión y el comentario de la partida. El formato SGF permite añadir comentarios en cada nodo, resaltar jugadas o intersecciones del tablero con distintas etiquetas, crear posiciones añadiendo piedras,...

Este formato fue inventado por Anders Kierulf en 1987 y rápidamente se convirtió en el formato preferido para distribuir y almacenar partidas de Go en Internet. Martin Müller desarrolló la versión 3 del formato y Arno Hollosi es el principal desarrollador de la cuarta y última versión.

Además del lógico interés que este formato ha suscitado entre los jugadores de Go debido a la posibilidad de almacenar las propias partidas para su posterior revisión, obtener partidas de jugadores profesionales para su estudio, etc. resulta de gran utilidad para los programadores de Go. En primer lugar la realización de una colección de tests como la STS-RV se convierte en una tarea accesible. Por otro lado su orientación en árbol nos permite, de manera sencilla, almacenar en un fichero SGF todo el árbol de búsqueda que recorre el módulo *Semeai-01ES* en la resolución de un problema concreto. Así la posterior revisión del fichero facilita el análisis de errores y la determinación de heurísticas útiles y fiables para la mejora de la selectividad en la búsqueda.

En la figura B.1 se muestra un fragmento del fichero SGF que produce el módulo *Semeai-01ES* tras la ejecución del test 2 de la colección GSAT. Las coordenadas del tablero se codifican como dos letras minúsculas a-s. Las letras mayúsculas codifican las distintas propiedades relevantes: GM game, FF format, SZ size, AB add black stone, AW add white stone, B black move, W white move, C comment,... En [21] se encuentra la especificación completa del formato SGF.

```
(;GM[1]FF[4]SZ[19]
;AB[db][ob][dc][oc][qc][ad][bd][cd][od][pd][qe][re][bf][cf][mp][np][op][cq][dq]
[eq][fq][jq][lq][mq][oq][qq][rq][br][fr][jr][lr][pr][rr][sr][qs]
AW[ab][cb][fb][mb][cc][fc][lc][nc][rc][dd][ed][fd][nd][qd][rd][ne][oe][pe][pg]
[qg][rg][rn][sn][bo][do][ro][ap][cp][pp][qp][rp][sp][bq][nq][pq][sq][cr][dr][er]
[mr][nr][or][ns][rs]

(;W[ar] C[: unknown Eval=-15001 Depth=1 a=-15003 b=-15001 in 1 nodes, node 3
: won Eval=-15001 Depth=2 a=-15003 b=15002 in 2 nodes, node 79
]
(;B[es] C[: won Eval=15002 Depth=0 a=-15003 b=15002 in 0 nodes, node 6
: won Eval=15002 Depth=1 a=-15003 b=15002 in 0 nodes, node 92
] )
(
(;B[ds] C[: won Eval=15002 Depth=0 a=-15003 b=15002 in 0 nodes, node 8
: won Eval=15002 Depth=1 a=-15003 b=15002 in 0 nodes, node 94
] )
(
(;B[cs] C[: won Eval=15002 Depth=0 a=-15003 b=15002 in 0 nodes, node 9
: won Eval=15002 Depth=1 a=-15003 b=15002 in 0 nodes, node 90
] )
(
(;B[fs] C[: won Eval=15002 Depth=0 a=-15003 b=15002 in 0 nodes, node 11
: won Eval=15002 Depth=1 a=-15003 b=15002 in 0 nodes, node 95
] )
(
(;B[bs] C[: unknown Eval=-15001 Depth=0 a=-15003 b=15002 in 0 nodes, node 14
: won Eval=-15001 Depth=1 a=15002 b=15002 in -1 nodes, node 82
]
(;W[as] C[: unknown Eval=-15001 Depth=0 a=-15003 b=15002 in 0 nodes, node 85
] )
(;W[cs] C[: won Eval=15002 Depth=0 a=-15001 b=15002 in 0 nodes, node 88
] ))
(;B[as] C[: won Eval=15002 Depth=0 a=-15003 b=-15001 in 0 nodes, node 17
: won Eval=15002 Depth=1 a=-15003 b=15002 in 0 nodes, node 98
] ))))))
```

Figura B.1: Fragmento del fi chero sgf de la solución del test GSAT-02.

Apéndice C

Terminología de Go

Atari Un grupo se dice que está en *atari* si sólo tiene una libertad. El grupo puede por tanto ser capturado por el adversario en una sola jugada. En la figura A.4 el grupo de cinco piedras negras de la izquierda está en *atari*.

Damezumari En la bibliografía anglosajona aparece también como «shortage of liberties». Situación en la un jugador no puede realizar un movimiento debido a una falta de libertades en su grupo.

Dan Ver *kyu*.

Grupo Conjunto maximal de piedras de un mismo color conectadas por líneas del tablero. En la literatura aparece también como cadena o armada.

Ko Regla del juego que impide la repetición de la misma situación del tablero entre el turno de juego i y el turno $i+2$ (cf. sección A.2.4).

Kyu La categoría de un jugador de Go está perfectamente tabulado permitiendo así un sistema de handicap muy versátil. Un jugador que acaba de aprender las reglas se le atribuye un nivel de 30 *kyu*. A medida que el jugador sube de nivel su nivel evoluciona a 20, 10, 3, 2 hasta 1 *kyu*. A partir de aquí el jugador entrará en la franja de amateurs fuertes obteniendo un nivel de 1 *Dan* y así hasta 7 *Dan* que es el máximo nivel de amateur. Por encima sólo están los profesionales que van desde 1 *Dan* profesional hasta 9 *Dan* profesional.

Hane Movimiento en diagonal en contacto con una piedra enemiga.

Miai Dos puntos del tablero son *miai* si cuando un jugador ocupa uno de ellos su oponente ocupará el otro y viceversa.

Nakade Se dice que una forma de ojo es *nakade* si puede ser reducida a un ojo de un solo espacio por el oponente bajo condiciones de juego alternado. En la figura C.1 se presentan todas las posibles formas *nakade*.

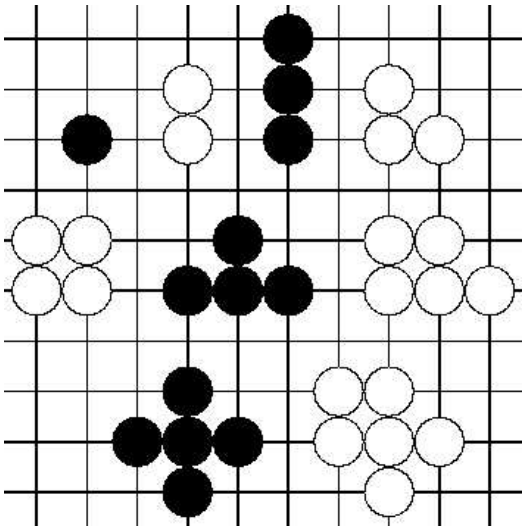

Figura C.1: Las distintas formas *nakade*.

Ojo Una intersección vacía completamente rodeada por uno o más grupos del mismo color. Hablaremos de un gran ojo cuando tengamos un conjunto de intersecciones vacías completamente rodeadas (normalmente hasta un máximo de ocho intersecciones). El teorema más importante del juego de Go está relacionado con el concepto de ojo y dice: Todo grupo que tenga dos o más ojos es inmortal independientemente de cuántas veces seguidas juegue el oponente.

Sente Tener la iniciativa en el juego. Un movimiento es *sente* si permite conservar la iniciativa tras completar la secuencia. Su contrario es *gote*.

Seki Se dice que dos grupos están vivos en *seki* cuando el que inicie la secuencia para matar al otro morirá. De esta manera ninguno de los dos juega localmente y la partida acaba con los dos grupos considerados vivos «en seki» (cf. sección A.2.3).

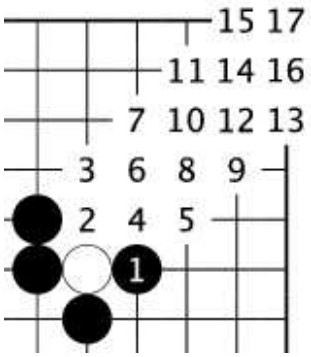

Figura C.2: Secuencia de 17 movimientos que permite la captura de blanco en *shisho*.

Semeai Carrera de libertades entre dos grupos que no pueden vivir simultáneamente (cf. secciones 3.2 y A.3)

Shisho Técnica de captura que provoca la muerte del adversario tras una secuencia de *ataris* que le aplasta contra el borde del tablero (cf. figura C.2).

Uttegaeshi El término anglosajón es *snapback*. Situación que se produce cuando tras capturar una piedra enemiga el grupo propio queda con una sola libertad y es capturado.

Tesuji El movimiento más hábil e ingenioso en una situación local.

Tsumego Problemas de vida y muerte normalmente localizados en la esquina.

Yose Es el final de la partida. Las tres fases de una partida son: el *fuseki* o inicio, el *chuban* o medio juego y el *yose*. El *yose* a su vez se divide en *yose* grande (al principio cuando podemos encontrar jugadas de hasta 15 puntos de valor) y *yose* pequeño (al final de todo donde las jugadas son como máximo de dos o tres puntos).

Apéndice D

Clases de equivalencia para {5,6,7}–formas de ojo

A continuación presentamos el conjunto completo de formas de ojo de tamaño 5, 6 y 7 agrupadas por classes de equivalencia según la *neighbour classification*.

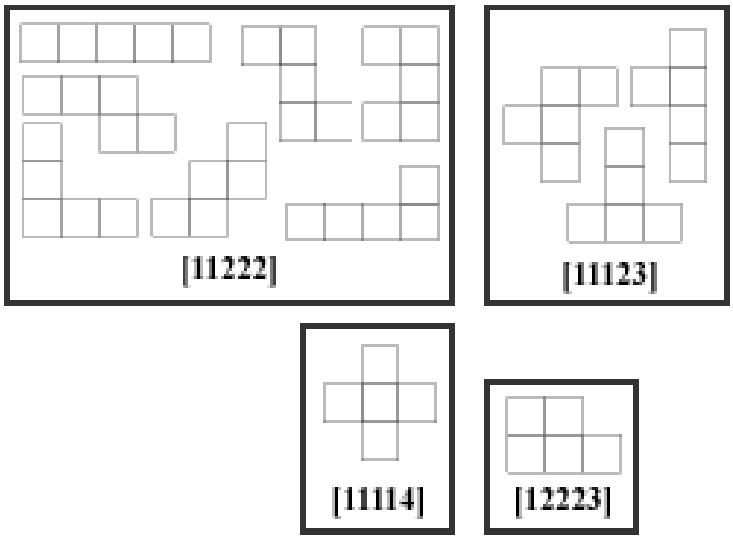

Figura D.1: El conjunto completo de pentominoes agrupados por clases.

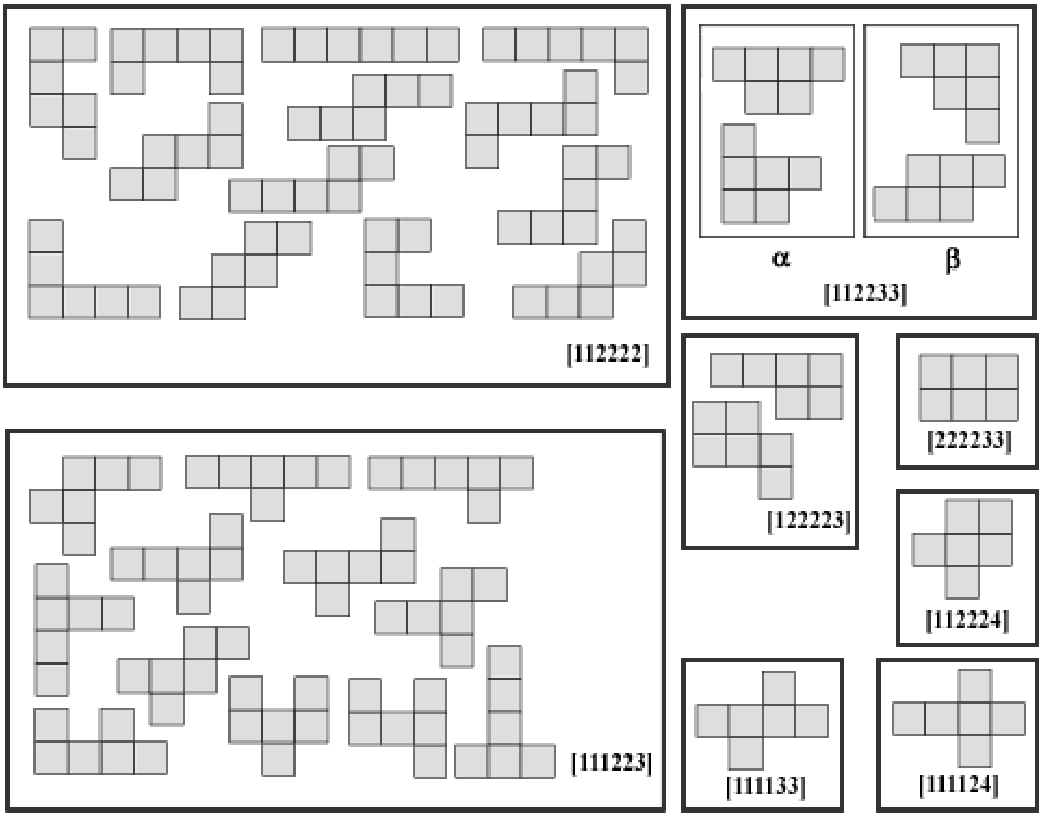

Figura D.2: El conjunto completo de hexominoes agrupados por clases.

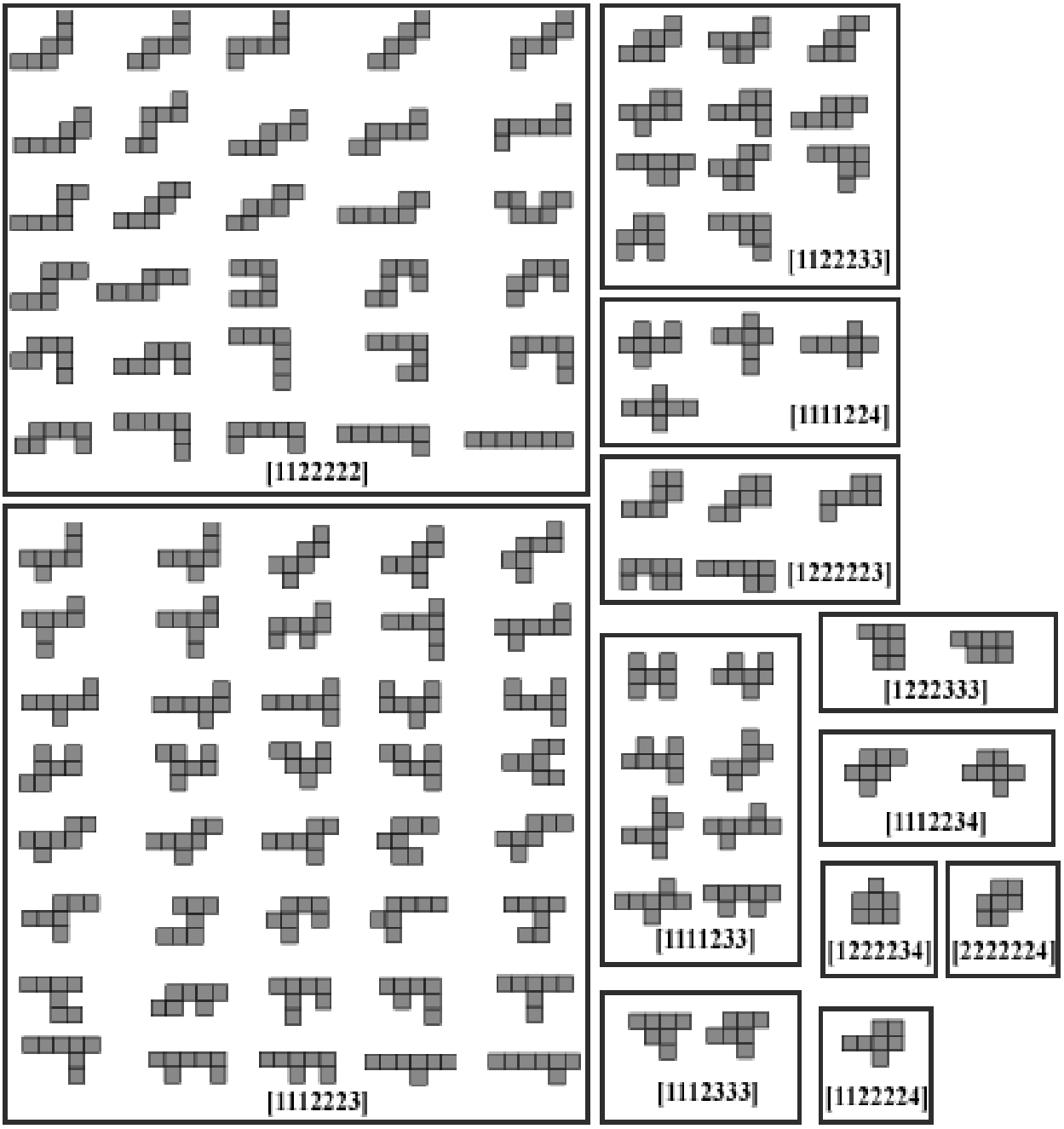

Figura D.3: El conjunto completo de heptominoes agrupados por clases.

Apéndice E

Resultados detallados de *Semeai-01ES* vs STS-RV

A continuación presentamos detalladamente los resultados obtenidos por el módulo *Semeai-01ES* frente a los tres conjuntos de test más relevantes de la STS-RV.

En la tabla E.1 explicamos los acrónimos utilizados en las tablas del presente apéndice:

NT	Número de test
PB	Profundidad de búsqueda
R	Resultado (conforme a la tabla 4.10)
M	Movimiento jugado
T	Tiempo (s)
NN	Número de nodos explorados
FE	Número de llamadas a la función de evaluación
Hi	Utilización de la heurística i

Tabla E.1: Leyenda

Los resultados presentados se han obtenido, al igual que en la sección 5.3, con un límite de exploración de 10000 nodos y con todas las heurísticas activadas. No hemos incluido los resultados para los tests de clase 0, 1 y e puesto que la gran mayoría de ellos se resuelven de manera estática al nivel del nodo raíz.

NT	PB	R	M	T	NN	FE	H0	H1	H2	H3	H4	H5
1	7	15002	B1	0.15	465	47	313	90	26	0	0	7
2	3	15002	A2	0.01	13	0	15	1	0	0	0	0
3	4	15002	Q1	0.01	34	0	29	2	0	0	0	0
4	5	-15002	Q1	0.02	48	1	59	1	0	0	0	0
5	7	15002	S18	0.59	1705	19	146	89	8	0	0	744
6	5	15002	R14	0.16	344	7	189	53	4	0	0	182
7	10	15002	B18	1.17	4139	450	1087	980	10	0	93	1990
8	5	15002	C19	0.10	276	14	37	31	0	0	5	183
9	8	-15001	R2	4.91	13105	0	2024	623	0	0	83	462
10	9	-15001	Q5	5.45	14863	26	5984	805	6	0	368	1742
11	5	15002	D1	0.07	166	0	54	23	0	0	0	35
12	5	15002	D1	0.13	313	0	195	83	0	0	7	39
13	5	15002	C18	0.02	55	9	53	17	9	0	0	2
14	2	15002	F19	0.00	1	0	0	1	0	0	0	0
15	6	15002	R18	0.14	304	26	102	56	23	0	0	53
16	3	15002	R18	0.01	20	4	0	7	2	0	0	4
17	7	15002	E1	0.13	376	12	198	81	0	0	0	24
18	2	15002	D1	0.00	2	0	5	1	0	0	0	0
19	5	15002	P2	0.17	315	0	40	46	0	0	0	6
20	9	15002	P3	8.90	19271	1	3971	9348	0	0	1	2409
21	7	15002	P17	0.16	370	46	240	99	35	0	0	112
22	2	15002	Q19	0.00	1	0	0	1	0	0	0	0
23	8	15002	D19	0.42	1336	17	513	254	0	0	0	187
24	2	15002	E19	0.01	1	0	0	1	0	0	0	0
25	10	-15001	D1	6.91	13656	355	7911	1571	97	0	0	594
26	5	15002	F1	0.10	252	5	41	45	4	0	0	0
27	5	15002	S1	0.03	87	27	50	18	7	0	0	7
28	2	15002	R2	0.00	2	1	5	0	1	0	0	0
29	3	15002	S18	0.01	8	3	9	1	0	0	0	20
30	2	15003	S18	0.00	4	1	2	0	0	0	0	10
31	5	15002	A18	0.03	50	93	0	0	0	0	0	83
32	2	15003	A18	0.00	6	18	0	0	0	0	0	11
33	5	15002	B1	0.01	65	5	28	15	2	0	0	7
34	5	15002	E3	0.03	74	5	43	16	0	0	0	7
35	6	15002	T2	0.06	186	26	122	23	0	0	0	0
36	2	15002	T2	0.00	2	0	5	1	0	0	0	0
37	7	15002	S19	0.05	155	36	79	30	0	0	0	18
38	2	15002	R19	0.00	1	0	0	1	0	0	0	0
39	3	15002	D19	0.00	9	0	18	2	0	0	0	0
40	2	15002	A19	0.00	2	0	4	0	0	0	0	0
41	7	15002	C2	0.20	592	1	181	92	1	0	0	0
42	2	15002	A4	0.00	1	0	0	1	0	0	0	0
43	4	15002	S1	0.03	49	1	39	8	1	0	0	0
44	7	15002	S1	0.21	615	52	370	59	5	0	17	93

Tabla E.2: Resultados detallados semeais_GSAT

NT	PB	R	M	T	NN	FE	H0	H1	H2	H3	H4	H5
45	9	15002	O18	4.55	10025	0	2895	1051	0	0	0	46
46	3	15002	O18	0.02	36	0	0	18	0	0	0	0
47	7	15002	B17	1.02	2307	95	354	255	31	0	0	64
48	3	15002	D19	0.01	25	0	0	11	0	0	0	0
49	5	15002	C1	0.02	39	4	22	3	0	0	0	25
50	2	15002	C2	0.00	3	1	5	0	0	0	0	1
51	9	15002	Q1	0.48	1332	21	617	359	0	0	16	939
52	5	15002	Q1	0.06	163	0	61	29	0	0	0	166
53	7	15002	S15	0.09	245	0	70	80	0	0	0	3
54	2	15002	T19	0.00	1	0	0	1	0	0	0	0
55	5	15002	B19	0.02	68	1	35	12	0	0	0	27
56	5	15002	D19	0.03	75	3	46	14	0	0	0	10
57	7	15002	C1	1.59	3464	2	758	381	0	0	0	182
58	3	15002	E4	0.02	31	0	0	13	0	0	0	0
59	8	-15001	R1	6.85	20166	146	2430	1069	56	0	73	1255
60	5	15002	R1	0.23	681	0	19	147	0	0	0	25
61	7	15002	Q19	0.14	412	76	153	46	25	0	0	99
62	2	15003	P19	0.00	5	2	0	0	2	0	0	0
63	0	-15003	Pass	0.00	0	1	0	0	0	0	0	0
64	0	15003	Pass	0.00	0	1	0	0	0	0	0	0
65	7	15002	D1	0.04	118	28	56	28	0	0	0	0
66	3	15002	D1	0.01	19	2	2	5	0	0	0	0
67	0	15002	T3	0.00	0	1	0	0	0	0	0	0
68	0	15002	O3	0.00	0	1	0	0	0	0	0	0
69	2	15003	S19	0.01	3	2	0	0	1	0	0	0
70	2	15003	R18	0.00	1	1	0	0	1	0	0	0
71	9	-15001	B17	8.59	18332	334	3789	569	255	0	3	1319
72	4	15002	E19	0.05	65	0	23	22	0	0	0	0
73	4	15002	D1	0.01	28	0	12	10	0	0	0	2
74	2	15002	D1	0.01	2	0	4	1	0	0	0	0
75	9	15002	R1	2.45	5425	41	2676	674	0	0	47	702
76	3	15002	Q1	0.01	21	0	2	8	0	0	0	19
77	5	15002	S19	0.12	247	2	150	20	0	0	0	13
78	9	15002	S19	1.67	4243	126	3058	552	19	0	705	271
79	5	15002	B17	0.03	105	1	38	18	1	0	0	0
80	2	15002	D19	0.00	1	0	0	1	0	0	0	0
81	12	-15001	A2	5.14	12681	339	3844	1667	0	0	76	645
82	2	15002	A2	0.00	1	0	0	1	0	0	0	0
83	8	-15001	O1	5.24	11336	224	1223	679	68	0	55	231
84	7	15002	Q2	0.93	2126	16	826	253	2	0	3	57
85	3	15002	T18	0.01	18	0	19	2	0	0	0	0
86	2	15002	P19	0.00	2	0	5	1	0	0	0	0
87	8	-15001	B18	3.99	11320	253	830	814	53	0	28	373
88	5	15002	F17	0.28	606	15	36	83	6	0	0	38

Tabla E.3: Resultados detallados semeais_GSAT (cont.)

NT	PB	R	M	T	NN	FE	H0	H1	H2	H3	H4	H5
1	7	15002	E9	0.40	846	81	781	89	52	0	0	6
2	2	15003	A13	0.00	3	1	11	0	1	0	0	0
3	3	15002	Q5	0.01	25	1	2	11	0	0	0	0
4	5	15002	T7	0.28	504	85	105	119	0	0	0	0
5	8	15002	O16	5.30	9893	42	2596	1051	0	0	0	4707
6	7	15002	R15	3.61	7474	35	1074	292	0	0	0	4912
7	4	15002	H19	0.02	45	0	23	14	0	0	0	0
8	4	15002	C19	0.03	49	0	24	15	0	0	0	0
9	7	15002	F18	0.53	1173	74	675	93	22	0	231	202
10	5	15002	A17	0.11	224	25	81	5	4	0	0	49
11	11	15002	M16	11.82	18485	73	3896	555	21	16	2255	22418
12	9	15002	T12	3.12	4797	16	1445	90	7	0	195	4445
13	9	-15001	G4	5.59	10194	937	3807	22	0	0	6728	40
14	9	-15001	G5	8.99	18009	2378	499	0	0	0	13419	92
15	8	-15001	T17	11.82	12137	11	723	243	0	0	0	8920
16	2	15002	Q16	0.01	2	0	17	1	0	0	0	0
17	8	-15001	E6	14.39	17197	7	8631	7	0	0	2	4327
18	7	-15001	D1	10.98	11549	11	1282	0	0	0	2	1188
19	3	15002	C18	0.01	17	0	2	6	0	0	1	8
20	5	15002	B14	0.07	175	23	42	14	0	0	3	60
21	0	15002	T4	0.00	0	1	0	0	0	0	0	0
22	0	15002	O6	0.00	0	1	0	0	0	0	0	0
24	9	-15001	B9	16.91	17736	26927	0	0	0	0	3211	133972
25	3	15002	M18	0.01	19	2	2	5	2	0	0	0
26	9	15002	P19	0.31	895	22	322	352	0	0	0	10
27	8	-15003	D19	0.68	1292	82	444	376	12	0	0	390
28	3	15002	G17	0.01	16	1	2	5	1	0	0	15
29	19	-15001	D1	0.31	1114	7	772	209	0	0	197	19
30	2	15002	B1	0.00	2	0	5	1	0	0	0	0
31	9	15002	P1	0.81	2658	274	1009	236	0	0	277	405
32	6	15002	M1	0.17	451	5	155	60	0	0	82	50
33	11	-15001	S19	4.49	14556	550	4395	984	0	0	1069	2578
34	7	15002	P19	0.83	2325	96	222	137	0	0	553	412
35	10	-15001	A14	9.66	26369	334	644	392	0	0	18	64076
36	9	-15001	D18	4.19	10197	28	84	26	0	0	0	30785
37	9	-15001	B1	6.58	15167	5133	2320	1042	32	0	6285	6060
38	2	15003	G1	0.01	9	1	0	0	0	0	0	5
39	8	-15001	R5	11.82	20163	0	4053	1512	0	0	105	4288
40	7	-15001	P1	11.25	12086	0	350	288	0	0	2	2769
41	12	15002	T18	2.67	7749	724	3234	868	11	0	2743	502
42	2	15002	T18	0.00	1	0	0	1	0	0	0	0
43	9	15002	E1	0.88	2544	36	1310	390	0	0	0	279
44	3	15002	E2	0.01	20	0	1	8	0	0	0	0
45	6	15002	O18	0.28	748	6	238	109	0	0	0	33
46	3	15002	R18	0.01	22	0	3	9	0	0	0	0
47	5	-15003	A9	0.04	100	7	64	24	0	0	0	0

Tabla E.4: Resultados detallados semeais_RH

NT	PB	R	M	T	NN	FE	H0	H1	H2	H3	H4	H5
48	3	15002	E10	0.01	12	5	2	2	0	0	0	0
49	8	-15001	D4	11.71	21686	748	1066	307	51	0	0	3944
50	7	-15001	G6	7.57	10787	145	1913	200	0	0	0	9067
51	10	3	M16	7.94	18155	1224	1626	854	0	0	0	34057
52	13	3	N17	5.18	13181	752	2828	1346	0	0	0	18792
53	8	-15001	F19	6.46	12838	496	421	189	78	0	0	20395
54	8	-15001	D16	6.84	14861	786	550	119	188	0	0	22709
55	13	15002	R12	0.25	523	908	0	0	0	0	535	20
56	14	-15003	R18	0.62	1343	2167	0	0	0	0	1594	8
57	9	-15001	D4	7.40	10162	0	12494	230	0	0	289	9789
58	8	-15001	E1	11.06	23072	0	1712	1	0	0	376	16925
59	5	15002	J7	0.40	704	0	86	99	0	0	3	660
60	8	-15001	K1	7.16	12270	82	3427	94	0	0	79	13200
61	0	-15003	P	0.00	0	1	0	0	0	0	0	0
62	0	15003	P	0.00	0	1	0	0	0	0	0	0
63	19	2	D5	3.70	7439	11699	0	0	0	0	6359	8214
64	12	15002	E1	6.44	11621	19837	0	0	0	0	8657	18480
65	8	-15001	P3	6.77	15825	547	950	251	117	0	5215	3451
66	8	-15001	P1	4.87	11817	366	1674	822	86	0	755	1065
67	7	-15001	A9	6.46	11630	69	278	8	3	0	815	5144
68	7	-15001	C1	6.71	11027	26	84	7	0	0	4423	2850
69	12	15002	N17	0.82	2506	163	1303	376	7	0	0	265
70	6	15002	O19	0.17	465	1	220	61	1	0	0	93
71	8	-15001	H8	16.50	17588	1331	2400	79	348	0	5076	2851
72	2	15003	H8	0.00	1	1	0	0	1	0	0	0
73	8	-15001	A18	13.72	21909	0	394	0	0	166	3058	61586
74	8	-15001	C18	16.50	23821	2	21	0	0	38	12451	63398
75	13	15002	O19	0.22	563	166	217	15	44	0	462	110
76	3	15002	O19	0.01	12	7	6	0	0	0	9	0
77	6	-15001	E7	12.32	12226	0	0	0	0	0	7537	58460
78	6	-15001	F14	16.60	16337	0	0	0	0	0	603	79854
79	6	-15001	E12	8.76	10732	0	0	0	0	0	4129	23140
80	6	-15001	B9	8.31	10994	0	0	0	0	0	2619	32284
81	8	-15001	A1	12.29	15254	427	85	0	11	0	56105	59131
82	9	-15001	A6	26.74	24822	689	2626	0	0	0	62226	88863
83	9	-15001	S10	17.02	20845	1589	1116	0	422	0	77702	56543
84	8	-15001	T19	11.68	16558	1970	108	0	643	0	73701	41590
85	9	15003	T11	1.59	4254	346	65	10	0	0	0	9178
86	10	-15001	T13	3.95	11360	1540	416	102	1	0	0	21077
87	10	15002	E18	1.04	3074	68	548	1029	27	0	0	71
88	5	15002	E18	0.12	258	4	154	54	0	0	0	1
89	9	-15001	L5	8.80	12967	3	7700	778	3	0	4	4756
90	4	15002	E3	0.05	70	0	47	22	0	0	0	6
91	8	-15001	H3	4.56	12300	1	493	195	0	0	82	7086
92	9	-15001	G2	12.93	27649	9	3294	1715	1	0	461	12637
93	0	-15003	P	0.00	0	1	0	0	0	0	0	0
94	0	15003	P	0.00	0	1	0	0	0	0	0	0

Tabla E.5: Resultados detallados semeais_RH (cont.)

NT	PB	R	M	T	NN	FE	H0	H1	H2	H3	H4	H5
1	0	-15003	P	0.00	0	1	0	0	0	0	0	0
2	0	15003	P	0.00	0	1	0	0	0	0	0	0
3	0	-15003	P	0.00	0	1	0	0	0	0	0	0
4	0	15003	P	0.00	0	1	0	0	0	0	0	0
5	2	15003	T15	0.00	1	3	0	0	0	0	2	3
6	0	15002	S11	0.00	0	1	0	0	0	0	0	0
7	0	15002	N7	0.00	0	1	0	0	0	0	0	0
8	0	15002	T7	0.00	0	1	0	0	0	0	0	0
9	3	15002	J1	0.01	17	11	0	0	0	0	73	0
10	2	15003	D4	0.00	7	2	0	0	0	0	32	0
11	0	15002	L11	0.00	0	1	0	0	0	0	0	0
12	8	3	L11	26.51	25897	1664	0	0	0	0	74779	50485
13	0	15002	B9	0.00	0	1	0	0	0	0	0	0
14	19	3	B9	0.03	18	127	0	0	0	0	108	0
15	2	15003	C19	0.00	1	1	0	0	0	0	1	2
16	5	15002	B19	0.07	132	44	0	0	0	0	101	272
17	5	15002	R15	0.05	102	1	91	22	1	0	0	123
18	2	15003	N19	0.01	6	1	6	1	1	0	0	10
19	9	15002	T5	0.10	218	45	55	64	0	0	3	488
20	2	15003	O1	0.00	2	1	1	0	0	0	0	9
21	5	15002	C16	0.08	166	14	62	32	9	0	0	235
22	3	15002	F16	0.01	24	1	14	5	1	0	0	35
23	7	15002	A10	0.26	660	21	215	36	0	0	0	354
24	12	15002	A10	3.68	6614	201	2341	842	0	0	0	3126
25	0	15002	P15	0.00	0	1	0	0	1	0	0	0
26	0	15002	Q18	0.00	0	1	0	0	1	0	0	0
27	0	15002	Q5	0.00	0	1	0	0	1	0	0	0
28	0	15002	Q2	0.00	0	1	0	0	1	0	0	0
29	5	15002	C16	0.04	86	8	35	19	6	0	0	113
30	3	15002	F16	0.00	10	2	4	4	2	0	0	13
31	6	15002	A10	0.11	254	10	82	20	0	0	0	132
32	12	15002	A10	3.88	6485	238	1727	285	0	0	2	1248
33	13	-15001	G1	4.59	11951	1886	1563	125	0	33	24381	5096
34	2	15003	G1	0.00	3	1	0	0	0	0	9	5
35	11	2	D4	6.54	12895	4750	113	244	14	276	36765	422
36	7	15003	D4	0.28	560	228	1	0	0	0	2072	125
37	11	2	D4	5.69	11844	2210	230	13	10	312	33882	300
38	7	15003	D4	0.37	719	225	12	0	0	0	2577	163
39	2	15003	O10	0.01	2	2	0	0	0	0	6	6
40	8	-15001	N11	7.92	10829	814	0	0	6	0	28722	33211
41	2	15003	C19	0.00	5	1	0	0	0	0	26	0
42	13	-15001	A9	4.84	12227	1331	2359	71	0	185	29616	27
43	2	15003	C19	0.00	5	1	0	0	0	0	26	0
44	12	-15001	B9	4.65	11980	1140	1529	31	0	122	21430	32
45	11	-15001	D19	7.28	13612	4760	628	0	0	0	24948	17058
46	9	15002	D19	5.36	9203	2637	455	1	0	0	15627	12894
47	2	15003	A18	0.01	2	3	0	0	0	0	12	13
48	2	15003	A18	0.00	2	2	0	0	0	0	12	12
49	2	15002	T19	0.00	1	0	0	1	0	0	0	0
50	7	15002	S16	0.09	163	33	47	52	0	0	0	29
51	7	15002	H12	2.08	2955	84	321	251	0	0	0	5537
52	8	-15001	J11	5.47	10555	324	444	108	0	0	0	14350
53	7	-15001	G19	7.48	10455	0	73	14	0	0	1118	2184
54	8	-15001	H19	14.28	21213	0	6333	215	0	0	257	4662

Tabla E.6: Resultados detallados semeais_Misc

Agradecimientos

Agradece a la llama su luz, pero no olvides
el pie del candil que, constante y paciente,
la sostiene en la sombra.
R. Tagore

Han sido muchos los que me han ayudado a llegar hasta aquí, hasta la última página de este largo proyecto. En mayor o menor medida estoy en deuda con todos los que a continuación aparecen, espero que algún día les pueda compensar.

Je veux remercier Tristan Cazenave qui un jour d'octobre 2003, après une brève reunion de même pas vingt minutes, décida de m'accepter en tant qu'etudiant en DEA.

Gràcies a en Josep M. Brunat, cap d'estudis de la FME, per saber avantposar les persones als papers, la realitat a la burocràcia i donar-me la oportunitat de marxar sis mesos a França.

Vull agraïr molt especialment el suport i l'ajuda d'en Marco A. Peña. Gràcies Marco per la teva honestedat i pels teus consells.

Estoy en deuda con Jaime L. Krahe que me trató en Paris VIII como a uno más de sus estudiantes sin ni tan siquiera serlo.

Je remercie tous mes copains du labo: Bernard, Vincent, Anaïs, Farrès, Marc, Dominique, JJ, Jean, Marie-Solange,... Ce fût un plaisir de se lever chaque jour pour aller au labo sachant que je vous y retrouverais.

Merci beaucoup Erik Adelbert pour être mon amphitrion, mon traducteur, mon professeur et mon ami. Je me souviens encore de la première chose que tu m'as dit: "Ici il n'y a pas des questions stupides".

Je veux aussi remercier Nicolas Jouandeau pour tout ce qu'il m'a appris et tout ce qu'il m'a donné. Merci Nico pour ton sourire, ton sens de l'humour, ton hospitalité, ta générosite et pour toutes les longues conversations qu'on a eu.

Many thanks to Richard Hunter for his book and for his colaboration with this project.

Thanks also to Gunnar Farnebäck for his valuable help during the process of

creation and distribution of the STS-RV.

Vull agraïr a en Pau Bofill haver-me descobert aquest apassionant e infinit món que és el Go. Gràcies Pau per aquest regal tan maco.

Gràcies als amics de la uni, als de GoFME i als de sempre per no haver-me oblidat quan estava fora, pels vostres e-mails, pel vostre suport i per les vostres visites.

Gracias a las Pilis por haber estado siempre ahí, por haberme apoyado en todo lo que hago y por haber seguido mis pasos desde la distancia.

Gràcies Mayita per marcar sempre el nord al mig de la negra nit.

Per últim vull agraïr a la Judith, culpable principal de l'existència d'aquest projecte, haver decidit recòrrer al meu costat aquest meravellós viatge que és la vida.

Índice de figuras

Índice de tablas

Bibliografía

[1] L. Allis, H. J. van den Herik, and M. Huntjens. Go-moku solved by new search techniques. *Computational Inteligence*, 1995.

[2] L.V. Allis. *Searching for solutions in Games and Artificial Intelligence*. PhD thesis, Vrije Universitat, Amsterdam, 1994.

[3] Dragos Bajenaru. Personal communication, February 2004.

[4] Karl Baker. *The Way to Go*. The American Go Association, 1986.

[5] David Benson. Life in the game of go. *Information Sciences*, 10:17–29, 1976.

[6] Elwyn Berlekamp, John H. Conway, and Richard Guy. *Winning Ways for your matematical plays*, volume I-II. Academic Press, 1982.

[7] Elwyn Berlekamp and David Wolfe. *Mathematical Go. Chilling gets the last point*. A K Peters, Ltd., 1994.

[8] Bruno Bouzy. *Modélisation cognitive du joueur de Go*. PhD thesis, Université Paris 6, 1995. http://www.math-info.univ-paris5.fr/ bouzy.

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

[10] Richard Bozulich. *The Go Player's ALMANAC*. Ishi Press, 1992.

[11] Richard Bozulich. *Get Strong at Tesuji*, volume 6 of *Get Strong at Go series*. Kiseido Publishing Company, Tokyo, Japan, 1996.

[12] Tristan Cazenave. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go*. PhD thesis, Université Pierre et Marie Curie, Paris 6, 1996.

[13] Tristan Cazenave. A generalized threats search algorithm. In *Proceedings or Computers and Games*, Edmonton, Canada, 2002.

[14] Ken Chen and Zhixing Chen. Static analysis of life and death in the game of go. *Information Sciences*, 121:113–134, August 1999.

[15] John H. Conway. *On numbers and games*. A K Peters, Ltd., 2nd edition, 2001.

[16] James Davies. *Life and Death*. Elementary Go Series. Ishi Press, 1975.

[17] Bernard Desgraupes. *LATEX Apprentissage, guide et référence*. Vuibert, 2e edition, 2003.

[18] Gunnar Farnebäck. Personal communication, April 2004.

[19] David Fotland. Static eye analysis in "The Many Faces of Go". *ICGA*, 25(4):203–210, 2002.

[20] The Free Software Foundation. *The GNU Go Program Documentation*, 3.2 edition.

[21] Arno Hollosi. Sgf file format 4. http://www.red-bean.com/sgf/.

[22] Richard Hunter. *Counting Liberties and Winning Capturing Races*. Slate and Shell, 2003.

[23] Richard Johnsonbaugh and Martin Kalin. *Object Oriented Programming in C++*. Prentice Hall, 2nd edition, 2000.

[24] et al. Jonathan Schaeffer. Building the checkers 10-piece endgame databases. In E. Heinz H. van den Herik, H. Iida, editor, *Advances in Computer Games 10*, pages 193–210. Kluwer Academic Publicshers, 2003.

[25] Donald Knuth. *The Art of Computer Programming*, volume I, III. Addison Wesley, 3rd edition, 1997.

[26] Howard Landman. Eyespaces values in go. *Games of No Chance*, 29:227–257, 1996.

[27] Mike Loukides and Andy Oram. *Programming with GNU software*. O'Reilly, 1st edition, January 1997.

[28] Martin Müller. *Computer Go as a sum of local games: an application of combinatorial game theory*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1995. http://www.cs.ualberta.ca/ mmueller.

[29] Martin Müller. Playing it safe: Recognizing secure territories in computer go by using static rules and search. In H. Matsubara, editor, *Game Programming Workshop in Japan '97*, Computer Shogi Association, pages 80–86, Tokyo, Japan, 1997.

[30] Martin Müller. Race to capture: Analyzing semeai in go. In *Game Programming Workshop in Japan*, volume 99(14) of *IPSJ Symposium Series*, pages 61–68, 1999.

[31] http://mathworld.wolfram.com/Polyomino.html.

[32] Stuart Russell and Peter Norvig. *Artificial Intelligence. A modern approach*. series in Artificial Intelligence. Prentice Hall, 2nd edition, 2003.

[33] Edward Thorpe and William Walden. A partial analysis of go. *Computer Journal*, 7(3):203–207, 1964.

[34] Edward Thorpe and William Walden. A computer assisted study of go on mxn boards. *Information Sciences*, 4:1–33, 1972.

[35] Ricard Vilà and Tristan Cazenave. When one eye is enough. In *Many Games, Many Challenges. Proceedings of the 10th Advances in Computer Games Conference*, pages 109–124. Kluwer Academic Publishers, 2003.

Índice alfabético