
**Universidad Carlos III de Madrid**

Escuela politécnica superior

Ingeniería Informática

Proyecto Fin de Carrera

# Creación de un cliente para razonar en juegos de propósito general

**Autor**: Ignacio Navarro Martín

**Tutor**: Carlos Linares López

*"La estupidez real siempre vence a la inteligencia artificial."*

Terry Pratchett

# *Agradecimientos*

Agradezco el apoyo, tutorización y consejo ofrecido por mi tutor de proyecto Carlos Linares López. Además agradezco el apoyo mostrado por mis padres durante el desarrollo de mis estudios de Ingeniería Informática ...

# Índice general

# Índice de figuras

# Índice de cuadros

# Capítulo 1

# Introducción

Uno de los objetivos de la inteligencia artificial (IA) es encontrar un sistema capaz de razonar de forma inteligente (Russel and Norving, 1995). Este sistema debería ser capaz de encontrar la solución para un problema planteado en cualquier dominio. Para intentar alcanzar este objetivo, los investigadores de IA han desarrollado técnicas capaces de encontrar soluciones a problemas determinados en dominios específicos.

Una de las fuentes para la obtención de estas técnicas han sido las competiciones de juegos. Estos certámenes no son más que el intento de resolución de un problema en un dominio específico, el juego establecido en las bases del concurso. Gracias a estas competiciones ha sido posible desarrollar sistemas capaces de jugar mejor que los humanos, con el fin de obtener la victoria en dichos juegos, como por ejemplo en el othello (Buro, 1999).

Aunque existen bastantes técnicas no dependientes del dominio, sólo existe una pequeña variedad de situaciones donde puedan ser usadas de manera eficiente. Es importante tener estas técnicas cambiantes porque existen un gran número de situaciones reales, con escenarios cambiantes que no permiten tener un gran período de adaptación. Estas situaciones, como por ejemplo un rover en Marte, pueden modelarse como problemas o juegos, y es en ese momento cuando surge la necesidad de resolverlos. Debido a esta característica y a que nos estamos alejando del objetivo de la IA, surgió la necesidad de crear competiciones donde se pudieran obtener técnicas no dependientes de dominio. Una de estas competiciones es la *General Game Playing Competition* (GGP).

El objetivo de la GGP es crear un *sistema capaz de aceptar una descripción formal de cualquier juego y jugarlo de manera eficiente sin intervención humana* (Genesereth and Love, 2005). Como se puede ver un sistema GGP, a diferencia de Deep Blue u otros sistemas, no es específico a un dominio concreto. Gracias a esta competición, se podrán

tener técnicas que sirvan para cualquier situación ayudando a obtener los objetivos de la inteligencia artificial. Para representar estas reglas se ha desarrollado un lenguaje específico conocido como Game Definition Laguage (GDL) (Love et˜al., 2008) que intenta modelar juegos de información completa.

La importancia de la GGP respecto a otras competiciones es que al existir un lenguaje para definir los juegos, GDL, y unos procedimientos ya definidos para crear la competición y los servidores que gestionen la misma, puede crearse con facilidad competiciones locales donde poner a prueba nuevos agentes. Además, gracias al apoyo que tiene la competición, se desarrollado una base de datos con un gran número de juegos diferentes en lenguaje GDL.

Debido al carácter genérico de la competición, existen multitud de técnicas diferentes para poder crear un sistema capaz de participar en la GGP. No obstante, el enfoque clásico para participar en la GGP ha sido el uso de técnicas de búsqueda. La idea básica de estos algoritmos es partir del estado actual e ir evaluando las múltiples opciones que se tienen para poder elegir la mejor sucesión de operaciones para alcanzar el estado objetivo.

Por último se detallará la estructura de está memoria:

- Capítulo 2: En el *Estado de la cuestión*, se introducirá en mayor profundidad en la competición GGP, en los recursos disponibles y en el lenguaje GDL. Además se explicará el funcionamiento de la competición y la forma básica de un agente que compita en la misma. Después de esta parte, se listarán y explicarán, diferentes técnicas de búsqueda que serán usadas para comprender el funcionamiento del agente que se va a desarrollar. Por último, se indicarán los enfoques realizados para la creación de un agente para competir en la GGP.
- Capítulo 3: En este capítulo se listarán todos los *objetivos*, explicando el porqué del presente documento y las metas a alcanzar.
- Capítulo 4: En el apartado de desarrollo, primero se explicará el comportamiento del agente y su estructura. Además se indicarán los elementos que usa para su funcionamiento. Por otra lado, existirá una sección describiendo los problemas del lenguaje GDL a la hora de representar cualquier clase de juego, y como ha sido solventado. Gracias al uso de diagramas de clases, casos de uso y diagramas arquitectónicos se expondrá el completo funcionamiento de la aplicación. Por último, se aportan manuales de usuario y referencia para el uso e instalación del agente.

- Capítulo 5: Este apartado se centra en la sección de *pruebas* para comprobar la validez del proyecto dentro del marco de trabajo descrito en los objetivos. Además se detallarán los resultados obtenidos en la competición de GGP del año 2009.
- Capítulo 6: En las *conclusiones* se comprueba si este trabajo ha alcanzado los objetivos descritos anteriormente, y expondrá el estado final de proyecto.
- Capítulo 7: Las *líneas futuras* enlazarán el actual trabajo con posibles futuros proyectos que puedan partir de este trabajo o estén relacionados de algún modo con el mismo. Además dará ciertas pautas e ideas para realizar ampliaciones de este proyecto.

# Capítulo 2

# Estado de la cuestión

En esta sección se va a ofrecer una panorámica de la *General Game Playing* y la forma en que ha sido abordada. Los problemas planteados dentro del dominio de la General Game Playing, pueden ser tratados desde múltiples perspectivas de la inteligencia artificial. La primera sección del estado de la cuestión plantea la necesidad de obtener sistemas capaces de responder ante cualquier situación. La segunda sección define la competición de *General Game Playing* describiendo el lenguaje GDL (Game Description Language), las herramientas que se pueden usar para trabajar en la competición y algunos aspectos adicionales de la competición. La tercera sección mostrará el listado de algunas técnicas de búsqueda y generación heurística para resolver juegos. La última sección, presenta el trabajo que ha sido realizado hasta el momento en la GGP hasta la fecha y los diferentes enfoques que se han ido tomando.

## 2.1. Inteligencia General

Uno de los objetivos de la inteligencia artificial es la creación de un agente inteligente, es decir, la creación de un sistema que dada una situación determinada sepa responder correctamente. Multitud de competiciones se han realizado para llevar a cabo este objetivo. Una de las más prolíficas han sido las creadas para resolver juegos concretos. Como se comenta en el artículo sobre la solución del juego Checkers (Goossens et˜al., 2007) existen programas capaces de resolver juegos determinados mejor de lo que haría un ser humano, como el ajedrez (dee, 2006) o el othello (Buro, 1999).

Si bien, gracias a estas aportaciones se han conseguidos avances en el campo de la inteligencia artificial o el de sistemas distribuidos, siguen sin resolver uno de los problemas fundacionales de la IA (Inteligencia Artificial), la creación de un agente inteligente. Por

eso, surge el interés de crear una competición donde el objetivo no es crear un agente capaz de resolver un juego concreto adaptándose a su dominio, si no crear un agente capaz de resolver cualquier juego planteado, adaptándose al dominio planteado.

## 2.2. GGP: General Game Playing

Un sistema *General Game Playing* es aquel que puede soportar la definición formal de un juego (las reglas que lo definen) sin conocerlo previamente y jugarlo eficazmente sin intervención de un ser humano para ganarlo (Genesereth and Love, 2005).

### 2.2.1. Introducción a GGP

Las competiciones que usan dominios concretos para jugar tienen un valor limitado para la IA. Por ejemplo, competiciones para obtener un buen jugador de ajedrez han dado lugar a buenos agentes como Deep Blue pero esto luego no siempre se ha traducido en grandes descubrimientos en el campo de la inteligencia artificial. En cambio, en la competición de GGP al no estar supeditadas a un solo dominio y no saber a priori que dominio van a recibir, no pueden usar técnicas adaptadas a dominio. En contraste tendrán que usar técnicas para cualquier dominio o situación posible. Debido a esto, los agentes creados para competir en GGP pueden usar las técnicas que deseen, siendo luego estas técnicas fácilmente reutilizables en otros dominios dado su carácter general.

Los juegos están definidos en GDL, permitiendo definir cualquier juego de una manera formal (aunque con ciertas limitaciones intrínsecas al lenguaje). No obstante en una sección posterior hablaremos con más detalle sobre GDL.

Una vez definido el dominio del juego en GDL es necesario hablar de su infraestructura. Es necesario definir una unidad central que reciba los movimientos de los jugadores evaluando su legalidad, ejecutándolos sobre el estado actual del juego y usando las reglas definidas en el dominio para pasar al siguiente estado. Además indicará los ganadores y perdedores del juego y cómo estos han jugado. Esta unidad de juego es conocida como **GameMaster**. El GameMaster se compone de los siguientes elementos (Love et˜al., 2008):

- El **Arcade**: es una base de datos que contiene información sobre los juegos, jugadores y partidas. Es un buen punto de partida para comprender el lenguaje GDL ya que contiene un gran número de definiciones formales de juegos.
- El **Game Editor**: Usa la lista de jugadores y el arcade para crear y analizar juegos.

- **Game Manager**: Es el componente más importante del GameMaster ya que se encarga de ejecutar juegos, realizar turnos y de la comunicación con los jugadores.

Para poder jugar con el GameMaster es necesario realizar los siguientes pasos:

1. Disponer de un juego en el sistema.
2. Que uno o varios jugadores hayan introducido sus datos en el sistema.
3. **Tiempo de inicio**: Indica el tiempo que se le suministra a cada jugador antes de empezar a jugar para familiarizarse con el dominio.
4. **Tiempo de juego**: Es el tiempo que tiene cada jugador para pensar su movimiento.

Una vez iniciado el juego en el GameMaster, se establece una comunicación entre el GameMaster y los jugadores mediante mensajes HTTP que pueden ser de tipo PLAY, STOP o de tipo START. Los mensajes de tipo START son enviados a cada jugador con la información necesaria para comenzar el juego, mientras que los mensajes de tipo PLAY mandan el turno actual a cada jugador hasta que el juego termina, por lo que se envía un mensaje de tipo STOP. Entre el mensaje START y el primer mensaje PLAY pasa un tiempo en el que cada agente mejorara su conocimiento del dominio y perfeccionara la búsqueda sobre el mismo.

La clase de juegos suministrados en la GGP pueden ser para un solo jugador (laberintos, mundos de los bloques...), competitivos o colaborativos, juegos donde el espacio de búsqueda es totalmente explorable (tic-tac-toe) y otros donde no lo es (ajedrez), por citar unos pocos ejemplos.

### 2.2.2. GDL: Game Description Language

GDL (Game Description Language) es el lenguaje usado para definir dominios (hechos y reglas) de juegos en GGP. Es una variante de *Datalog*, siendo un lenguaje similar a Prolog. Gracias a este lenguaje se pueden describir la mayor parte de los dominios siendo las principales restricciones (Love et˜al., 2008) las siguientes:

- **Juegos deterministas**: Dado un estado y una acción, un jugador debe desplazarse siempre al mismo estado. Esto excluye cualquier representación del problema de un juego basado en probabilidades o lógica borrosa.
- **Información completa**: Todos los jugadores conocen con exactitud toda la información del juego, teniendo todos los jugadores exactamente la misma información.

Se supone que se puede extender con facilidad el lenguaje para albergar juegos de información incompleta, pero a día de hoy no se ha abordado este problema (Genesereth and Love, 2005).

- **Finitos**: El grafo que representa todos los posibles estados de los juegos debe ser finito. Esto se consigue teniendo un número limitado de operadores en cada nodo y un número límite de movimientos que un jugador puede realizar en un juego.
- **Fuertemente ganable**: Todo jugador debe tener la posibilidad de ganar al comienzo de un juego, es decir existe un camino desde el estado inicial a un estado final donde el valor para ese jugador es máximo.

Incluso con estas restricciones, GDL, puede ser lo suficientemente flexible como para permitir juegos en tiempo real, por turnos, cooperativos, un solo jugador, de suma no nula ... El lenguaje usa una variante del Knowledge Interchange Format (KIF) (basado a su vez en una variante de Datalog), y está basado en una serie de hechos que representan el estado actual del problema (usa la hipótesis del mundo cerrado) y reglas que permiten modificar estos hechos de manera determinista. Los hechos se componen del nombre del hecho seguido de un número indeterminado de constantes o variables. Una variable se define con un `?` , indicando que puede ser instanciado por cualquier elemento. Las relaciones pueden ir agrupadas en unas relaciones especiales con el símbolo `<=` llamadas reglas. Dependiendo de las relaciones que hayan dentro las reglas estas pueden servir para indicar si el operador usado es válido `(legal)`, reglas de operación `(does)`... Las relaciones que existen son las siguientes:

1. `role`: Describe a cada jugador que participa en el juego.

   ```
   (role player1)
   ```

2. `true`: Indica que en la relación actual el hecho que acompaña a la relación debe ser verdad. En el ejemplo indica que el hecho de controlar `x` es verdad.

   ```
   (true (control x))
   ```

3. `init`: Es una relación similar a la de `true`, pero con la diferencia de que indica el estado inicial del dominio.

   ```
   (init (cell 1 1 b))
   ```

4. `next`: Es una relación similar a la de `true`, pero indicando que será verdad en el siguiente estado.

```
(init (cell 1 1 b))
```

5. `legal`: Dada una acción indica las precondiciones necesarias para que pueda realizarse. La estructura de esta regla es: `<legal><rolJugador><accion>`.El rol de jugador indica a un jugador definido con anterioridad con la relación `role`. `Acción` indica la acción a llevar a cabo y las variables/constantes que usa. Un ejemplo de este operador es:

   ```
   (<= (legal ?player (mark ?x ?y))
   (true (cell ?x ?y b))
   (true (control ?player)))
   ```

   En este ejemplo, para realizar el movimiento en el mismo lugar `mark` debe existir el hecho `cell` y estar en blanco. Además para poder realizar el movimiento debe ser el turno del jugador.

6. `does`: Esta acción indica el movimiento realizado por un jugador, indicando las precondiciones y postcondiciones. Para poder ejecutar esta regla se deben ejecutar tanto las reglas de tipo `legal` como las precondiciones de esta regla y de todas las que sean de operador. El ejemplo siguiente indica que al marcar un casilla en el siguiente estado se pondrá una marca en ella.

   ```
   (<= (next (cell ?x ?y ?player))
   (does ?player (mark ?x ?y)))
   ```

   Todos los hechos dentro de la regla, encima de la relación `does` existirán el siguiente estado, y las precondiciones son aquellos hechos en la regla debajo de la relación `does`.

7. `goal`: Indica las condiciones que se deben dar para que un jugador tenga una puntuación determinada. En este ejemplo para que el jugador tenga una puntuación de 100 se debe hacer una línea.

   ```
   (<= (goal ?player 100)
   (line ?player))
   ```

8. `terminal`: El juego sabe que ha terminado con esta relación. Si existen todos los hechos o relaciones indicadas es un estado terminal. Por ejemplo, si se da una línea para un jugador determinado se acaba el juego.

   ```
   (<= terminal
   (role ?player)
   (line ?player))
   ```

Los hechos que definen un estado de un juego de GDL, solo están en ese estado hasta el siguiente turno, por lo que en cada turno las reglas de elección de movimientos `(does)` o las automáticas `(next)` deben definir completamente que hechos estarán en el siguiente turno.

### 2.2.3. Estructura agente GGP

Un agente de la GGP tiene al menos 4 elementos básicos (Kaiser, 2007), como podemos ver en el siguiente diagrama:

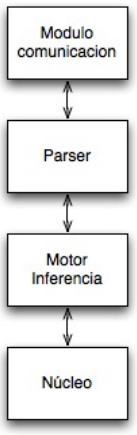


FIGURA 2.1: Agente GGP.

- **El módulo de comunicación**: Se encarga de recibir y enviar paquetes HTTP con los movimientos del agente, o de recibir la descripción de un nuevo dominio. Establece la comunicación con el GameMaster esperando la recepción del dominio hasta el final del juego.
- **Parser**: Este módulo transforma los paquetes HTTP recibidos, en estructuras que el agente puede usar. Este módulo es importante, ya que puede optimizar las búsquedas al mejorar la forma de transformar el dominio o movimientos recibidos. Tras recibir el mensaje de inicio de ronda, el agente convierte la información no relacionada directamente con las reglas a variables, por ejemplo los temporizadores. Las reglas del juego recibido, en cambio, son transformadas a una lista de hechos y reglas almacenando todos los símbolos en una tabla de símbolos.

  Las reglas del juego en formato GDL se transforman en una sucesión de elementos que más adelante usará el motor de inferencia. Los elementos más pequeños con significado GDL son los átomos o variables. Un átomo es una palabra, número o letra. Una variable puede representar cualquier átomo, relación o concepto. Debido

a esto, lo primero que se realiza tras obtener las reglas es convertir cada átomo en la **estructura átomo** y cada variable en la **estructura variable**. Cada una de estas estructuras almacena una referencia a la descripción de la variable/átomo que representa en la tabla de símbolos.

La tabla de símbolos es una *tabla hash* que se encarga de almacenar cada nombre de átomo, variable o relación y un número asignado de forma arbitraria. La ventaja de la tabla de símbolos es su rapidez a la hora de unificar (en el motor de inferencia) y de realizar operaciones sobre las reglas.

Las variables y los átomos por sí solos no tiene significado, pero si se juntan uno o varios de ellos se forma **una expresión GDL**. Una expresión GDL es un conjunto de elementos que tienen significado entre sí, pudiendo ser átomos, variables o listas de elementos. Una **lista** no es más que una sucesión de elementos. Para facilitar la unificación y ahorrar posteriores comprobación, cada expresión indica si contiene o no varias variables. Por lo tanto tras conseguir identificar las expresiones más básicas, estas a su vez se van agrupando en otras expresiones, hasta que al final toda la descripción del juego en GDL sea una única expresión.

Las relaciones, hechos o acciones existentes en las reglas no se identifican hasta no instanciar el motor de inferencia. De esta forma se puede ampliar el agente con distintos motores de inferencia sin necesidad de modificar el parseo de los mensajes entrantes/salientes.

- **Motor de inferencia**: Dadas las estructuras provistas por el *parser*, es necesario saber cual es el estado siguiente dado un estado actual y un operador. El motor de inferencia se usa para expandir nodos del árbol de juego y poder hacer simulaciones del mismo. Esta parte también es optimizable, ya sea ofreciendo una mayor velocidad en la generación de un estado a partir de otro o un mejor acceso más simple a la interfaz del motor de inferencia. Algunos motores de inferencia que se han usado son JTP (Java Theorem Prover), JESS (Java Expert System Shell) o incluso se han traducido las reglas a Prolog.

- **Núcleo del agente/Motor de búsqueda**: El núcleo de cualquier agente de la GGP. Se encarga de ejecutar el/los algoritmos de búsqueda necesarios para encontrar la solución. Este núcleo es importante porque es donde recae la función de búsqueda del agente.

Además de estos módulos, suele existir uno que analiza el juego durante el tiempo desde que se recibe el dominio del juego hasta que empieza la partida, encargado de analizar toda la información del juego y mejorar la función heurística. Obviamente un agente se

puede construir de cualquier forma, pero todos los agentes creados hasta la fechan siguen estructura similar a la anteriormente expuesta.

### 2.2.4. Recursos disponibles

La universidad de Stanford [1] ofrece software para poder establecer la comunicación entre el agente y el GameMaster y crear un agente. Los dos principales códigos para crear un agente en la GGP son:

- **Common Lisp Reference Player**: Es código **Lisp** con las funciones necesarias para comunicarse con el GameMaster, computar movimientos correctos y cargar agentes. No existe documentación adicional al código.
- **Jocular**: Realizado en Java, permite crear agentes que además interactúan con el GameMaster. Tiene un javadoc realizado y en un futuro se pretende realizar un manual del mismo. Además existe un repositorio de ficheros *kif* con descripciones de juegos [2]. Estos juegos no tienen porque ser los mismos que se usen en la competición final, aunque muchos de ellos serán similares. Para probar estas descripciones de juegos o realizar nuevas descripciones de los mismos, existe un plugin del IDE Eclipse para editar, simular y depurar ficheros con la descripción de juegos [3].

La universidad de Dresden tiene un simulador parecido al anterior [4] para ejecutar y probar los agentes creados. Además existe una versión del simulador de escritorio que simula jugar contra varios oponentes, [5].

## 2.3. Técnicas de búsqueda

Existe una gran variedad de técnicas para abordar problemas de búsqueda en IA. Una forma habitual de resolver juegos es la búsqueda heurística (Pearl, 1984). Un valor heurístico, es una función que provee información referente a como de bueno es un estado. Otra forma puede ser usar simulaciones para estimar los valores de los estados, como es el caso del método de Monte Carlo (H.Hoos and Stützle, 2005).

Al usar las técnicas de búsqueda usaremos una representación de **árbol**. Si bien era posible usar un grafo, hemos decidido usar la forma de representación de árbol para mejorar las explicaciones de cada algoritmo.

[1] http://games.stanford.edu/
[2] http://www.general-game-playing.de/game_db/doku.php
[3] http://palamedes-ide.sourceforge.net/
[4] http://euklid.inf.tu-dresden.de:8180/ggpserver/
[5] http://www.general-game-playing.de/downloads.html

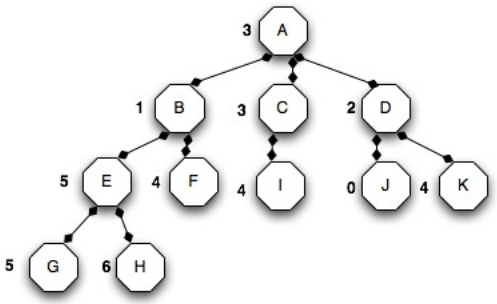


FIGURA 2.2: Ejemplo de árbol de búsqueda con valores heurísticos.

Cada nodo del árbol representa un estado del juego, siendo un estado una *foto fija* de la situación del problema en ese momento. Desde un estado se puede ir a otro mediante operadores, representados como flechas entre nodos. En este momento es importante distinguir dos conceptos. El **coste** de una operación, es la cantidad de esfuerzo conocido para ejecutar un operador determinado, siendo normalmente este valor de 1. El valor **heurístico** es el coste estimado que hay entre un estado y un estado meta. La suma de ambas funciones da el valor de evaluación de dicho nodo, que normalmente se intenta minimizar. Al hablar de la función heurística es importante señalar:

- **Admisibilidad**: Una función heurística es admisible si nunca sobreestima los costes de la función objetivo.
- **Óptima**: Una función heurística es óptima si la información heurística que ofrece es igual a la real.

### 2.3.1. A*

A* es un algoritmo mejor primero donde la función de evaluación $f(n)$ es la suma de la función de coste $g(n)$ y la función de heurística $h(n)$. El algoritmo termina cuando expande un nodo terminal o no quedan más nodos. El algoritmo A* es óptimo si la función heurística $h(n)$ es admisible, es decir que la función heurística nunca **sobreestime** el coste de llegar a la meta (Russel and Norving, 1995). Además este algoritmo es completo si la solución existe. En un problema finito, aquel problema en el que el árbol de búsqueda tiene un número finito de nodos, el algoritmo A* encontrará la solución o expandirá todo el árbol sino existe solución y en uno infinito, siempre y cuando las heurísticas y costes sean positivos y admisibles, encontrará la solución.

Uno de los mayores problemas que tiene este algoritmo es el crecimiento exponencial en memoria debido a que se deben almacenar en memoria los nodos sucesores de cada

estado. Para obtener una mayor comprensión del algoritmo A* se muestra a continuación un ejemplo de su uso:

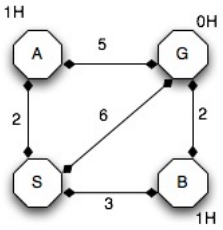


FIGURA 2.3: Paso 1º del algoritmo A*.

El algoritmo 87 tiene una lista cerrada y otra abierta. Para este ejemplo se ha usado un ejemplo sencillo de 4 nodos conectados entre sí. En la imagen indicamos con un número al lado de cada nodo su valor heurístico. Es un grafo no dirigido con nodo inicial S y nodo final G.

$$Paso\#1 : \begin{cases} Open : \{S\} \\ Close : \{\} \end{cases}$$

En este paso la lista abierta contiene el nodo inicial S y la lista cerrada esta vacía. Entonces se decide expandir el único nodo de la lista abierta, en este caso el nodo S.

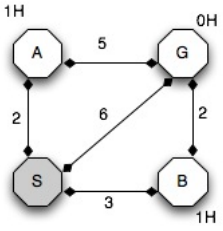


FIGURA 2.4: Paso 2º del algoritmo A*.

Tras expandir el nodo S, este pasa de la lista abierta a la lista cerrada y a continuación se indican los contenidos de ambas listas en el segundo paso y se indica entre paréntesis el valor $f(n)$ de cada nodo:

$$Paso\#2 : \begin{cases} Open : \{A(3), B(4), G(6)\} \\ Close : \{S\} \end{cases}$$

Entonces se decide expandir el nodo con menor función de evaluación. En este caso el nodo A, que pasaría a la lista cerrada añadiendo todos sus sucesores. Nótese que en la

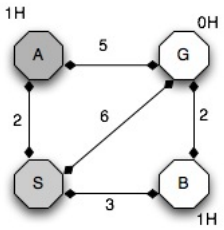


FIGURA 2.5: Paso 3º del algoritmo A*.

lista abierta puede entrar varias veces el mismo nodo si a éste se puede llegar desde varios caminos.

$$Paso\#3 : \begin{cases} Open : \{B(4), G(6), G(7)\} \\ Close : \{S, A\} \end{cases}$$

Tras expandir el nodo A, se expande el nodo con menor valor de evaluación, el nodo B.

$$Paso\#4 : \begin{cases} Open : \{G(5), G(6), G(7)\} \\ Close : \{S, A, B\} \end{cases}$$

En este paso se debería expandir el nodo G ya que es el de menor valor de evaluación. El algoritmo acaba en este paso ya que va a expandir el nodo final o meta G.

En el apéndice A.1 en la página 87, esta detallado el pseudocódigo del algoritmo A*.

### 2.3.2. RTA*

Uno de los problemas del algoritmo A* es que no ofrece la solución dentro de un tiempo determinado. Además el consumo de memoria es demasiado alto, como para poder ser usado en la práctica. Para evitar estos problemas se propone usar el algoritmo RTA* (Korf, 2007). La idea de esta algoritmo es explorar el árbol de búsqueda y en caso de encontrar que el camino elegido implica demasiado coste, volver hacía atras para tomar un camino anteriormente descartado. Una implementación de este algoritmo es detallada en la sección A.2 de la página 88

#### 2.3.2.1. Minimin

Si se especializa el algoritmo minimax para búsqueda de un solo agente, obtenemos el algoritmo minimin (Korf, 2007). El algoritmo explora hasta una determinada profundidad

y devuelve la función de evaluación $f(n) = g(n)+h(n)$, retornando el menor valor $f(n)$ de la frontera. Si se encuentra un valor meta, se termina el algoritmo. Si bien este algoritmo puede ser mejorado usando *branch and bound* (Balas, 1968), el valor de todos los hijos es necesario para realizar el algoritmo RTA*.

#### 2.3.2.2. Algoritmo RTA*

En RTA*, $g(n)$ es la distancia que hay entre el nodo $n$ y el nodo actual desde donde se realiza la búsqueda. En esto estriba la diferencia entre RTA* y A*. A RTA* sólo le interesa la posición relativa entre el nodo actual y el nodo analizado. Para cada paso del algoritmo, se expanden los nodos del estado raíz y se genera la función de evaluación de cada estado sucesor. Entonces se usa como nuevo estado raíz el estado sucesor con menor función de evaluación y la función heurística del estado raíz es la función de evaluación del segundo menor estado sucesor. De esta forma sabemos si merece la pena volver hacia atrás o no.

Una de las ventajas principales de usar este algoritmo frente a A* es que RTA* sólo tiene que almacenar en memoria la función heurística de los estados raíz visitados, luego el crecimiento en memoria sólo depende de cada paso realizado.

#### 2.3.2.3. Ejemplo RTA*

Para este ejemplo usaremos la figura 2.2 de la página 13. El algoritmo no acaba hasta no encontrar la solución. El objetivo de este ejemplo es llegar al nodo J y demostrar que funciona el algoritmo RTA* incluso cuando la función heurística es no admisible.

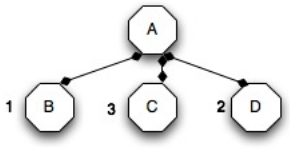


FIGURA 2.6: Paso 1º del algoritmo RTA*.

El primer paso consiste en realizar un subárbol con el nodo inicial, en nuestro caso el nodo A, y obtener las funciones de evaluación de cada nodo.

$$\left.\begin{array}{l} \mathbf{f(B) = g(B) + h(B) = 1 + 1 = 2} \\ f(C) = g(C) + h(C) = 1 + 3 = 4 \\ f(D) = g(D) + h(D) = 1 + 2 = 3 \end{array}\right\} = h(A) = f(D) = 3$$

Se toma como nuevo valor heurístico del subárbol el *segundo* mejor valor de función de evaluación de un sucesor del nodo raíz, en nuestro caso el nodo D. Y como nuevo árbol de búsqueda se toma aquel nodo con mejor valor de evaluación, en este ejemplo el nodo B. Tras esto se realiza el siguiente paso con el nuevo subárbol de búsqueda.

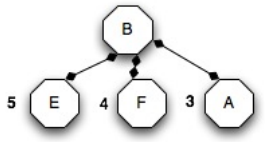


FIGURA 2.7: Paso 2º del algoritmo RTA*.

$$\left.\begin{array}{l} f(E) = g(E) + h(E) = 1 + 5 = 6 \\ f(F) = g(F) + h(F) = 1 + 4 = 5 \\ \mathbf{f(A) = g(A) + h(A) = 1 + 3 = 4} \end{array}\right\} = h(B) = f(F) = 5$$

En este paso podemos ver que el algoritmo considera mejor retroceder que seguir explorando este árbol. La representación de retroceder se hace usando como nueva función heurística de un nodo, la función de evaluación del segundo mejor nodo. De esa forma si nos vemos que el camino actual no es una buena solución para llegar al objetivo, como este caso, podemos evaluar la posibilidad de volver hacia atrás.

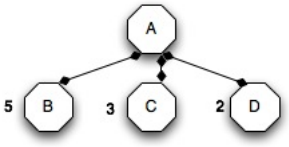


FIGURA 2.8: Paso 3º del algoritmo RTA*.

$$\left.\begin{array}{l} f(B) = g(B) + h(B) = 1 + 5 = 6 \\ f(C) = g(C) + h(C) = 1 + 3 = 4 \\ \mathbf{f(D) = g(D) + h(D) = 1 + 2 = 3} \end{array}\right\} = h(A) = f(C) = 4$$

En el 3º paso podemos ver como el algoritmo decide explorar otro nodo, en vez de volver a retroceder e ir al nodo B.

$$\begin{array}{l} \mathbf{f(J) = g(J) + h(J) = 1 + 0 = 1} \\ f(K) = g(K) + h(K) = 1 + 4 = 5 \\ f(A) = g(A) + h(A) = 1 + 4 = 5 \end{array}$$

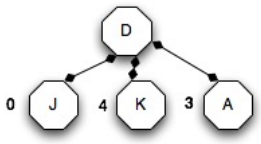


FIGURA 2.9: Paso 4º del algoritmo RTA*.

En este último paso, podemos ver como la mejor opción desde el nodo D, era ir a un nodo terminal por lo que en principio sería el nodo más cercano al nodo raíz. Para saber el camino para llegar a este nodo desde la raíz, basta con ir retrocediendo nodos raíz hasta llegar al nodo raíz del árbol. En nuestro caso sería: J ->D ->A. Como se puede ver este algoritmo toma decisiones locales óptimas almacenándose como nuevo valor heurístico de un nodo, el segundo mejor valor de evaluación de dicho nodo. Si en vez del segundo mejor valor, hablamos del mejor valor, nos referimos al algoritmo LRTA*.

### 2.3.3. LRTA*

Existen ciertas situaciones donde el algoritmo RTA* puede quedarse atrapado. Por ello surgió una variante que no toma el segundo mejor valor de los hijos, sino que *aprende* el mejor valor. Learning-RTA* o LRTA* (Korf, 2007) es un algoritmo completo (como el RTA*), que no siempre toma las decisiones óptimas locales. La idea del algoritmo consiste en almacenar en cada iteración el mejor valor para cada nodo, en lugar del segundo mejor como el RTA*.

La idea del algoritmo es ir explorando las posibles alternativas que existen para un nodo determinado e ir a aquella con menor valor de función de evaluación. El valor heurístico de este nodo entonces, se almacenaría usando el valor del estado sucesor con menor valor de función de evaluación. Tras esto se usa ese estado como raíz realizando el proceso anterior de manera análoga. El pseudocódigo del algoritmo viene en el apéndice A.3 en la página 89.

Según Korf, el algoritmo LRTA* converge a la solución óptima en un problema finito de costes positivos, con estados metas alcanzables, desde cada meta y con valor heurístico no sobreestimado. Con RTA* no es necesaria que la función heurística sea admisible, aunque, cuanto más precisa sea ésta, antes se encontrará la solución.

En un problema finito, con costes y funciones de evaluación finitas, donde desde cualquier estado se puede alcanzar la solución, RTA* encontrará la solución. El factor que más influye en RTA* es la función heurística. Por eso a más grande sea el horizonte de búsqueda (número de niveles visitados), mejor será la solución encontrada.

Para el ejemplo de algoritmo LRTA* usaremos el árbol anteriormente descrito en la página 2.2 de la página 13. A diferencia del RTA*, el algoritmo LRTA* almacena la que tenga menor función de evaluación, usando el algoritmo minimin.

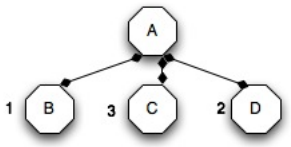


FIGURA 2.10: Paso 1º del algoritmo LRTA*.

El primer paso consiste en realizar un subárbol con el nodo inicial, en nuestro caso el nodo **A**, y obtener las funciones de evaluación de cada nodo.

$$\left.\begin{array}{l} \mathbf{f(B) = g(B) + h(B) = 1 + 1 = 2} \\ f(C) = g(C) + h(C) = 1 + 3 = 4 \\ f(D) = g(D) + h(D) = 1 + 2 = 3 \end{array}\right\} = h(A) = f(B) = 2$$

Se toma como nuevo valor heurístico del nodo del subárbol el mejor valor de función de evaluación en vez del segundo mejor como en el algoritmo RTA*, en este caso el nuevo valor heurístico de A es el valor de evaluación de B. Y como nuevo árbol de búsqueda se toma aquel nodo con mejor valor de evaluación, en este ejemplo el valor del nodo B. Tras esto se realiza el siguiente paso con un nuevo subárbol de búsqueda.

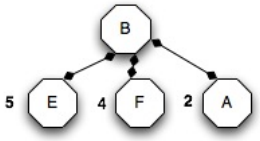


FIGURA 2.11: Paso 2º del algoritmo LRTA*.

$$\left.\begin{array}{l} f(E) = g(E) + h(E) = 1 + 5 = 6 \\ f(F) = g(F) + h(F) = 1 + 4 = 5 \\ \mathbf{f(A) = g(A) + h(A) = 1 + 2 = 3} \end{array}\right\} = h(B) = f(A) = 3$$

En este estado el algoritmo decide no seguir explorando este nodo ya que el nodo anterior tiene mejor función de evaluación, y vuelve entonces a expandir el nodo A.

$$\left.\begin{array}{l} f(B) = g(B) + h(B) = 1 + 3 = 4 \\ f(C) = g(C) + h(C) = 1 + 3 = 4 \\ \mathbf{f(D) = g(D) + h(D) = 1 + 2 = 3} \end{array}\right\} = h(A) = f(D) = 3$$

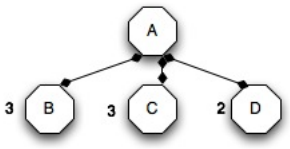


FIGURA 2.12: Paso 3º del algoritmo LRTA*.

En el 3º paso podemos ver como el algoritmo decide explorar otro nodo, en vez de volver a retroceder e ir al nodo B.

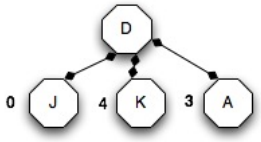


FIGURA 2.13: Paso 4º del algoritmo LRTA*.

$$\mathbf{f(J) = g(J) + h(J) = 1 + 0 = 1}$$
$$f(K) = g(K) + h(K) = 1 + 4 = 5$$
$$f(A) = g(A) + h(A) = 1 + 3 = 4$$

En este punto se para el algoritmo de búsqueda al llegar al nodo terminal J con el valor mínimo.

### 2.3.4. MiniMax

Hasta este momento todos los algoritmos descritos intentan siempre minimizar la función de evaluación, es decir toman aquellos caminos mejores para llegar a la solución. Uno de los problemas que existen, son las situaciones multiagentes. Tras realizar un movimiento, el siguiente movimiento es realizado por otro jugador. El movimiento del otro jugador no suele llevarnos al estado objetivo. De estas ideas surgió el algoritmo **minimax** para juegos con adversario e información completa. Si bien yo, como jugador intentaré maximizar mi utilidad de realizar el movimiento, el contrario intentará minimizar la utilidad eligiendo el peor movimiento para mí.

La idea básica de este algoritmo es ir expandiendo un árbol de búsqueda e ir alternando los niveles del árbol entre los jugadores. En los nodos *max*, el jugador mueve intentando maximizar la función de utilidad, mientras que en los nodos *min*, el adversario intentará minimizar la función de utilidad. En el ejemplo anterior, el nodo superior es un nodo

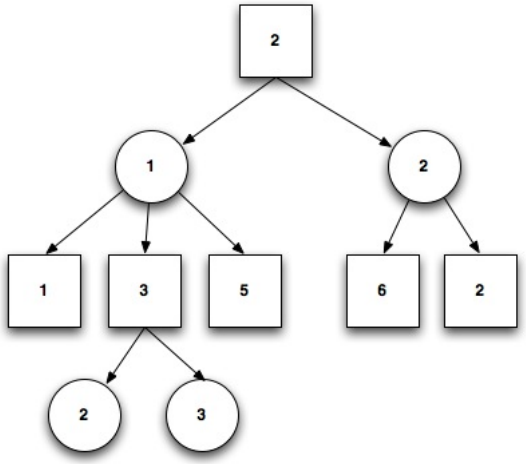


FIGURA 2.14: Un árbol minimax.

*max* y el nivel siguiente son nodos *min*. Ser un nodo *max* significa que tomará el mayor valor de los nodos sucesores y *min* que tomará el menor de sus sucesores. Si el nodo es terminal se devuelve el valor de su función de evaluación.

El algoritmo minimax , explicado en el apéndice A.4 de la página 90 sólo funciona en juegos para dos jugadores, finitos y de suma nula.

En el siguiente ejemplo tomaremos como nodo *max* los cuadrados y como nodo *min* los círculos. En el estado actual al no ser un nodo terminal desconocemos su valor por lo que usaremos la función de evaluación que nos dará el valor de **5**. Tras esto se expandirá el nodo actual dando dos sucesores, tomando como valor el nodo *max*, el valor del nodo sucesor con más valor, 7. Si seguimos expandiendo los nodos, en este caso el nodo con valor 3, nos encontramos con dos nodos terminales y uno no terminal. Los nodos terminales toman el valor de su estado, y al ser el nodo padre un nodo min, este tomará el valor del nodo sucesor con menor valor 1. Este algoritmo se repetirá hasta que todo el árbol esté expandido o hasta que haya alcanzado una profundidad máxima prefijada de antemano.

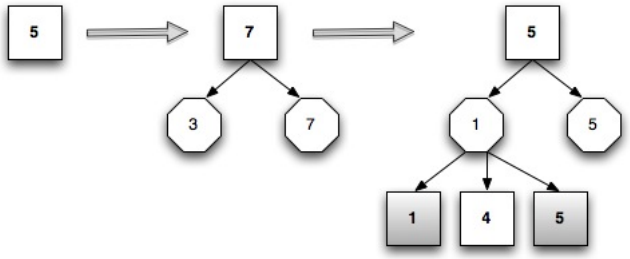


FIGURA 2.15: Ejemplo del algoritmo minimax.

Existen multitud de variaciones a este algoritmo y de mejoras, ya sea cambiando la forma de expandir los nodos (por niveles en vez de izquierda a derecha) o mejoras sobre la función heurística.

### 2.3.5. Algoritmo Alfa-Beta

Uno de los problemas del algoritmo minimax es que para obtener el valor de un nodo es necesario comprobar todos sus nodos sucesores y si el *factor de ramificación* es alto, el coste computacional sería muy grande. Para esto surgió el algoritmo Alfa-Beta, sobre el cual subyace la idea de que solo es necesario buscar en los sucesores de un nodo si existe la posibilidad de que haya un valor mejor.

El algoritmo alfa-beta, mostrado en la sección A.5 de la página 91 toma dos valores $\alpha$ y $\beta$, que toman valores de $-\infty$ y $+\infty$. Si se llega a un nodo terminal se toma como valor $\alpha$ y $\beta$ el valor de la función de evaluación. En caso de no encontrar un nodo terminal se toma como valor $\alpha$ el mayor valor de *función heurística* de los hijos y como valor $\beta$ el menor. No se explora más en caso de que ($\alpha >= \beta$), devolviendo el valor actual. Este algoritmo realiza llamadas recursivas hasta que se tiene el valor del nodo inicial, pero gracias a la función de poda no tiene que explorar todos los sucesores.

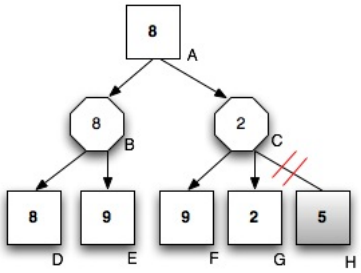


FIGURA 2.16: Ejemplo del algoritmo alfa beta.

En el ejemplo anterior el algoritmo va evaluando todos los nodos hasta llegar al nodo G. El nodo A es un nodo *max* y en el momento anterior a evaluar el nodo H tiene el valor de sus mayores sucesores, 8. Para que tome el valor del nodo C este debe tener un valor mayor que el nodo B, pero el nodo B es un nodo *min*. Eso significa que si el nodo C quiere tener un valor mayor que 8 , todos sus sucesores deben tener un valor mayor que 8, cosa que no ocurre con el nodo G. Por esto ya no es necesario evaluar más nodos sucesores del nodo C, ya que cualquier nodo sucesor del nodo C, solo puede darle un valor menor o igual a dicho nodo.

### 2.3.6. CN-Search

Un método que surgió a partir de la búsqueda del minimax son los **números conspiratorios**. A diferencia del algoritmo minimax, cn-search no indica un movimiento a realizar, sino que indica la estabilidad de un nodo determinado.

#### 2.3.6.1. Definición del algoritmo de números conspiratorios

La idea principal de este algoritmo de búsqueda es mostrar la dificultad de cambiar el valor minimax de un nodo determinado (McAllester, 1988). Para hacer esto se define como **número conspiratorio** (CN) al mínimo número de nodos terminales que deben cambiar para que cambie su valor. Se dicen que estos nodos *conspiran* para cambiar el valor del nodo raíz. El algoritmo usa un *umbral*, para saber cuando parar la búsqueda y como medida para saber como de buena es la solución. Esto se debe a que el algoritmo de números conspiratorios, no garantiza encontrar la solución correcta, sino la mejor dado el umbral. A mayor sea el umbral (número de nodos necesarios para cambiar), más grande es la probabilidad de que el movimiento elegido sea correcto.

#### 2.3.6.2. Explicación del algoritmo de números conspiratorios

El algoritmo A.6 de la página 92 se basa en realizar llamadas recursivas desde la raíz a los nodos hoja. Un nodo hoja tendrá un número conspiratorio de 0 si su valor minimax $m$ es igual al valor $v$ estimado, pero 1 si debe cambiar para tener el valor pedido. En caso de ser un nodo terminal, al ser su valor fijo, *no depende de una función heurística*, es imposible que cambie su valor y el valor de número conspiratorio sería $+\infty$.

$$CN(v) = \begin{cases} 0 & \text{si } v = m \\ 1 & \text{si} v \neq 1 \\ +\infty & \text{si nodo terminal} \end{cases}$$

El algoritmo de números conspiratorios usa los valores minimax para calcular los valores en cada nodo. Dependiendo si es un nodo *min* o un nodo *max* se usarán unos valores u otros. Para cada nodo, interior, se genera una lista de valores $v$ que se usan para realizar el algoritmo de números conspiratorios. Además cada nodo interior tiene dos listas complementarias de números conspiratorios:

- $\uparrow$ **CN**: Valores $v$ mayores que el valor minimax.

- $\downarrow$ **CN**: Valores $v$ menores que el valor minimax.

Para el resto de valores ambos conjuntos son vacíos, y su unión genera el conjunto de números conspiratorios para un nodo interior. Para calcular el valor de números conspiratorios en un nodo *min* y *max* serían respectivamente:

$$CN(v)_{max} = \begin{cases} \sum^{i}_{\forall sucesores} \downarrow CN_i(v) & \text{si } v < m \\ min^{i}_{\forall sucesores} \uparrow CN_i(v) & \text{si } v > m \end{cases}$$

$$CN(v)_{min} = \begin{cases} min^{i}_{\forall sucesores} \downarrow CN_i(v) & \text{si } v < m \\ \sum^{i}_{\forall sucesores} \uparrow CN_i(v) & \text{si } v > m \end{cases}$$

El sumatorio de los hijos de un nodo es la suma del número conspiratorio de cada hijo de ese nodo para el valor determinado. Mientras que el mínimo, es el menor valor de nodo conspiratorio que tiene un hijo.

Cuando se tienen los valores de números conspiratorios de la raíz y no existe ningún valor que sobrepase el umbral, el algoritmo ha finalizado. En ese momento se aumenta el umbral o se toma como acción aquella que sea más estable (necesite menos nodos para cambiar). En caso de que no se sobrepase el umbral se eligen el valor máximo/mínimo del nodo raíz y se elige la hoja correcta, expandiéndola y realizando otra vez el proceso de cálculo. El algoritmo puede iniciarse en umbral de 2 e incrementarlo iterativamente.

### 2.3.7. Ejemplo del algoritmo de números conspiratorios

El algoritmo de números conspiratorios se basa en el algoritmo minimax. En cada nodo se calculará el número conspiratorio para un valor determinado. Obviamente al haber un número infinito de posibles valores a los que cambiar el nodo raíz se toma un dominio reducido de éste, en este caso del 1 al 6.

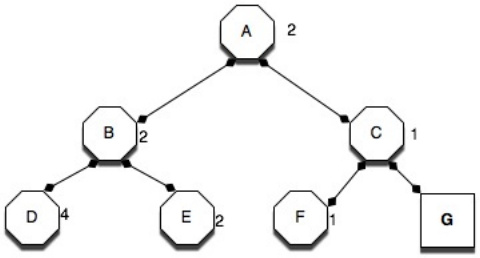


FIGURA 2.17: Árbol minimax para CN-Search.

En el ejemplo podemos ver como tenemos un árbol minimax, expandido a profundidad 2 y con el valor de cada nodo, el valor de la función de evaluación salvo en el nodo G que es un nodo terminal. Los nodos hoja tendrán para todos los valores un número conspiratorio (a partir de ahora cn-number) 1 excepto en el caso de que sea igual a su valor minimax que será 0.

$$CN(D) = \{1, 1, 1, 0, 1, 1\}$$

$$CN(E) = \{1, 0, 1, 1, 1, 1\}$$

$$CN(F) = \{0, 1, 1, 1, 1, 1\}$$

En el nodo G al ser terminal el valor del nodo nunca podrá cambiar, luego en caso de que el valor minimax y el valor a obtener sea diferente tendrá un cn-number de infinito.

$$CN(G) = \{+\infty, +\infty, 0, +\infty, +\infty, +\infty\}$$

Al llegar al nivel 1, los nodos son min luego habría que aplicar las ecuaciones vistas anteriormente para un nodo min. Esas ecuaciones necesitan de los conjuntos $\uparrow$ **CN** y $\downarrow$ **CN** de cada nodo hijo. Un ejemplo de esos conjuntos con el nodo D sería:

$$CN(D) = \{1, 1, 1, 0, 1, 1\}$$

$$\downarrow CN(D) = \{1, 1, 1, 0, 0, 0\}$$

$$\uparrow CN(D) = \{0, 0, 0, 0, 1, 1\}$$

Como se puede ver ambos conjuntos son complementarios y son los valores por encima o por debajo del valor minimax del nodo, por lo que solo sería necesario almacenar ese valor minimax y el valor de todos sus cn-number. El resultado sería el siguiente para el nodo B.

$$CN(B) = \begin{pmatrix} cn_1 \\ cn_2 \\ cn_3 \\ cn_4 \\ cn_5 \\ cn_6 \end{pmatrix} = \begin{pmatrix} min\{\downarrow CN(D)_1, \downarrow CN(E)_1\} \\ 0 \\ \sum\{\uparrow CN(D)_3, \uparrow CN(E)_3\} \\ \sum\{\uparrow CN(D)_4, \uparrow CN(E)_4\} \\ \sum\{\uparrow CN(D)_5, \uparrow CN(E)_5\} \\ \sum\{\uparrow CN(D)_6, \uparrow CN(E)_6\} \end{pmatrix} = \begin{pmatrix} min\{1, 1\} \\ 0 \\ \sum\{0, 1\} \\ \sum\{0, 1\} \\ \sum\{1, 1\} \\ \sum\{1, 1\} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 2 \\ 2 \end{pmatrix}$$

Y de forma análoga para el nodo C.

$$CN(C) = \begin{pmatrix} cn_1 \\ cn_2 \\ cn_3 \\ cn_4 \\ cn_5 \\ cn_6 \end{pmatrix} = \begin{pmatrix} 0 \\ \sum\{\uparrow CN(F)_2, \uparrow CN(G)_2\} \\ \sum\{\uparrow CN(F)_3, \uparrow CN(G)_3\} \\ \sum\{\uparrow CN(F)_4, \uparrow CN(G)_4\} \\ \sum\{\uparrow CN(F)_5, \uparrow CN(G)_5\} \\ \sum\{\uparrow CN(F)_6, \uparrow CN(G)_6\} \end{pmatrix} = \begin{pmatrix} 0 \\ \sum\{1,0\} \\ \sum\{1,0\} \\ \sum\{1,+\infty\} \\ \sum\{1,+\infty\} \\ \sum\{1,+\infty\} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ +\infty \\ +\infty \\ +\infty \end{pmatrix}$$

En el nodo raíz la situación es inversa al usarse la ecuaciones de un nodo max.

$$CN(A) = \begin{pmatrix} cn_1 \\ cn_2 \\ cn_3 \\ cn_4 \\ cn_5 \\ cn_6 \end{pmatrix} = \begin{pmatrix} \sum\{\downarrow CN(B)_1, \downarrow CN(C)_1\} \\ 0 \\ min\{\uparrow CN(B)_3, \uparrow CN(C)_3\} \\ min\{\uparrow CN(B)_4, \uparrow CN(C)_4\} \\ min\{\uparrow CN(B)_5, \uparrow CN(C)_5\} \\ min\{\uparrow CN(B)_6, \uparrow CN(C)_6\} \end{pmatrix} = \begin{pmatrix} \sum\{1,0\} \\ 0 \\ min\{1,1\} \\ min\{1,+\infty\} \\ min\{2,+\infty\} \\ min\{2,+\infty\} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 2 \\ 2 \end{pmatrix}$$

Tras tener los valores de números conspiratorios de la raíz se comprueban con el valor del umbral. Si se consideran que los valores son los suficientemente estables se para el algoritmo. También puede aumentarse el umbral y seguir con el algoritmo ya que si no se sobrepasa el umbral se expande un nodo hoja reiniciando el árbol minimax y recalculando todo el árbol de números conspiratorios. Los números conspiratorios nos dan la probabilidad de cambiar un nodo por lo que una opción para usarlos puede ser tomar aquellas rutas que sea más probable que nos den los valores esperados. Por ejemplo si nos interesa el valor 5 nuestro próximo movimiento será aquel que nos lleve a un nodo con menor valor heurístico, en este caso el nodo B.

#### 2.3.7.1. Mejoras sobre el algoritmo de números conspiratorios

El algoritmo de números conspiratorios recoge las probabilidad de realizar un movimiento u otro, pero a diferencia de otros algoritmos de búsqueda no tiene porqué converger (Klingbeil and Schaeffer, 1990). Esto se debe a que el algoritmo realiza búsqueda en profundidad sin la certeza de que esa rama sea la correcta. Para evitar que sólo se investigue un conjunto de los números conspiratorios se puede seleccionar el nodo a explorar mediante diferentes técnicas: round-robin, nodos AND/OR ...

Existen varias formas de implementar este algoritmo en función a la información almacenada:

- **Modo básico**: Usa solo el almacenamiento mínimo y necesario para cada nodo. La ventaja de este modo es su bajo consumo de memoria, pero es necesario calcular los números conspiratorios de cada nodo.
- **Almacenar toda la información**: Almacena los números conspiratorios en cada nodo, y estos sólo se actualizarán cada vez que se expanda un nodo y esté en la ruta de búsqueda. Este modo funciona más rápido que el modo básico, ya que no necesita calcular cada vez los números conspiratorios de cada nodo. Una mejora de esta aproximación consiste en almacenar los límites de los números conspiratorios en vez de almacenar todos los valores.
- **Solución de compromiso**: Almacena solo los valores más importantes en cada nodo: *Vmin*, *Vmin - 1*, *Vmax*.

### 2.3.8. Mejoras sobre los algoritmo de búsqueda

Los algoritmos de búsqueda basados en el minimax o en búsquedas alfa-beta tienen una gran variedad de mejoras disponibles, pero se basan en reducir el tamaño de los árboles de búsqueda y evaluar menos nodos. Las mejoras sugeridas suelen referirse a reordenar los nodos, cambiar el tamaño de la ventana de búsqueda y rehusar información. A continuación se enumeran algunas de las mejoras posibles (Schaeffer, 1989).

- **Profundidad iterativa**: Esta técnica va aumentando de forma iterativa la profundidad de búsqueda, de manera que se aumente la probabilidad de encontrar el mejor movimiento en la raíz.
- **Tablas de transposición**: En un árbol de búsqueda es posible llegar a la misma posición por distintos caminos, siendo posible almacenar esta información para evitar volver a evaluar el mismo subárbol. La información almacenada es, el valor del árbol, el movimiento que llevó al árbol y la profundidad del mismo.
- **Tablas de rechazo**: Reducen el tamaño de las tablas de transposición almacenando menos información que las tablas de transposición.
- **Ventana de aspiración**: Esta búsqueda modifica los valores iniciales de algoritmo alfa-beta haciéndolos más restrictivos. En caso de funcionar encuentra la solución antes y si no, se necesita volver a realizar la búsqueda con una ventana de búsqueda mayor.
- **Heurística asesina**: Durante una búsqueda la mayor parte de los movimientos son rechazados por el mismo movimiento. Usar este movimiento para reducir el tamaño del árbol de búsqueda puede funcionar aunque no siempre es así.

Si bien lo mejor es usar una combinación de estas mejoras dependiendo de la situación, la experimentación ha demostrado que es mejor usar tablas de transposición(Schaeffer, 1989).

## 2.4. Aproximaciones realizadas para construir un agente

Uno de los primeros problemas que nos podemos encontrar al realizar un agente para la GGP, es la propia definición de GGP. No se basa en un solo dominio, por lo que no podemos especializarnos. Este problema puede resumirse en que la dificultad de construir un agente para la GGP es la creación de una función de evaluación que funcione eficientemente en diferentes dominios (Kaiser, 2007). Además tenemos que contar con los límites de la propia creación del agente, el tiempo de ejecución es finito y el espacio en memoria es limitado. En esta sección vamos a hacer un repaso de ciertos agentes creados con el objetivo de abordar la competición de GGP.

### 2.4.1. Cluneplayer

Es el ganador de la primera competición de GGP en 2005 (J, 2007). Su aproximación fue usar una función heurística sobre un árbol MiniMax con poda alfa-beta y tablas de transposición. Para mejorar la función heurística, el agente extrae de la propia definición del juego mejoras que pueden ayudar a entender el juego. Incluye técnicas como cuantas veces aparecen un número determinado de átomos, la distancia entre éstos, lo avanzado que esta en el juego u otras características (como las simetrías del juego). Muchas de estas características no son realmente útiles para el juego, pero para obtener qué valores son los realmente importantes se usa un factor conocido como **estabilidad**. Un factor que cambia mucho si se ejecuta el juego de manera aleatoria, es marcado como muy inestable. Si el cambio es casi imperceptible se considera poco inestable. Usando estos factores de estabilidad, podemos usarlos como pesos para obtener la función heurística.

### 2.4.2. FluxPlayer

Es un jugador para la General Game Playing (GGP) que ganó la competición en 2006 mediante un motor de inferencia basado en Prolog y usando funciones heurísticas. Siguiendo la estructura básica de un agente, FLUXPLAYER (Schiffel and Thielscher, 2007) tiene la siguiente estructura:

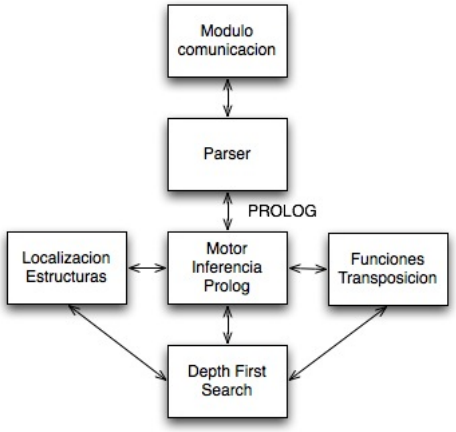


FIGURA 2.18: La estructura del FluxPlayer.

Como se puede ver en la estructura el *parser* convierte el lenguaje a Prolog para luego ser usado por un eficiente motor de inferencia de Prolog, Fluent Calculus. Según indican, aunque funciona correctamente este *parser*, puede ser mejorado para obtener mejor provecho de la traducción a Prolog.

El motor de búsqueda usa una versión iterativa del *depth-first search*, con dos mejoras provistas por el analizador del juego: Tablas de transposición y almacenaje de heurísticas. En juegos pequeños donde el número de estados se puede expandir en un tuno, como es en el caso del Tic-Tac-Toe, no es necesario usar funciones heurísticas ya que el propio motor de búsqueda puede expandir completamente el árbol de estados. Además dependiendo del juego usado (un solo jugador, multiples jugadores, suma nula o no suma nula), se usan diferentes técnicas de poda.

Como la mayor parte de juegos son demasiado grandes como para encontrar una solución en el tiempo de turno, resulta necesaria la implementación de una buena función heurística. La idea principal es usar intervalos *difusos* para indicar los hechos verdaderos o los hechos falsos. Después, usando funciones de evaluación que analizan la relación entre estados verdaderos y estados falsos, se puede obtener la solución.

Una forma de mejorar la función heurística es identificar estructuras conocidas que aporten información adicional. La estructura más conocida que se usa es la de los sucesores, que consiste en una serie de hechos que indican una secuencia de pasos hasta uno final, para limitar la profundidad del árbol de decisión. Otra posible estructura que se suele usar muchos es *identificar* un tablero mediante el reconocimiento de los hechos que reflejan las celdas del mismo.

Si se reconoce una estructura de tablero, se pueden usar heurísticas ya conocidas, como la distancia de Manhattan, para mejorar la función heurística general encontrada. Los juegos donde el objetivo es llegar a una posición determinada de un tablero son jugados correctamente, mientras que los juegos donde las funciones heurísticas son difíciles de realizar y es complicado diferenciar los distintos estados no terminales suelen dar problemas a este agente.

### 2.4.3. CadiaPlayer

CADIA-Player (Finnson, 2007) ganó la competición de GGP en 2007, mediante el uso de técnicas derivadas de Monte Carlo como UCT (Kocsis and Szepesvári, 2006). Al igual que FluxPlayer, CADIA-Player también usa un motor de inferencia basado en Prolog. Para poder traducir las reglas de GDL a Prolog se usa un parser que convierte todo el fichero KIF en Prolog. La función de búsqueda usada depende del número de jugadores usados: si es un solo jugador usa el algoritmo IDA* (Korf, 1985), mientras que si son más de 1 jugador usará UCT con el enfoque paranoico, todos los jugadores están contra el jugador, por lo que el movimiento de todos los jugadores puede reducirse a uno solo. Se han usado métodos de paralelización para hacer más efectivo UCT y encontrar la solución más rápidamente. Para ello se han añadido extensiones al módulo HTTP para poderse comunicar con otras máquinas que realicen simulaciones.

## 2.5. Resumen

En el estado de la cuestión anteriormente expuesto hemos realizado una introducción de la competición GGP. Tras plantearse la necesidad de crear una competición que presentara problemas de IA que no estuvieran asociados a un dominio concreto, se plantea la competición GGP. En primer lugar se ha explicado el funcionamiento de la competición y como esta crea las partidas para cada clase diferente de juego. En la sección posterior se detalla el lenguaje GDL indicando la estructura del mismo y usando pequeños fragmentos del código para mejor la explicación. Después se ha detallado el funcionamiento interno del sistema que soporta las partidas de la competición, como se enlaza con los competidores, su estructura interna, y la forma que tiene de comunicarse entre los diferentes componentes. Para acabar la explicación de la competición GGP, se enumeran una serie de recursos para empezar el desarrollo de un agente que pueda competir en la misma.

A continuación se detallan una grupo de técnicas de búsqueda que van a ser usadas en el presente proyecto. Para poder explicarlas con una mayor profundidad, se han adjuntado

las técnicas sobre las que se asientan de forma que se puede ver el porque de su existencia. Además esta sección incluye una breve sección indicando como mejorar los algoritmos de búsqueda mediante el uso de diferentes técnicas. Por último se han detallado cada uno de los participantes en las previas competiciones de GGP, y cual su enfoque a la hora de participar en la misma.

# Capítulo 3

# Objetivos

Los objetivos de este proyecto son estudiar y crear un agente que compita en la General Game Playing Competition siguiendo los estándares definidos por la competición GGP y el Game Definition Language (Love et˜al., 2008). Este agente debe ser capaz de resolver un problema con cierta eficiencia, sin conocerlo de antemano y sí que medie ninguna intervención humana. Los objetivos de la realización de este agente son los siguientes:

1. Descubrir y estudiar los pasos necesarios para desarrollar un sistema GGP. No sólo es necesario saber que debe hacer un agente, sino que se debe conocer su estructura y dividir su desarrollo en diferentes partes que sean independientes entre sí.

2. Obtener una visión general de la competición *General Game Playing* y de las diferentes formas que se han usado para abordarla.

3. Comprobar si el lenguaje que formaliza los juegos y la competición en sí, permite el uso de cualquier clase de juego. No solo consistiría en criticar los fallos sino en tratar de resolver en mayor medida esas limitaciones. Si bien, estas limitaciones podrían venir impuestas para sólo centrarse en un subdominio del problema, es necesario saber cómo poder superar las limitaciones para poder enfrentarnos a todo el problema.

4. Encontrar funciones, algoritmos o técnicas que aporten la mayor cantidad posible de información de un dominio a priori desconocido. No solo consiste en buscar técnicas que aporten una información completa para cualquier problema, sino que la información que aporten sea válida.

5. Por último, pero no por ello menos importante, es necesario crear un agente que compita en la competición GGP. De esta forma se debe poner a prueba cada uno de los problemas anteriores. Además de crear el agente es necesario participar en

la competición para obtener la mayor información posible de esta para futuros encuentros.

# Capítulo 4

# Desarrollo

Esta sección describe el proceso de creación de un agente para participar en la *General Game Playing Competition*. En primer lugar se presenta la propuesta de creación del agente, detallando las ideas que se han usado para realizarlo y las razones de haberlo hecho de una determinada manera. Además se ha incluido un estudio del lenguaje GDL y de la competición, analizando sus problemas a la hora de representar cualquier clase de juego y varias propuestas para poder hacerlo. A continuación se presentan los casos de uso en UML [1], permitiéndonos representar las relaciones entre los diferentes actores involucrados y sus relaciones.

A continuación se explica la arquitectura de la aplicación, primero desde un punto de vista de componentes y luego un diagrama de clases explicando la aplicación en sí. En el diagrama de clases, se representa la estructura del agente indicando sus componentes, clases, métodos y atributos y las relaciones existentes entre éllos.

El manual de usuario describe todos los procedimientos y pasos necesarios para poder usar el agente GGP con el simulador o para participar en la competición. Adicionalmente, se ha incluido un manual de referencia, para explicar todos los pasos necesarios para ampliar y mejorar el agente.

## 4.1. Explicación del Agente

Para poder crear un agente para la *General Playing Competition* (GGP) es necesario crear un sistema capaz de responder frente a cualquier acción dando una respuesta adecuada a la misma. Desde un punto de vista lógico el problema consiste en encontrar la solución de un árbol de búsqueda del que no se conoce nada. Para afrontar el problema, se ha

[1] UML: Unified Modeling Language

reducido a dos subproblemas, encontrar el algoritmo de búsqueda dentro del árbol de estados y la forma de obtener la mayor información posible del juego usado.

Si nos basamos en la forma extensiva de juego, podemos modelizar el espacio de búsqueda de un juego como un árbol con nodos. En los problema de dos jugadores un jugador realiza un turno en una fila del árbol y la siguiente fila es el turno del siguiente jugador. Por supuesto esto implica dos importantes supuestos: Sólo dos jugadores y juegos por turnos. Los algoritmos que se basarán en esos supuestos no valdrían para resolver problemas en GGP ya que estos pueden tener varios jugadores y la forma de ejecutarlos siempre es simultánea (en los juegos por turnos todo los jugadores realizan un movimiento, aunque este movimiento sea un noop [2]).

Una forma de tratar a todos los jugadores es suponer que todos los jugadores conspiran entre sí para derrotar al jugador (intentan todos minimizar su puntuación). Esta suposición no siempre es válida porque muchas veces los jugadores pueden tener objetivos diferentes, o intentar minimizar la puntuación de todos los jugadores o incluso realizar acciones en contra de sus intereses. No obstante y debido a la naturaleza del problema de realizar un agente para la GGP, las desventajas de tener información errónea por realizar el supuesto se minimizan si las comparamos con la facilidad de representar el problema y por consiguiente poder resolverlo.

Los juegos en GGP son simultáneos, los turnos de todos los jugadores se ejecutan a la vez, por lo que para pasar de un estado al siguiente es necesario realizar el turno de todos los jugadores. No obstante se puede simplificar esta situación de forma que se puedan representar como juegos por turnos. La idea básica es usar estados intermedios que representen la acción de realizar el turno del jugador y los turnos del resto de jugadores. Por ejemplo, en un juego de estrategia, donde se mueva una unidad a la vez de cada jugador se haría en dos acciones. Una sería la acción de mover la unidad del jugador dejando el estado en un estado intermedio y la otra sería la acción de mover el resto de unidades, dejando el estado en el siguiente estado.

### 4.1.1. Elección del algoritmo de búsqueda

En teoría de juegos llamamos forma extensiva de un juego, a una representación de este en forma de árbol, donde cada nodo representa una situación donde un jugador toma una decisión. En el documento que describe la GGP (Genesereth and Love, 2005) un juego se puede reducir a una máquina de estados donde todos los jugadores realizan las acciones de manera síncrona. Esto se debe a que la representación como una máquina

[2] No operation

de estados es más compacta y menos compleja que la de forma de árbol, pero desde un punto de vista lógico representan la misma información.

Para poder explorar este árbol es necesario usar un algoritmo de búsqueda para poder hallar una posible encontrar un determinado elemento. Una posible opción es usar algoritmos de búsqueda no basados en información, pero como se ha ido demostrando en la experiencia (y en las pruebas realizadas), normalmente un algoritmo con cierta información sobre el entorno suele ser mejor que un algoritmo sin ninguna información. En nuestro caso a la hora de seleccionar un algoritmo decidimos elegir dos algoritmos diferentes en función del número de jugadores. Esto se debe a que el comportamiento de otra entidad *inteligente* añade una incertidumbre que debe resolverse de una forma diferente a una búsqueda sin otros agentes.

#### 4.1.1.1. Búsqueda contra la naturaleza

En las búsquedas contra la naturaleza el objetivo del agente es encontrar el mejor camino (sucesión de movimientos) para llegar a una posible solución. La primera opción planteada fue usar el algoritmo A* visto en la página 13. Este algoritmo usa una función heurística para hallar la solución. Si bien es un buen algoritmo, existen dos grandes problemas que nos impiden su uso en el agente a realizar. El primer problema es que al usar una *lista abierta* el algoritmo tiene una complejidad en memoria demasiado elevada por lo que desde un punto de vista práctico no es útil, ya que expandiría en memoria un número exponencial de nodos. Otro problema encontrado es el temporizador de turno, que limita la búsqueda a un tiempo determinado y el algoritmo funciona mejor en situaciones donde no hay una dependencia temporal. Por estas razones se evalúo el algoritmo RTA*, visto en la página 15, como implementación al problema encontrado.

El algoritmo RTA* es capaz de alcanzar soluciones en entornos con información limitada. La diferencia fundamental con el algoritmo A* es que las funciones $g$ y $h$ se basan en el estado actual del problema (Korf, 2007). El consumo de memoria de este algoritmo es menor que el de A*. Debido a estas razones es un algoritmo que es factible de ser implementado. No obstante uno de los problemas que se nos plantea a la hora de usar RTA* es el hecho de que nos lleve a una solución errónea debido a una mala estimación de las heurísticas. Para solucionarlo surge el algoritmo Learning RTA* o LRTA* 2.3.3, como se vio en la página 18. La primera ventaja de este algoritmo respecto al RTA* es su convergencia, es decir que en algún momento este algoritmo es capaz de encontrar la solución correcta al problema. Para ello el espacio de búsqueda debe tener un espacio finito de estados, con costes positivos y no sobreestimados (Korf, 2007). Una ventaja

adicional sobre este algoritmo es que puede retroceder algunos pasos en una solución en caso de la solución tomada implique mayor coste que volver hacia atrás.

Este algoritmo funciona bien en problemas contra la naturaleza, pero aún así nos hemos encontrado con ciertos problemas:

- **Estados iguales**: Si recordamos el funcionamiento del algoritmo LRTA*, este permitía volver a un nodo ya visitado en caso de que la función de evaluación almacenada fuera menor que la función de evaluación del resto de posibles opciones. No obstante debido a la forma en que se representan los problemas no es posible volver retroceder en el árbol de búsqueda. Cada nodo del árbol es único ya que una de las condiciones para hacer las reglas de un juego de GGP es que su espacio de estados sea finito. Un ejemplo de esto se puede ver en el siguiente ejemplo del mundo de los bloques.

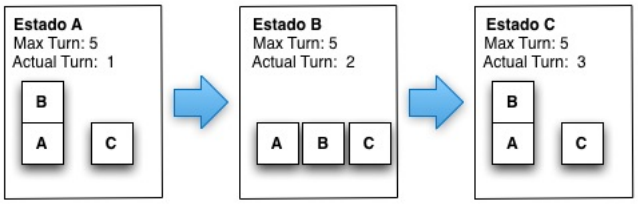


FIGURA 4.1: Ejemplo de dos estados iguales.

  En este ejemplo los estados A y B son iguales si consideramos únicamente las variables no temporales. En el algoritmo LRTA* el movimiento representado entre el estado B y el estado C significaría que el estado A y C son el mismo y que la función heurística del estado A se volvería a modificar. No obstante al no ser el mismo estado se almacenarían en diferentes posiciones de la tabla *hash* de funciones heurísticas de LRTA*. Esto se debe a que en la mayor parte de juegos existen hechos que indican el número de turno y de esta forma evitar que el juego tenga una duración ilimitada.

  Para resolver este problema ha sido necesario crear otro método de comparación entre estados que indique si dos estados son iguales absteniéndose de comprobar los hechos *temporales*. Este método compara uno a uno cada hecho entre dos estados menos aquellos que indiquen el turno actual. La explicación de cómo identificar los hechos temporales se presenta en la sección 4.1.2.1, página 40.

- **Funciones heurísticas**: El problema de usar un algoritmo heurístico es que una mala función heurística puede impedir la convergencia del algoritmo o ir por caminos no óptimos. Si bien este apartado se tratará con más detalle en la siguiente

sección, es importante señalar que una función heurística para un jugador o varios no tienen porque construirse de forma igual.

#### 4.1.1.2. Búsqueda multijugador

Como se comentó anteriormente se ha decidido suponer que todos los jugadores conspiran contra el jugador. De esta forma se pueden usar diferentes técnicas de búsqueda para dos jugadores como el minimax 2.3.4, página 20. No obstante para realizar la búsqueda con varios jugadores se opto por otro algoritmo, los números conspiratorios 2.3.6, 23. Para llegar a este algoritmo nos basamos en una serie de supuestos e ideas que nos llevaron a elegirlo.

En primer lugar suponemos que los oponentes no son inteligentes. El minimax, en el turno del adversario, intenta minimizar los puntos del jugador. No obstante en GGP, los agentes no son perfectos y sus funciones heurísticas no siempre son correctas o admisibles, por lo que no siempre es cierto que el jugador contrario intente minimizar al jugador rival. Debido a esto ya no buscamos como en el minimax maximizar nuestra función sino elegir aquella rama que de la solución más probable, la más estable.

Una posición estable es aquella que tiene menor probabilidad de cambiar durante la partida. Si recordamos el algoritmo, una posición estable necesita muchos más nodos que conspiren para cambiar su valor. En la competición GGP, al tener funciones heurísticas que aportan poca información, es mejor tomar caminos que nos aseguran que su valor es poco probable que cambie, a tomar caminos con mejor valor con una baja probabilidad de que ese valor sea correcto.

Un ejemplo en el mundo real de porqué nos interesan posiciones estables sería la búsqueda de un tesoro. En una situación normal donde la función heurística nos aporta mucha información, por ejemplo si tuviéramos el mapa, es mejor tomar el movimiento que nos indique la función heurística. Si en cambio, sólo nos podemos basar en rumores, una situación donde la función heurística aporta baja información (como es el caso de la competición), nos tendríamos que basar en la robustez de la acción a tomar. Si 2 personas nos dicen que el tesoro está a 10 metros, y 50 que esta a 100 metros, haríamos caso a estos últimos porque es menos probable que todos ellos estén equivocados.

En conclusión, usamos el algoritmo de números conspiratorios, porque en situaciones de baja información heurística nos interesa más la **robustez de la solución** que cómo es de buena.

### 4.1.2. Obtención de información

Una vez elegidos los algoritmos de búsqueda es necesario obtener información adicional que guíe la búsqueda. Lo más lógico es usar una función heurística, ya que es necesaria para cualquiera de los algoritmos usados. Por otro lado mejorar la búsqueda analizando las reglas del juego es también una buena fuente de información.

#### 4.1.2.1. Reconocimiento de Patrones

Definimos patrón como una estructura que se repite en diferentes situaciones. Si se logran identificar esas estructuras y conociendo su utilidad, se puede obtener información de un conjunto nuevo de reglas. En el desarrollo del proyecto no se ha avanzado en este tema debido a su complejidad dada la cantidad de combinaciones de elementos en las reglas GGP. La idea básica sería identificar trozos de las reglas de un juego, como por ejemplo las reglas relacionadas con el tamaño del tablero, y añadir información a la función heurística relacionada con esas reglas, sabiendo el tablero se puede deducir la forma de este y por ejemplo aplicar heurísticas conocidas como una distancia de Manhattan.

No obstante el reconocimiento de patrones se ha usado en una situación en concreto, en la identificación de **estructuras temporales**. Llamamos estructura temporal al conjunto de relaciones y hechos encargados de representar la secuencia de turnos de un juego. Por ejemplo, en el ajedrez se pueden dar situaciones donde el árbol de búsqueda es infinito ya que una pieza puede realizar los mismo movimientos de forma cíclica. Para evitar esto, ya que todo juego en la GGP debe ser finito, se añade un número máximo de turnos, siendo sólo necesario en los juegos no finitos (por ejemplo, el tic-tac-toe).

La estructura temporal se basa en un hecho que indique el turno inicial y de un conjunto de hechos que indiquen la secuencia de los mismos, todos éllos deben ser información estática del juego (deben permanecer constantes durante toda la partida).

```
(at k1)
(s k1 k2)
(s k2 k3)
...
(s kn-1 kn)
```

El conjunto de hechos anterior muestra la típica estructura de turnos. Hay dos tipos de hechos, `at` que indica el turno inicial y `s` que indica el consecutivo. El hecho consecutivo indica la secuencia de hechos, empezando estos en el turno `k1` hasta el turno `kn`. Además

para asegurar que esta estructura es correcta, durante las simulaciones realizadas para obtener la función heurística se calcula la profundidad máxima del árbol de búsqueda, que debería coincidir con el número de turnos. Aun así siempre existe la posibilidad de que la estructura identificada como un patrón no lo sea, en cuyo caso todos los elementos que lo hubieran asumido fallarían.

#### 4.1.2.2. Función Heurística

Tanto LRTA*, como CN-Search usan una función heurística para guiar la búsqueda. Dado un estado, la función heurística indica cual de las acciones, desde ese estado, es la mejor para alcanzar la solución. Debido a que los juegos en GGP son generales, encontrar una función heurística que sea optima (o cercana a ser óptima) y que sea admisible es complicado ya que hablamos de funciones generales para cualquier tipo de situación. Además las técnicas típicas de obtención de la función heurística (relajación de variables y uso de subproblemas) no son posibles debido al carácter general de la competición. Para ayudar a las funciones heurísticas a obtener información se usaron simulaciones durante el tiempo previo al comienzo del primer turno y tras recibir el mensaje de inicio de partida (el `startclock`).

Tras iniciarse la partida, después de recibir el mensaje `play`, se ejecuta un agente especial llamado Simulador. Este agente explora de forma exhaustiva el árbol para obtener una serie de valores con los que realizar la función heurística. Los valores obtenidos en cada partida son:

- **Profundidad Máxima**: Es la profundidad máxima del árbol de búsqueda a la que ha sido posible llegar usando el simulador. Este valor se usa además para comprobar que el número de turnos es el correcto. Además con el tamaño máximo de ramificación se puede obtener una cota superior del número de nodos del árbol de búsqueda.
- **Profundidad Media**: Es la profundidad media del árbol de búsqueda. Usando además el valor de ramificación media se puede obtener el valor medio de nodos en el árbol. Para calcular este valor se explora el árbol obteniendo las diversas profundidades de cada solución y luego se divide entre el número de nodos terminales.
- **Estabilidad Profundidad**: Este valor se calcula después de realizar la simulación y representa la diferencia que hay entre la profundidad media y la máxima.
- **Ramificación Máxima**: Tras explorar el árbol de búsqueda, este valor representa el nodo con mayor valor de ramificación de todo el árbol.

- **Ramificación Media**: Es la media de ramificación por cada nodo del árbol. Para calcularlo es necesario explorar la mayor parte de los nodos.
- **Número de nodos terminales**: Es el número de nodos terminales que han sido explorados por el simulador.
- **Número de nodos del árbol**: Cuanto más grande sea el árbol, más complicado será encontrar una solución que se base en los datos de éste.
- **Tamaño medio de un estado terminal**: Para saber si se ha llegado a un estado terminal, se usan las relaciones `terminal` indicadas en las reglas del juego. Cada estado final posible debe cumplirlas, pero eso no significa que todos los estados finales tengan el mismo número de hechos. Esta variable representa el número medio de hechos que tienen los estados terminales.

Una vez acabado el tiempo de inicio de la partida se dan estos datos a la clase `Heurística` 4.5 en la página 60, la cual los usa para generar la función heurística. La función heurística se genera usando los datos de la simulación y del estado actual del juego, teniendo el siguiente aspecto.

$$h(x) = c * \sum_{i=1}^{n} f_i(x, y) * \alpha_i$$

Además debe cumplirse la siguiente condición:

$$1 = \sum_{i=1}^{n} \alpha_i$$

La función heurística se compone de una serie de funciones, basadas en el estado actual y las reglas del juego, que se multiplican por unos pesos en función de la importancia relativa de cada función. Esta función se multiplica por una constante $c$ de forma que su valor este comprendido siempre entre 0 y 100 y sólo se realizará si el nodo al que llegamos usando la acción no es terminal (en ese caso conoceríamos el valor exacto y dejaría de tener sentido una función heurística). Una estado con un valor por debajo de 50 es un estado meta donde se pierde la partida. La condición de la función indica que los pesos que se usan en la función heurística deben sumar 1 de forma que todo el valor heurístico salga de dicha función.

En el caso de nuestro agente se han usado tres funciones, aunque se ha dejado la posibilidad de que se pueda ampliar el número de funciones que usa la función heurística. Las funciones realizan supuestos sobre las simulaciones de forma que si bien no son más que aproximaciones, aunque alguna no sea del todo certera podrán aproximarse mejor a la solución que sin existir. Las tres funciones son:

- **Función de ramificación**: Esta función va en relación con el factor de ramificación de cada nodo. La idea de esta función es devolver un valor cada vez más bajo a medida que se acerca al tamaño medio de ramificación y no devolver nada cuando se llegue al máximo o se supere. Con esto indicamos que a menor libertad de movimiento más cerca estaremos de la meta. Obviamente esto no siempre es cierto, pero podemos asumir que en la mayoría de juegos a medida que se avanza a la meta en ellos el número de operaciones permitidas se reduce.

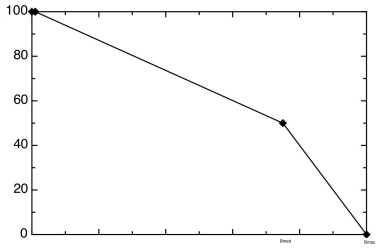


FIGURA 4.2: Función del factor de ramificación.

- **Función de profundidad**: Esta función se basa en el hecho de que a más cerca este de la profundidad media más cerca está de la solución.

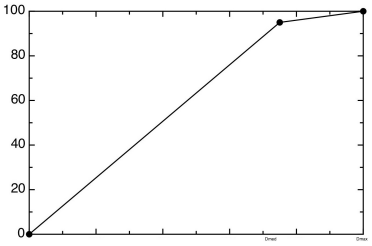


FIGURA 4.3: Función del factor de profundidad.

- **Similitud de tamaño**: La idea de esta función es calcular el porcentaje de diferencia entre el número hechos del estado actual y la media del número de hechos de la solución. A menor sea la diferencia en hechos es más probable que nos estemos acercando a la solución.

Si bien estas funciones nos sirven como base para crear la función heurística tanto su relevancia, medida con pesos, como su utilidad necesitan de un mayor estudio. Los valores usados para esta función heurística son:0.3, 0.3 y 0.4. Estos valores pueden ser mejorados mediante una batería de pruebas y el uso de algoritmos que modifique dichos pesos.

## 4.2. Estudio del lenguaje GDL y competición GGP

En este apartado se realiza un estudio para extender el lenguaje GDL y la competición GGP para poder usar cualquier tipo de juego. Tal y como se explicó en el estado de la cuestión 2, página 5, GDL [3] es un lenguaje que permite definir en principio cualquier tipo de juego. No obstante mientras se iba desarrollando el lenguaje nos dimos cuenta de ciertos limitaciones que tenían tanto el lenguaje, como la competición y la manera de resolverlas.

### 4.2.1. Juegos no deterministas

Un juego donde cada transición entre estados tiene una cierta probabilidad de que se realice es un juego no determinista. Esta clase de juegos son importantes porque la mayor parte de las situaciones del mundo real implican un cierto grado de incertidumbre.

Ni el lenguaje, ni la competición tiene una forma de representar aleatoriedad sin cambiar su estructura. Como se ha visto en secciones anteriores, un fichero GDL se compone solo de reglas y hechos. Una forma de representar juegos no deterministas es crear secuencias de números pseudo-aletorios en el fichero GDL. De manera similar a la estructura creada para representar los turnos de cada juego, vista en la sección 4.1.2.1, página 40, se puede crear una estructura para representar una secuencia de números *aleatorios*.

```
(init (ude 1 3))

(<= (next (seq ?y ?z))
    (true (seq ?x ?k))
    (con ?x ?y ?z))

(ite 1 2 5)
(ite 2 3 7)
(ite 3 4 9)
(ite 4 1 8)
```

El número aleatorio es la tercera constante en el hecho `ite`. Estos hechos definen en un bucle la secuencia de números 5, 7, 9, 8. La ventaja de esta solución es que no es necesario cambiar ni la estructura de GDL ni de la competición GGP. Además esta solución ofrece un método sencillo de crear estructuras aleatorias. Uno de los problemas de esta solución es que las secuencias de números son conocidas por todos los jugadores,

[3] Game Description Language

por lo que estos podrían conocer el resultado de sus acciones interpretando las reglas. Además esta solución crea secuencias de números pseudo-aleatorios, que nunca cambian durante las ejecuciones.

Otra solución es crear un *jugador aleatorio*, gestionado por el *GameMaster* y que simula la aleatoriedad. Justo después de que un jugador realice un movimiento, el *jugador aleatorio* realiza un movimiento que simula la aleatoriedad del movimiento del jugador. Esta solución puede ser implementada usando una regla que cree un hecho, y el *jugador aleatorio* asigne un número aleatorio a esta regla.

```
(<= (next (rfact ?x))
      (does randomplayer (rmove ?x)))
```

No obstante, esta solución implica cambiar la competición de GGP, además del Game Master para poder gestionar el *jugador aleatorio*. Aún así esta solución ofrece una mayor aleatoriedad que la solución anterior y no necesita cambiar la estructura de GDL.

Por último se puede crear una primitiva que genere aleatoriedad, ampliando de esa forma el lenguaje GDL. No obstante esto implicaría modificar tanto el lenguaje GDL como el Game Master para poder interpretar dicha solución.

### 4.2.2. Comunicación y juegos cooperativos

Los juegos cooperativos pueden ser teóricamente implementados en GDL sin necesidad de añadir información adicional. La competición de GGP permite jugar juegos multijugador incluso si estos no son simétricos. La única condición impuesta en la competición (no en GDL) es que estos sean *weakly-winnable*. Un juego es *weakly-winnable* si y sólo si para cada jugador existe al menos una serie de pasos donde el valor del objetivo es máximo para dicho jugador.

No obstante uno de los problemas que existe en el lenguaje GDL, es la imposibilidad de añadir una comunicación entre los diferentes jugadores. Por ejemplo, en un juego de estrategia se pude permitir intercambiar tropas entre los jugadores. Una posible regla que permitiese esta comunicación entre jugadores sería:

```
(<= (next (comarmies ?player ?x ?s ?player2 ?y ?t))
     (does ?player (comexchange ?x ?s ?y ?t))
 (true (armies ?player ?x ?n1))
 (true (armies ?player2 ?y ?n2))
)
```

Esta regla crea un hecho auxiliar, para representar la comunicación del intercambio de ejércitos entre jugadores. No obstante esta solución tiene el problema de que la comunicación es visible para todos los jugadores, no es un hecho diferenciado, y que no se permite dividir las reglas entre acciones y mensajes de comunicación. No obstante es mejor no dividir acciones y mensajes de comunicación como entidades separadas ya que la idea original de GDL es ser lo más general posible. Por lo tanto el problema principal que nos encontramos, ya que será analizado en el siguiente punto es la forma de ocultar información en GDL.

### 4.2.3. Información Imperfecta

Los juegos de información imperfecta se caracterizan porque parte de las reglas, acciones, o hechos que los definen están ocultos a algunos o todos los jugadores. La forma en que está planteada la competición no se permite jugar juegos imperfectos. Al principio del juego, el *GameMaster* envía a todos los jugadores las reglas del juego. Esta es la única comunicación, además de la acción realizada por cada jugador en el último turno, que tienen los jugadores sobre el juego.

Para entender las limitaciones de GDL sobre juegos imperfectos es mejor verlo a través de un ejemplo, Kriegspiel (Williams, 1950). Kriegspiel es una variante del ajedrez en la que sólo se ven las piezas del jugador y la información de capturas y mates. En este juego es necesario tener un árbitro para aceptar o rechazar los movimientos, informar sobre movimientos e informar del nuevo estado del tablero tras cada movimiento. Es decir hay tres jugadores, un árbitro con información completa del juego y dos jugadores con información parcial del mismo. Los jugadores no pueden saber a qué estado les va a llevar un determinado movimiento, ni las consecuencias de un movimiento.

El primer problema es distribuir ficheros con información incompleta a cada jugador. Sin embargo no es posible enviar diferentes reglas a cada jugador sin cambiar el *GameMaster*. Si bien una primera solución consistiría en crear un fichero diferente por cada jugador, el mantenimiento de los mismo además de la realización de los turnos es demasiado complicada. Nuestra solución consiste en añadir una nueva relación para marcar los hechos y reglas según los jugadores que puedan usarlos. Un ejemplo sería:

```
(mark-info (players) (rules/facts))
```

Con esta relación sabemos que las reglas o hechos dentro de la relación `mark-info` son visibles para los jugadores indicados. En el caso de Kriegspiel:

```
(mark-info black-player ( (cell a 1 br) (cell d 3 br)))
```

En este ejemplo **sólo** el jugador negro puede saber que tiene un alfil en la celda `a1` y otro en la celda `d3`. Usando esta solución el *GameMaster* solo tiene que filtrar las reglas en función del jugador al que la envía y la relación `mark-info`.

Un ejemplo para una regla sería:

```
(mark-info (black-player)
(<= (next (cell ?p ?x ?y))
(mark-info white-player (next (cell ?u ?v b)))
    (does ?player (move ?p ?u ?v ?x ?y)))
)
```

En este ejemplo el jugador negro conoce todas las reglas, sin embargo el jugador blanco solo conoce una. De esta forma la relación `mark-info` puede agrupar a otras relaciones `mark-info` siempre teniendo presente que en caso de duda siempre manda la relación superior. En el ejemplo aunque el segundo hecho sólo sea visible para el jugador blanco, también lo es para el negro porque este agrupa a toda la relación `mark-info`.

Otro problema existente es leer movimientos ilegales de un jugador y transmitir una información adecuada. Si un jugador en Kriegspiel mueve un peón pero existe una pieza del contrario en la siguiente casilla, se indicaría con un error, y se pediría una nueva información. En GGP, esta situación sería resuelta generando un movimiento aleatorio pero valido por jugador. Para evitar esto la solución propuesta consiste en crear un mensaje de *turno ilegal* y reiniciar el tiempo de juego para ese jugador. No obstante tampoco es una solución perfecta, ya que un jugador podría constantemente realizar un movimiento ilegal para tener más tiempo que su oponente. Una solución consiste en es añadir un hecho especial en las reglas de juego que permita al diseñador del mismo indicar la cantidad de movimientos ilegales que se pueden hacer en un turno. Un ejemplo de esta relación sería:

```
(n-illegal  5)
```

Usando estas soluciones podemos realizar operadores de comunicación de una forma más sencilla y mejor que la planteada en la sección anterior 4.2, página44.

```
(mark-info (player1 player2)
(<=
(next (armies ?p1 ?a2 ?k2))
(next (armies ?p2 ?a1 ?k1))
```

```
     (does ?player (comexchange ?p1 ?a1 ?k1 ?p2 ?a2 ?k2))
(true (armies ?p1 ?a1 ?k1))
(true (armies ?p2 ?a2 ?k2))
)
)
```

En este caso sería necesario duplicar esta regla en caso de que se tuviera más jugadores (se tendrían que hacer todas las posibles combinaciones de reglas). Para evitar esta duplicidad innecesaria de reglas, una solución propuesta sería usar una variable para representar a cualquier rol de jugador y que cambiase según la instanciación de la regla (es decir que sirviese tanto para una comunicación entre el jugador 1 y 2, como para el de 3 y 4).

```
(mark-info (?p1 ?p2)
(<=
(next (armies ?p1 ?a2 ?k2))
(next (armies ?p2 ?a1 ?k1))
     (does ?player (comexchange ?p1 ?a1 ?k1 ?p2 ?a2 ?k2))
(true (armies ?p1 ?a1 ?k1))
(true (armies ?p2 ?a2 ?k2))
)
)
```

## 4.3. Casos de uso

El diagrama de casos de uso nos muestra como debería interactuar el sistema con los usuarios en diferentes escenarios. De esta forma obtenemos el comportamiento del sistema, y podremos obtener datos adicionales que nos serán útiles para el diseño del mismo.

### 4.3.1. Actores de los casos de uso

Los actores son entidades externas al sistema que se encargan de interactuar con él para obtener ciertos comportamientos. En este caso hay dos actores, uno encargado de ejecutar al agente y otro encargado de crear las partidas y realizar las acciones de comunicación con el agente.

- **Game Master**: Se encarga de realizar las acciones sobre el agente durante el transcurso de la partida.

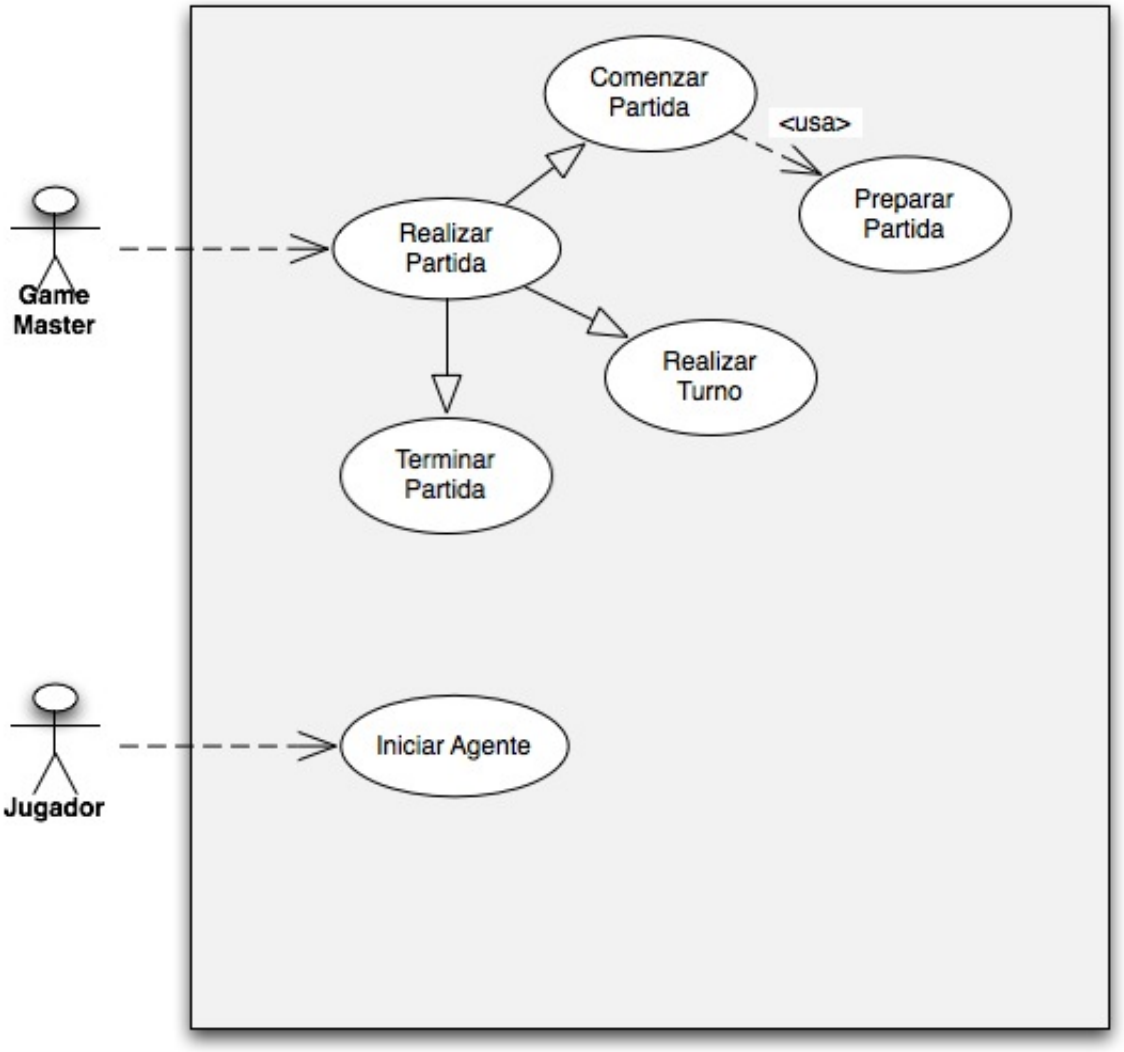


FIGURA 4.4: Diagrama de casos de uso.

- **Jugador**: Se encarga de arrancar al agente introduciendo los parámetros necesarios para ejecutarse.

### 4.3.2. Listado de casos de uso

A continuación se describen los casos de uso relacionados con cada uno de los actores implicados. Los apartados de cada caso de uso son:

- **Id**: Identifica de forma unívoca cada caso de uso mediante el siguiente código: **CUXX** siendo XX un número natural empezado en 01.
- **Título**: Identifica al caso de uso de manera clara y sencilla con unas pocas palabras.

- **Descripción**: Describe en profundidad el caso de uso y todas las relaciones que tiene.
- **Actores**: Indica a los actores participantes en el caso de uso.
- **Precondiciones**: Elementos que deben estar/ocurrir para que pueda realizarse el caso de uso.
- **Escenario**: Pasos necesarios que se producen para realizar el caso de uso.
- **PostCondiciones**: Acciones y efectos producidos tras realizarse el caso de uso.

| | |
|---|---|
| Id | CU01 |
| Título | Realizar Partida |
| Descripción | Encargado de realizar la partida. Este caso de uso contiene otros subcasos que representan la comunicación entre el Game Master y el agente. |
| Actores | Game Master |
| Precondiciones | No Procede |
| Escenario | Este caso de uso se divide en 3 casos de uso:<br>▪ Comenzar Partida: Encargado de iniciar la partida<br>▪ Realizar Turno: Realiza un turno de la partida.<br>▪ Terminar Partida: Realiza las operaciones necesarias para acabar la partida. |
| Postcondiciones | No procede |

Cuadro 4.1: Caso de Uso 01: Realizar Partida

| | |
|---|---|
| Id | CU02 |
| Título | Comenzar Partida |
| Descripción | El sistema inicia una partida nueva, usando los jugadores, reglas de juego, tiempos de comienzo y valores definidos en el Game Editor por un gestor de partidas. |
| Actores | Game Master |
| Precondiciones | ▪ La partida no existe.<br>▪ Están cargadas las reglas del juego a usar en el Game Editor.<br>▪ Los agentes están cargados en el Game Editor con sus datos asociados: Dirección IP, puerto y nombre.<br>▪ Los agentes han sido iniciados en la dirección/puerto especificada en el Game Editor.<br>▪ Se ajustan los datos de los temporizadores de proceso de datos y de turno. |
| Escenario | ▪ Mediante la interfaz del Game Editor se envia al agente un mensaje de inicio de la partida.<br>▪ El agente recibe los datos de la partida. |
| Postcondiciones | ▪ El agente crea una instancia del módulo de juego para realizar las simulaciones.<br>▪ Se inicializan las variables del agente encargadas de su ejecución.<br>▪ El agente envia un mensaje de *Acknowledge* para indicar que se recibió el mensaje de inicio de partida de manera correcta. |

Cuadro 4.2: Caso de Uso 02: Comenzar Partida

| | |
|---|---|
| Id | CU03 |
| Título | Preparar Partida |
| Descripción | Tras recibir un mensaje de recibir una partida el sistema arranca las simulaciones para obtener la información necesaria para jugar. |
| Actores | Game Master |
| Precondiciones | ▪ La partida ya existe.<br>▪ Están cargadas las reglas del juego a usar en el Game Editor.<br>▪ Los agentes están cargados en el Game Editor con sus datos asociados: Dirección IP, puerto y nombre.<br>▪ Los agentes han sido iniciados en la dirección/puerto especificada en el Game Editor.<br>▪ Se ajustan los datos de los temporizadores de proceso de datos y de turno.<br>▪ Se ha recibido el mensaje de iniciar la partida. |
| Escenario | ▪ El agente inicia los simuladores durante el tiempo estimado para el inicio de la partida.<br>▪ Se recaba información para las heurísticas de los simuladores. |
| Postcondiciones | ▪ El agente envia un mensaje de *Acknowledge* para indicar que se recibió el mensaje de inicio de partida de manera correcta.<br>▪ Se arranca el agente para realizar el primer turno. |

Cuadro 4.3: Caso de Uso 03: Preparar Partida

| | |
|---|---|
| Id | CU04 |
| Título | Realizar Turno |
| Descripción | Tras recibir un mensaje con los movimientos de todos los jugadores realiza su turno. |
| Actores | Game Master |
| Precondiciones | ▪ La partida ya existe.<br>▪ Se ha recibido el mensaje de iniciar la partida.<br>▪ Se ha creado el agente tras haber obtenido los datos el simulador.<br>▪ Se ha recibido un mensaje de inicio de turno. |
| Escenario | ▪ El agente arranca el temporizador de turno.<br>▪ El agente actualiza su estado en interno en función del movimiento de todos los jugadores.<br>▪ Se empieza a buscar el siguiente movimiento. |
| Postcondiciones | ▪ Una vez acabado el temporizador se elige el mejor movimiento y se envía al Game Master. |

Cuadro 4.4: Caso de Uso 04: Realizar Turno

| | |
|---|---|
| Id | CU05 |
| Título | Terminar Partida |
| Descripción | Se procesa el mensaje de fin de partida mostrando al usuario la puntuación final. |
| Actores | Game Master |
| Precondiciones | ▪ La partida esta creada.<br>▪ Se ha recibido el mensaje de fin de partida. |
| Escenario | ▪ El agente lee la puntuación final obtenida y muestra el resultado por pantalla. |
| Postcondiciones | ▪ El agente se queda esperando a ser cerrado o a recibir otro mensaje de inicio de partida. |

Cuadro 4.5: Caso de Uso 05: Terminar Partida

| | |
|---|---|
| Id | CU06 |
| Título | Iniciar Agente |
| Descripción | Se procesa el mensaje de fin de partida mostrando al usuario la puntuación final. |
| Actores | Jugador |
| Precondiciones | ▪ Se debe elegir el tipo de algoritmo de búsqueda a usar y los pesos de la función heurística.<br>▪ Se arranca el agente en el puerto y dirección especificada en el Game Editor. |
| Escenario | ▪ El agente lee la puntuación final obtenida y muestra el resultado por pantalla. |
| Postcondiciones | ▪ El agente se queda escuchando y a la espera de comenzar la partida en un puerto y dirección determinados. |

Cuadro 4.6: Caso de Uso 06: Iniciar Agente

## 4.4. Arquitectura

Para poder realizar un agente GGP se ha partido del agente GGP básico creado por la universidad de Stanford, **Jocular** [4] y se ha mejorado ampliando su funcionalidad. Como todo sistema GGP, nuestro agente sigue una estructura de capas. Este agente se relaciona con el Game Manager (el módulo encargado de gestionar partidas del Game Master) que a su vez se relaciona con el resto de agente.

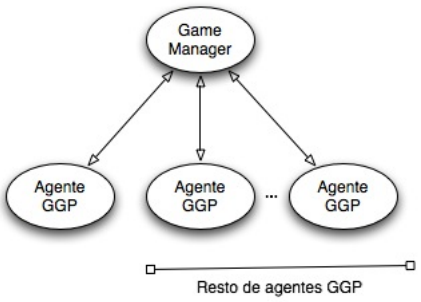


FIGURA 4.5: Relaciones del agente GGP.

Como se pudo ver en un apartado anterior 2.2.3, en la página 10 la estructura de capas de nuestro agente se divide en 4 módulos, cada uno de ellos con una funcionalidad independiente e interconectados entre sí.

### 4.4.1. Módulo de comunicación

Se encarga de realizar la comunicación entre el agente y el Game Master. Para hacerlo envía los mensajes HTTP definidos en la competición GGP. Los mensajes que puede recibir el módulo son:

- **START**: Este mensaje se usa para iniciar el juego. Contiene la información que necesita un jugador para jugar: Su rol, las reglas del juego y los temporizadores. Para obtener una descripción más formal acudir a la descripción del lenguaje GDL (Love et˜al., 2008).
- **PLAY**: Contiene el listado del último movimiento realizado por cada jugador en un determinado juego.
- **STOP**: Contiene la misma información que el comando PLAY, pero además indica el fin de la partida.

[4] http://games.stanford.edu/resources/reference/jocular/jocular.html

El agente además tiene una serie de mensajes para enviar en función del mensaje recibido.

- **READY**: Indica que el agente esta listo para comenzar la partida tras recibir un mensaje de comienzo, START.
- **Accion**: Se envia una acción instanciada indicando que ese jugador realiza ese movimiento. Este mensaje se envia tras recibir un mensaje de inicio de turno, PLAY. Por ejemplo, en el juego del tic-tac-toe si se manda el mensaje `(MARK 1 2)` indicaría que ese jugador desea realizar ese movimiento.
- **DONE**: Tras recibir un mensaje de fin de juego, STOP, este mensaje indica que el agente acaba la partida y se desacopla de la misma, quedando en espera para empezar la siguiente.

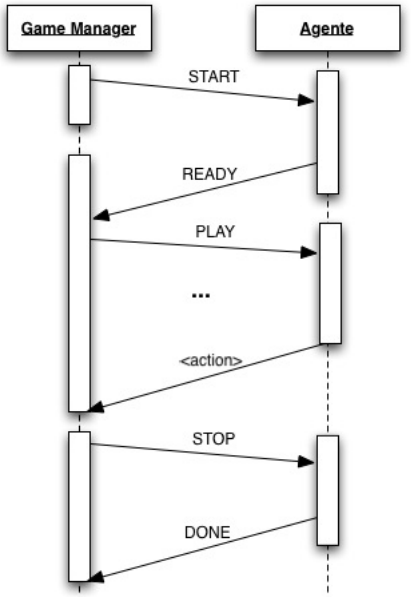


FIGURA 4.6: Mensajes intercambiados entre el GM y el Agente.

Tras arrancar el agente en un puerto especificado (por defecto 4001), el agente se queda en espera a recibir el mensaje de comienzo de juego. Tras recibirlo el agente mandara la respuesta de listo, cuando haya procesado su turno. En cualquier caso transcurrido el temporizador asignado, la partida comienza recibiendo el mensaje de inicio de turno. Se recibirá un nuevo mensaje temporizador cada turno, mandando cada agente al final de su turno la acción a realizar. La partida llegará finalizará tras recibir el mensaje STOP y el cual será respondido por el agente. Tras acabar la partida, el agente se quedará a la espera de recibir un nuevo mensaje START con la información de la partida.

### 4.4.2. Parser

El módulo encargado en Jocular de interpretar los mensajes recibidos y convertirlos en estructuras internas que pueden usadas por el motor de inferencia es el *parser*. Las entradas del *parser* son los mensajes HTTP realizados por el *GameMaster*. El objetivo del *parser* es interpretar estos mensajes en información relevante para la aplicación. Esta información relevante pueden ser, la información de inicio de partida (temporizadores, roles de jugadores ...) o la información actual del juego para que sea actualizada. El *parser* esta enlazado con el motor de inferencia, de forma que la salida del *parser*, consiste en actualizar las estructuras que usará el motor de inferencia.

### 4.4.3. Motor de inferencia

El motor de inferencia y el *parser* están relacionados entre sí, ya que el motor usa la salida del *parser* para sus propósitos. Una vez el *parser* ha generado los objetos GDL se crea una base de conocimiento. Una **base de conocimiento** es el conocimiento estático que se posee: los hechos. Para identificar los hechos simplemente hay que identificar aquellas expresiones que sólo tengan variables y átomos. Si una expresión que contiene un hecho contiene una palabra clave de hecho (`true`, `init`...) hablamos de un hecho normal y si no hablamos de un **hecho base**. Los hechos base representa conocimiento estático e inalterable durante el transcurso de la partida, como por ejemplo la secuencia de turnos. Todo hecho que no éste en la base de conocimiento se considera falso.

Después se definen las implicaciones como las reglas que dados unos antecedentes implican unos consecuentes. Estas implicaciones se ayudan de ciertas estructuras auxiliares para representar negaciones, intersecciones o conjunciones de hechos. Para representar el conocimiento volátil de los hechos existe una estructura simular al conocimiento base llamada *proof knowledge*. Esta estructura representa los cambios existentes en el conocimiento en cada momento. Por ello cuando se realiza una inferencia siempre se asocia dos conocimientos uno estático, conocimiento de base, para representar el estado actual y otro sobre el que realizar los cambios, denominado también *proof contex*.

Para poder comprobar que dos expresiones son iguales se unifican entre sí. Para ello se ha implementado la funcionalidad de unificar expresiones GDL. El motor de inferencia, permite poner en verdadero todos los hechos existentes. Además permite obtener las acciones posibles, o sólo las legales, y ejecutarlas para obtener el siguiente estado.

Una de las ventajas del motor de inferencia usado es su generalidad, como se puede ver en el diagrama de clases expuesto. Esto se debe a que se puede realizar cualquier *pregunta* sobre la base de conocimiento y obtener las reglas existentes. La ventaja de esto es que

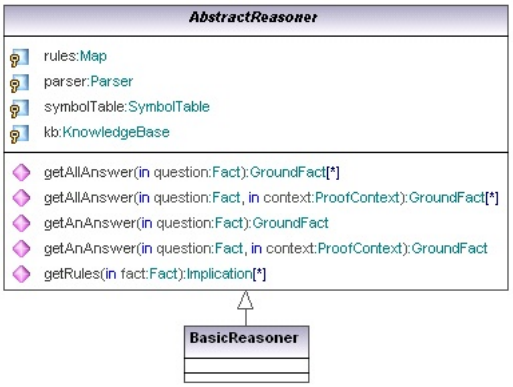


FIGURA 4.7: Diagrama de clases del M. Inferencia.

se puede comprobar el comportamiento de una regla sin necesidad de ejecutar todas las reglas (hay reglas que siempre se ejecutan para pasar de un estado al siguiente). No obstante es necesario realizar métodos más específicos para realizar búsquedas dentro del árbol de búsqueda.

### 4.4.4. Núcleo del agente

Una vez se tienen las reglas del juego y la información adicional representada en estructuras y elementos para poder usarlas ya se puede crear el agente. La función básica del núcleo del agente es seleccionar una acción dado un estado actual. Para ello podemos dividir el componente en subcomponentes, cada uno de ellos representando una funcionalidad diferente. En el siguiente punto se explica con más detalle el funcionamiento del núcleo, el cual se basa en los siguientes subcomponentes:

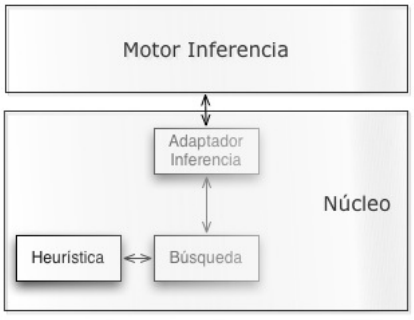


FIGURA 4.8: Diagrama de componentes del Núcleo.

- **Adaptador inferencia**: Como vimos en el punto anterior es necesario realizar métodos para poder realizar de una forma más sencilla la exploración del árbol de búsqueda. Para eso se ha desarrollado este componente.
- **Búsqueda**: El subcomponente de búsqueda se encarga de encontrar, dado un estado, la mejor acción a realizar dentro de un tiempo determinado. Para ello, es preciso, necesita usar una función heurística.
- **Heurística**: Este componente se encarga de obtener una función heurística a partir de otros subcomponentes. Este subcomponente puede realizar simulaciones y búsqueda de patrones dentro del código para buscar información adicional necesaria para generar la función heurística.

Si bien estos componentes definen el agente realizado, no tienen por qué estar realizados de forma completa debido a que su realización, en algunos casos, excede la complejidad propuesta en los objetivos del proyecto. Para entender mejor la estructura del núcleo del agente en el siguiente punto se explica el diagrama de clases.

## 4.5. Diagrama de clases

Gracias al diagrama de clases en UML podemos plasmar la estructura del sistema y las relaciones entre clases. La idea de este diagrama es hacerse una idea de las relaciones entre clases y su composición. Se ha decidido plasmar en el diagrama sólo la parte referida al núcleo de la aplicación 4.4.4 como se puede ver en la página 59, ya que es la parte que suele ser diferente en cada tipo de agente. Además es la parte que menos código se ha reutilizado/modificado del agente Jocular de Standford.

Como se puede ver en el diagrama 4.9 de la página 61, el núcleo del sistema se compone de una clase que ejecuta la aplicación y un conjunto de clases que implementan la lógica del agente. A continuación se describe cada clase:

`Main` El objetivo de esta clase es servir de punto de arranque del agente, dejándolo a la escucha del mensaje de inicio de partida. Además de poder configurar mediante parámetros el inicio del agente, permite configurar el agente para que realice diferentes tipos de búsqueda. Por defecto, el puerto donde se coloca el agente a escuchar es el 4001. Además contiene una referencia a la clase *GameManager* encargada de arrancar el jugador y darle la información leída por el *parser*.

`Parser` Tal y como se explico en la sección 4.4.2, página 58, en elementos que pueden ser usados por el motor de inferencia. El *parser* se comunica a través del *GameManager*


FIGURA 4.9: Diagrama de clases.

con la tabla de símbolos y de esa forma es capaz de preparar las reglas del juego para poder interpretables.

`SymbolTable` La tabla de símbolos almacena todos los posibles elementos que se usan en las reglas de un juego y los coloca en una tabla *hash* de forma que su acceso sea rápido. Sirve tanto para el *parsing*, como para las operaciones de unificación y para dar la información necesaria a cada agente.

`GameManager` Esta estructura tiene la finalidad de procesar y almacenar las diferentes partidas que tienen lugar durante la ejecución de un agente. No sólo es capaz de crear el núcleo de un agente para que se ejecute, sino que además almacena las partidas finalizadas. Almacena la instancia del jugador actual en la variable `gamer`. Cuando recibe un mensaje del *GameMaster*, usa el *parser* para obtener la información y actuar en consecuencia, creando, finalizando o ejecutando un turno de una partida.

`Gamer` Esta clase representa el motor de búsqueda del núcleo del agente 4.4.4, página 59. Es una clase abstracta que contiene los métodos necesarios para la ejecución de un agente. Almacena los roles del resto de jugadores de la partida, además del suyo propio. También almacena el estado actual de la partida que se actualizará según pasen los turnos. Para que el agente sepa el tiempo que tiene para realizar el aprendizaje o cada turno tiene dos variables con el tiempo en segundos y un

método `stopIt` para parar los temporizadores. Esta clase provee varios métodos públicos para realizar la partida, cada uno de ellos asociado a un mensaje enviado por el *GameMaster*:

- `initializeGame`: Este método se encarga de iniciar el agente, configurando los temporizadores, creando las variables necesarias para buscar en el árbol de juego... Para ello se le ofrece una variable con la información referente al juego: reglas, roles, tiempos de juego...
- `play`: Este método devuelve una acción a tomar dado un conjunto de acciones pasadas. Cada vez que se termina un turno y se recibe el nuevo mensaje `play`, este contiene los movimientos de todos los jugadores en el último turno. El método `play` debe actualizar el estado del juego con estos movimientos y dar una nueva acción para el nuevo estado del juego. Este método usa al método `moveThink` que se encarga de obtener la nueva acción en función de los movimientos previos.

`Skeleton` Esta clase extiende la lógica de `Gamer` y además sirve de base al resto de clases ya que crea métodos sencillos para explorar el árbol de búsqueda. Tiene un temporizador encargado de medir el tiempo de los turnos y dos variables para guardar el mejor movimiento hasta el momento encontrado por el algoritmo de búsqueda y su valor. Como los métodos provistos por el motor de inferencia eran muy complicados, se han creado una serie de métodos para facilitar las operaciones de exploración en el árbol de búsqueda. Los métodos creados son:

- `CalculteTerminal`: En caso de ser un estado final, devuelve el valor del mismo y la información del mismo.
- `getCost`: Dado dos estados obtiene el coste que existe entre ellos, es decir se obtiene $g(x)$.
- `getLegalMoves`: Devuelve la lista de todos los movimientos posibles en un estado.
- `getNextMove`: Obtiene el siguiente estado partiendo de un estado previo y las acciones ejecutadas previamente.
- `getNextRules`: Obtiene la lista de reglas `next`, es decir aquellas que se ejecutan automáticamente, como se puede ver en la sección 4 de la página 8.
- `getOtherMoves`: Devuelve la lista de la combinación de movimientos que pueden realizar el resto de jugadores, tras realizar nuestro agente una acción en un estado dado. Esto significa que se da una lista de todas las combinaciones entre movimientos del resto de jugadores. Por ejemplo, si un jugador tiene 5 posibles movimientos y otro 3, en total habría 15 movimientos.

- `isTerminal`: Devuelve `true` si el estado actual es terminal, y `false` en caso de no serlo.

`Heuristic` Esta clase engloba la lógica de generación de la heurística. Tiene una variable para guardar cada uno de los atributos que usa, como se explicó anteriormente en la sección 4.1.2.2, página 41. Esta clase funciona como el patrón `Singleton`, ya que solo permite crear una instancia en toda la ejecución. De esta forma se puede usar no sólo para almacenar los datos de las variables mediante las simulaciones sino obtener el valor de la heurística en función del estado actual, de si el nodo es terminal, del factor de ramificación o de la profundidad del mismo. Se pueden configurar las constantes (que ponderan las funciones de la función heurística) almacenadas en la variable `alpha` de la función heurística gracias al método `actualizarVariables`.

`Hex` Representa la lógica del agente creado, es decir esta clase es realmente el núcleo de nuestro agente. Esta clase se ejecuta en dos pasos. Tras recibir la información de inicio de la partida, ejecuta el simulador hasta que el tiempo de inicio de partida ha finalizado. En ese momento se indica al simulador que almacene la información obtenida en la función heurística. Después, cada vez que recibe un movimiento ejecuta la función adecuada, para juegos contra la naturaleza usa la clase `LRTA` y para clases multijugador usa la clase `CN`.

`Simulador` Esta clase realiza las simulaciones antes del comienzo de la partida, para obtener la información de la misma y poder ajustar la función heurística. Tiene prácticamente las mismas variables que la función heurística, ya que al acabar el tiempo de ejecución mediante el método `guardarDatos` almacena estas variables en la función heurística. Las simulaciones consisten en explorar el árbol de búsqueda lo mejor que se pueda e ir midiendo diferentes factores, profundidad, ramificación...

`CN` La clase `CN` representa el algoritmo de búsqueda de los números conspiratorios. Recordando dicho algoritmo en la sección 2.3.6, página 23 , muestra la robustez de cada nodo. Por eso lo que se indica en la variable de estabilidad es si lo que se desea buscar es el nodo más estable o el nodo más inestable (por defecto el más estable). Además están las variables que usa el algoritmo para funcionar, como el nivel de cada nodo, el umbral hasta donde explorar o una referencia a la función heurística. El algoritmo se inicia con el `cnAgent` que su vez cada turno llama a la búsqueda `cnSearch` para encontrar la solución.

`LRTA` Como el algoritmo LRTA* expuesto anteriormente, esta clase permite al agente realizar esta búsqueda. Para ello es necesario almacenar una tabla *hash* con los costes de cada nodo usado con el valor de su función heurística actual. Para ejecutar la búsqueda se ha creado el `lrtaAgent` que dado el estado actual obtiene la acción a tomar.

Además de estas clases descritas, están todas las usadas para describir tanto el motor de inferencia, como el *parser*. No obstante debido a que se reutilizan las clases creadas en Jocular y que no forman parte del núcleo de la aplicación se considera que la explicación hecha en la arquitectura 4.4, página 56, es suficiente .

## 4.6. Manual de Usuario

El manual de usuario explica los pasos necesarios para instalar y ejecutar el agente creado para la competición GGP y un ejemplo de su funcionamiento. Además se explicará el uso del simulador de la universidad de Dresden, que sirve para realizar y ejecutar las pruebas. Para realizar el manual de usuario se ha usado Mac OS X 10.5, pero las explicaciones son igualmente válidas para otros sistemas operativos.

### 4.6.1. Requisitos mínimos

Para poder ejecutar el agente es necesario tener un sistema que cumpla al menos las siguientes características:

- Permitir conexiones de entrada y salida HTTP por un puerto designado por el usuario.
- JVM 6 [5] o superior para ejecutar el **simulador**.
- JVM 5 o superior para ejecutar el **agente**.
- Permisos de ejecución en el directorio donde se vaya a ejecutar la aplicación.
- Interfaz gráfica para ejecutar el simulador.

En los requisitos no se ha especificado sistema operativo, se ha probado sobre *Windows XP* y *Mac OS X 10.5*. No obstante dada la independencia de la plataforma de Java, no debería existir ningún problema a la hora de instalar la JVM en otro sistema operativo [6].

[5] Java Virtual Machine

[6] En sistemas menores a Mac OS X 10.5 o sin doble núcleo no se permite la instalación de la JVM 1.6. No obstante se puede usar la siguiente url para hacerlo http://landonf.bikemonkey.org/static/soylatte/

### 4.6.2. Instalación del agente y creación de una partida

Si bien el simulador no forma parte del proyecto, es necesario a la hora de realizar pruebas sin poder contar con una conexión a internet. Por ello, en este apartado se explica primero como instalar el simulador, para a continuación hacer lo mismo con el agente para crear y realizar una partida.

1. El primer paso consiste en comprobar que Java está en el sistema. Para ello se abre una consola de comandos y se ejecuta el comando `java -version`, en caso de que la versión de java mostrada no sea igual o superior a la 1.6 es necesario instalarla [7]. No obstante en el manual de refencia se explica como instalar java en el sistema operativo Windows XP 4.7.1.1, página 69.

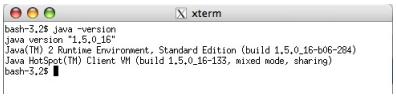


FIGURA 4.10: Comprobar la versión de java.

2. Después de comprobar que el simulador funciona correctamente es necesario bajarse tanto el simulador [8] (GUI Versión ya que es la que tiene interfaz gráfica), como una definición de un juego. En nuestro caso nos bajaremos la definición del Tic-Tac-Toe, ya que es un juego para varios jugadores [9].

3. Una vez hecho esto es necesario ejecutar el simulador para dejar lista la partida. Para ello se ejecutará con el comando `java -jar gamecontroller-gui-1.1.jar`, seleccionando el lugar donde se encuentra el archivo .jar del simulador.

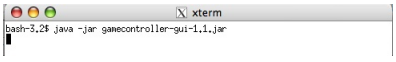


FIGURA 4.11: Ejecutar el simulador.

4. Una vez hecho, se busca con la ventana emergente el archivo donde estaba la definición del juego y se selecciona el botón de *Start Game*.

5. Tras seleccionar el juego, se muestra una ventana donde se puede elegir el nombre de la partida, los temporizadores de inicio de partida (`StartClock`) y de turno (`PlayClock`) en segundos y los jugadores de la misma.

[7] http://java.sun.com/javase/downloads/index.jsp
[8] http://www.general-game-playing.de/downloads.html
[9] http://www.general-game-playing.de/game_db/doku.php?id=games:tictactoe

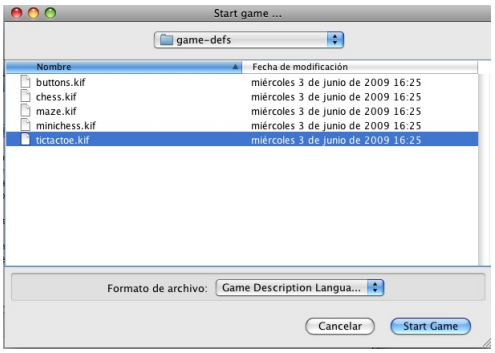


FIGURA 4.12: Seleccionar las reglas del juego.

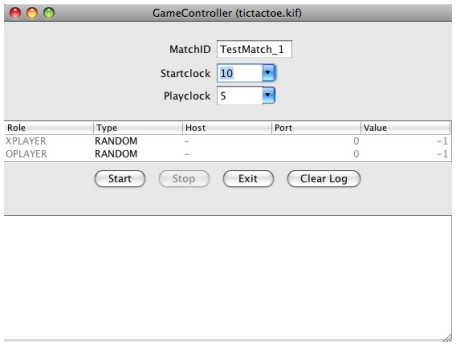


FIGURA 4.13: Configuración del simulador.

En la lista de jugadores aparece cada jugador y el rol asignado (en función de las reglas del juego asignadas). Los valores `Host` y `Port` son para configurar al agente en caso de elegir un tipo remoto. Para ello es necesario pulsar sobre el tipo de agente en la columna `type` y seleccionar el tipo REMOTE. Los tipos posibles de agentes son:

- **RANDOM**: Realiza sus movimientos de manera aleatoria sin ninguna inteligencia.
- **LEGAL**: Selecciona siempre el primer movimiento de entre la lista de los posibles movimientos a realizar.
- **REMOTE**: Con esto indicamos al simulador que es un jugador externo. De esta forma se puede configurar el puerto y la IP del mismo para ejecutarlo de forma remota.

Por eso se elige el tipo REMOTE y se configura el puerto y la IP donde se va a ejecutar el agente. Como en nuestro caso se ejecutará en la misma máquina que el simulador, la pantalla de configuración quedaría de la siguiente forma:

Una vez configurado el simulador y antes de empezar la partida, se debe instalar y ejecutar el agente.

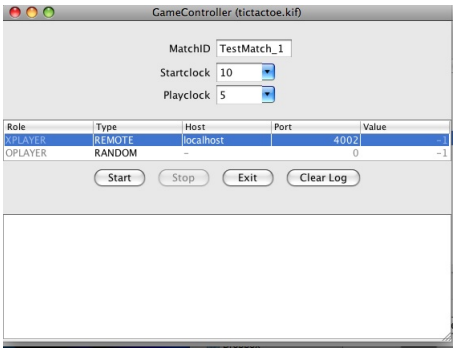


FIGURA 4.14: Preparado el simulador.

6. Para ejecutar el agente se ha de colocar el mismo en el lugar donde se va a ejecutar el fichero **hex.jar** adjunto en la memoria. Para ejecutarlo es necesario que este en la misma IP (misma máquina) y puerto que el simulador. En nuestro caso al ejecutarse localmente, ambos están en *localhost* y se ejecutarán en el puerto 4002. Para configurarlo existen una serie de parámetros. La suma de los valores de los tres pesos, tanto explícitos como por defecto debe ser 1. Los parámetros que deben ejecutarse son (todos van precedidos con dos líneas –)

   - `-daemon`: El agente se ejecuta en modo demonio, es decir en segundo plano (background).
   - `-port=X`: Ejecuta al agente en el puerto X. Por defecto el agente se ejecuta en el puerto 4001.
   - `-a1=p1`: Indica el valor del peso p1 de la función heurística, siendo su valor por defecto 0.3. Corresponde a la subfunción relacionada con el factor de ramificación.
   - `-a1=p2`: Indica el valor del peso p2 de la función heurística, siendo su valor por defecto 0.3. Corresponde a la subfunción relacionada con el factor de profundidad.
   - `-a1=p3`: Indica el valor del peso p3 de la función heurística, siendo su valor por defecto 0.4. Corresponde a la subfunción relacionada con la diferencia de hechos de los estados.

   De esta forma la ejecución del agente quedaría del siguiente modo:

7. Una vez ejecutado el agente sólo queda iniciar la partida seleccionando el botón `start` del simulador y espera a la finalización de la misma. Se puede ver en la ventana del simulador como se van mandando mensajes a cada agente, la hora de envío y el resultado de los mismos. Cuando en el simulador aparezca la palabra

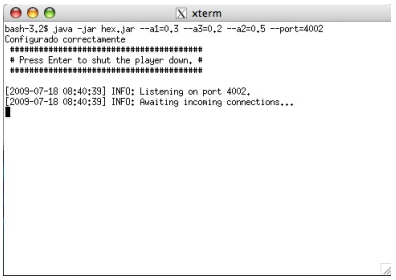


FIGURA 4.15: Ejecución del agente.

`Done`, habrá terminado la ejecución. En la columna `value` aparecen los resultados de cada partida, siendo un 100 una victoria total, un 50 un empate y 0 una derrota.

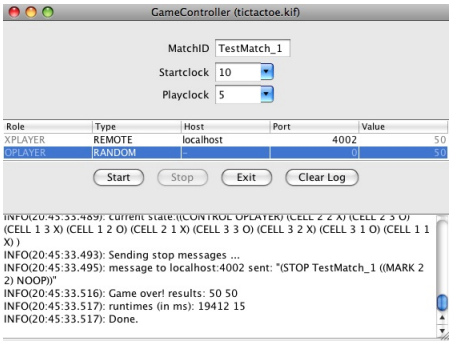


FIGURA 4.16: Final de la partida.

Para una mejor comprensión de los mensajes enviados y recibidos por el agente y el simulador que aparecen en la ventana, consultar la definición de los mismos (Love et˜al., 2008). Por último para reiniciar la partida, simplemente se debe pulsar el botón `start`. Si se quiere borrar el registro de mensajes `clear log`. Y para elegir otro juego, volver a usar la pantalla de selección de reglas del juego 4.12, página 66.

## 4.7. Manual de Referencia

Este manual de referencia sirve como punto de partida para mejorar el agente. En este apartado no se va a explicar ni la arquitectura 4.4, página 56, ni el diseño del agente 4.5, página 60. Lo que se va a detallar en este apartado es la forma de ampliar el agente o partes de este.

### 4.7.1. Instalación del entorno de trabajo

El lenguaje usado para desarrollar la aplicación es **Java** en la versión 1.5 mediante la plataforma de desarrollo **Eclipse** [10]. Primero se va a explica la configuración de Java en un sistema operativo, seguido de la configuración e instalación del código del agente en el entorno de trabajo Eclipse.

#### 4.7.1.1. Instalación de Java

Si bien la aplicación puede desarrollarse en cualquier sistema operativo, dada la alta capacidad técnica de los usuario de sistemas Unix y que en Mac OS 10.x viene instalado por defecto Java, sólo se va a explicar la instalación de Java en Windows XP.

Para instalar Java, se debe bajar desde la página web de Sun el JDK 1.5 [11] [12]. Después de descargar el fichero de instalación se pulsará sobre él, apareciendo la página de aceptación de la licencia, sobre la que pulsaremos en el botón de aceptar.

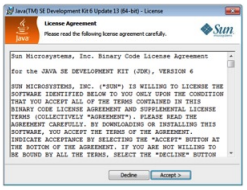


FIGURA 4.17: Licencia de Java.

Tras aceptar la licencia, se debe indicar la ruta donde instalar el JDK. Una vez hecho esto y siguiendo los pasos que indique el programa de instalación, estará instalado Java en el sistema.

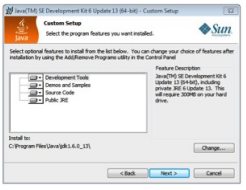


FIGURA 4.18: Seleccionar carpeta para instalar el JDK.

Aunque ya esta instalado Java en el sistema es necesario que el sistema reconozca la configuración del mismo en todas partes. Por ello se debe pulsar con el botón derecho

[10] www.eclipse.org
[11] Java Developer Kit
[12] http://java.sun.com/javase/downloads/index.jsp

sobre el icono de *Mi PC* e ir a propiedades del sistema. En la pestaña de opciones avanzadas, seleccionar variables de entorno añadiendo las siguientes variables:

- **PATH**: Debe tener como valor la ruta donde este instalada la carpeta bin de java.
- **CLASSPATH**: Debe contener el valor **.** (Un punto).

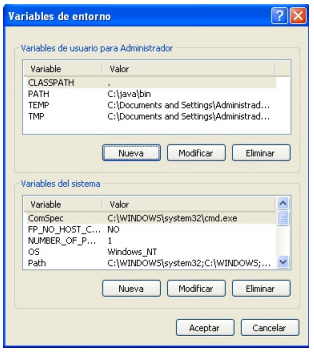


FIGURA 4.19: Configurar variables globales.

#### 4.7.1.2. Instalación del código en Eclipse

Una vez instalado Java es necesario descargar Eclipse de la url indicada anteriormente. Una vez hecho esto y descomprimido adecuadamente el archivo, se debe ejecutar el archivo con nombre `eclipse`. Una vez abierto Eclipse es necesario importar el proyecto del agente. Para ello se pulsará con el botón derecho sobre el explorador de proyectos seleccionando la opción de importar.

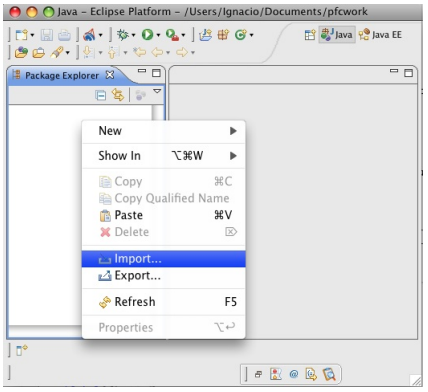


FIGURA 4.20: Importar un proyecto eclipse 1/3.

Después de seleccionar la opción importar es necesario indicar qué clase de proyecto deseamos importar. En nuestro caso queremos importar un proyecto existente en el *workspace*. Por último hay que seleccionar la opción de importar un archivo, seleccionando el

fichero **Hex.zip**, y apareciendo una ventana como la que se muestra a continuación. De esta forma Eclipse estará listo para desarrollar o modificar un nuevo agente.

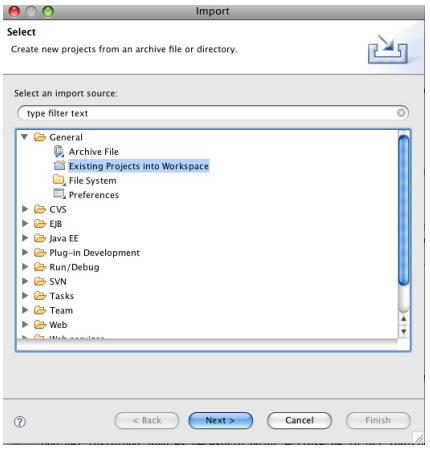


FIGURA 4.21: Importar un proyecto eclipse 2/3.

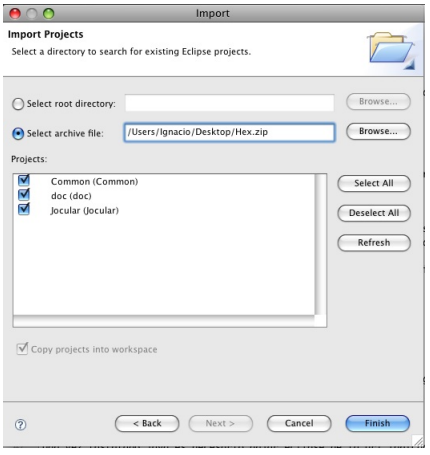


FIGURA 4.22: Importar un proyecto eclipse 3/3.

Como se puede ver en el explorador de proyectos de Eclipse, existen tres proyectos diferenciados:

- **Common**: Librerías usadas para desarrollar cualquier agente. Contienen las librerías relacionadas con el *parser*, el módulo de comunicación y el motor de inferencia.
- **doc**: Documentación asociada a la librería anteriormente expuesta.
- **Jocular**: Usa los dos proyectos anteriores para realizar el agente, es decir es una base para desarrollar el agente.

### 4.7.2. Crear nuevo agente

Para crear un agente se puede partir del código ya realizado e irlo modificándo (tal y como se va a describir en este apartado), o se puede crear un agente desde cero si no se desea usar código Java. En ese caso la solución más sencilla es acudir a la sección de enlaces de Dresden [13], donde vienen detallados diferentes agentes basados en otros lenguajes para ser usados de partida.

Para poder crear un agente basándonos en el código del agente que se presenta en este proyecto, se debe tener en mente la arquitectura del mismo detallada en la sección 4.4 en la página 4.4. Además hay que ser consciente de que existe un tiempo de aprendizaje de las reglas, de ahí la estructura del agente Hex. Para realizar el agente, se debe crear una clase que extienda la clase `Skeleton` ya que tiene los métodos necesarios para explorar el árbol de búsqueda. Una vez hecho esto hay que sobreescribir el método `moveThink` ya que es el encargado de realizar el movimiento en el nuevo estado creado.

Para mejorar la búsqueda se puede hacer referencia a la clase `Heuristic`, que genera la función heurística en función del estado. Por ello en el tiempo de aprendizaje es necesario realizar simulaciones de las reglas para obtener datos e información de las mismas.

En resumen, los pasos a seguir para crear un agente son los siguientes:

1. Extender la clase `Skeleton`.
2. Realizar simulaciones en el tiempo de aprendizaje usando para ello un temporizador.
3. Usar la clase `heuristic` para obtener la función heurística enlazándola con el simulador.
4. Sobreescribir el método `movethink` para obtener el siguiente movimiento.

### 4.7.3. Mejora de la función heurística

La función heurística esta en la clase `heuristic`. Las mejoras que se pueden realizar a la función heurística parten de dos puntos diferentes. Crear nuevas subfunciones o optimizar los pesos de las mismas.

[13] http://www.general-game-playing.de/downloads.html

#### 4.7.3.1. Añadir nuevas subfunciones heurísticas

Cuando se explicó el agente y la clase heurística 4.1.2.2, explicada en la página 41, se indicó que esta se componía de subfunciones que eran multiplicadas por unos pesos. De esta forma se conseguía que la función estuviera dentro de un rango, y a la vez se diera importancia a las diferentes subfunciones. Para añadir nuevas subfunciones a la función heurística se deben realizar los siguientes pasos:

1. Crear un método, que devuelva un valor entre 0 y 100, siendo 100 el mejor valor posible. Se aconseja crear este método dentro de la clase de la función heurística para tener cerca los valores de la simulación, aunque no es estrictamente necesario.
2. Una vez creada la subfunción, es necesario aumentar en una unidad el tamaño del array de pesos de la subfunción asignándole un valor. Para ello se debe ir al constructor de la clase `heuristic` y asignar un valor por defecto que sume en total con el resto de valores 1.
3. A la función `getHeuristic` se le debe añadir una llamada al método nuevo multiplicada por el valor del peso asignado.
4. Por último es importante modificar la clase `main`, para asignar el nuevo peso a la función heurística.

#### 4.7.3.2. Optimización de los pesos de la función heurística

La clase `Pruebas`, dentro de Jocular sirve para generar un gran número de ejecuciones con el simulador para probar el agente. Usando estas simulaciones, se pueden ir modificando los pesos, mediante diferentes técnicas de optimización de parámetros para encontrar unos que mejoren la función heurística.

### 4.7.4. Modificar las simulaciones

Como se explicó en el diagrama de clases se obtienen diferentes variables para crear la función heurística 4.5. Las simulaciones realizan una exploración sobre el árbol de búsqueda modificando dichas variables. Para ejecutar las simulaciones se debe crear la clase `Simulacion`, usar el método `initialize`, y ejecutar usando el método `play` hasta que se acabe el tiempo de aprendizaje. Para obtener más variables que se puedan usar en la función heurística simplemente hay que modificar el método `play`, que es donde se recopila toda la información del resto de variables.

### 4.7.5. Modificar el parser y/o motor de inferencia

El *parser* y el motor de inferencia están en el proyecto `Common`. Es complicado modificarlos independientemente ya que están muy entralazados entre ellos. El paquete `GDL` contiene todos los elementos del *parser*, mientras que el paquete `knowledge` y `prover` contienen la información del motor de inferencia. La mejor forma de modificarlos es modificar las clases de estos paquetes y usar los paquetes de test asociados a cada uno de éstos.

# Capítulo 5

# Resultados y Pruebas

Una vez realizado el agente es importante comprobar como es de bueno. Para ello se ha diseñado una batería de pruebas para comprobar la eficacia del agente. Después de pasar la batería de pruebas, el agente deberá realizar su objetivo inicial que es participar en la competición GGP. Para ello se incluirá un segundo apartado en este capítulo mostrando los resultados de la competición y el desarrollo de la misma.

## 5.1. Pruebas

En este apartado se detalla un conjunto de pruebas, para comprobar la validez del agente creado. Los agentes que se usarán en las pruebas son:

- **LegalPlayer**: Ejecuta siempre el primer movimiento permitido en cada estado.
- **RandomPlayer**: Este jugador ejecuta movimientos aleatorios, dentro de los que puede realizar.
- **Minimax**: Implementa el algoritmo minimax, pero sin ninguna función heurística.
- **Hex**: Es el agente implementado en este proyecto.

Las pruebas que se realizaran van en función de los tipos existentes de juegos. Las pruebas se han inspirado en las realizadas en la competición de 2009:

- **Juego para 1 jugador**: En este caso será el **mundo de los bloques** [1]. Se realizarán 10 pruebas por cada jugador.

[1] http://www.general-game-playing.de/game_db/doku.php?id=games:blocks

- **Juego para 2 jugadores**: En este caso será el **tic-tac-toe** [2]. Se realizarán 10 pruebas por cada oponente.
- **Juego para 4 jugadores**: En este caso será el **othello 4** [3]. Se realizarán 10 pruebas con todos los jugadores.

Para realizar las pruebas se ha creado una clase dentro del paquete Jocular llamada Pruebas, que realiza un bacth de las mismas. Los resultados de las pruebas son:

- **Mundo de los bloques**: Como se puede ver en las pruebas del mundo de los bloques, nuestro agentes es el único que gana la mitad de las partidas. El resto de agentes no gana ninguna, ya que en juegos de un solo jugador dependes exclusivamente de su validez como jugador y no de la incapacidad de los rivales. Es importante destacar que la mitad de las partidas han acabado en derrota, porque la función heurística no está perfeccionada, pero al menos, a la luz de los resultados obtenidos el agente funciona adecuadamente.

| Nombre | Victoria | Derrota | Eficiencia |
|---|---|---|---|
| Hex | 5 | 5 | 0.5 |
| Random | 0 | 0 | 0 |
| Legal | 0 | 0 | 0 |
| MiniMax | 0 | 0 | 0 |

Cuadro 5.1: Pruebas en el mundo de los bloques

- **Tic-Tac-Toe**: Es un juego para dos jugadores, por lo que depende también de la capacidad del rival para ganar la partida.

| Tipo | Legal | Random | Minimax |
|---|---|---|---|
| Partidas | 10 | 10 | 10 |
| Victorias | 9 | 5 | 4 |
| Derrotas | 1 | 3 | 3 |
| Empates | 0 | 2 | 3 |
| Porcentaje Exito | 0.9 | 0.5 | 0.4 |
| Media | 90 | 60 | 55 |
| Total | 900 | 600 | 550 |

Cuadro 5.2: Pruebas en el tic-tac-toe

Como se puede ver en la gráfica a medida que aumenta la capacidad de decisión del agente, más difícil es ganarle. Aunque parece que se ha perdido contra el minimax sin función heurística, en realidad se puede considerar que se ha obtenido

[2] http://www.general-game-playing.de/game_db/doku.php?id=games:tictactoe
[3] http://www.general-game-playing.de/game_db/doku.php?id=games:othello-fourway

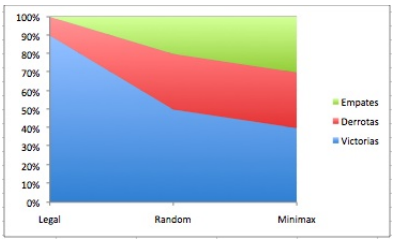


FIGURA 5.1: Pruebas en el tic-tac-toe.

una buena puntuación ya que nuestro agente ha ganado 4 de las 10 partidas y empatado 3 partidas, perdiendo sólo una. No obstante estos resultados demuestran que se necesita mejorar la función heurística, para poder jugar de una manera más competitiva.

- **Othello 4**: Es un juego para 4 jugadores, donde pueden ganar varios simultáneamente. En los resultados obtenidos se ve una clara tendencia de obtener un mayor número de puntos a medida que el algoritmo obtiene más complejidad del entorno. Como se puede ver, todavía le falta a nuestro agente una clara ventaja respecto al resto de agentes. Esto se puede ver en la cantidad de victorias obtenidas por el jugador aleatorio en este juego y el anterior. Si nuestro agente hubiera tenido más información del entorno, probablemente, esta diferencia de puntos sería más clara. No obstante, con estos datos podemos decir que nuestro agente aporta una mejora suficientemente buena, para poder competir en la GGP *Competition*.

| Tipo | Hex | Legal | Random | Minimax |
|---|---|---|---|---|
| Partidas 10 | 10 | 10 | 10 | |
| Victorias 1 | 4 | 6 | 7 | |
| Derrotas 9 | 6 | 4 | 3 | |
| Empates 0 | 0 | 0 | 0 | |

CUADRO 5.3: Pruebas en el Othelo 4

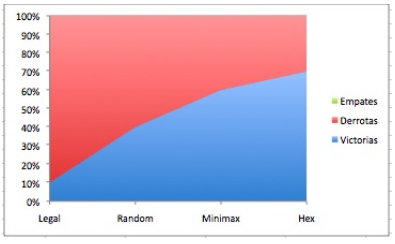


FIGURA 5.2: Pruebas en el othello 4.

Además de estas pruebas se han realizado una serie de pruebas de rendimiento, para obtener el número de nodos que puede explorar el agente. Para ello se ha contado para diferentes juegos el número de nodos explorados por el algoritmo de números conspiratorios, ya que al ser iterativo reexplora el árbol. El resultado promedio tras 40 ejecuciones es de aproximadamente **8500 nodos explorados en 3 segundos**. Si bien no es una cantidad muy alta, se puede incrementar hasta **15000 nodos** explorados **modificando** la forma de representación del motor de búsqueda. De ahí que una de las prioridades para mejoras del agente, sea cambiar el motor de inferencia del agente por otro más potente.

## 5.2. General Game Playing Competition 2009

Este año la competición se llevó a cabo en GIGA'09 *workshop* del IJCAI el sábado 11 de Julio. Siendo las pruebas clasificatorias las dos semanas previas. Desgraciadamente ha sido imposible participar en la competición este año debido a que por primera vez se exige para toda la competición, y no sólo la ronda final, que los jugadores se ejecuten de forma local. No obstante se han recogido los resultados de las pruebas clasificatorias, que sí se pudieron ejecutar a distancia, siendo los siguientes:

| Posición | Nombre | Puntuación | Partidos Jugados | Partidos sin error | Tasa error |
|---|---|---|---|---|---|
| 1 | CadiaPlayerlite | 70.1856 | 862 | 848 | 0.0162 |
| 2 | ary | 66.9859 | 924 | 899 | 0.0271 |
| 3 | TurboTurtle | 56.1768 | 164 | 136 | 0.1707 |
| 4 | Fluxplayer | 52.2034 | 875 | 678 | 0.2251 |
| 5 | centurio | 49.3675 | 721 | 647 | 0.1026 |
| 6 | Gamer | 31.0576 | 330 | 171 | 0.4818 |
| 7 | Hex | 0.1053 | 19 | 2 | 0.8947 |
| 8 | Ethan | 0 | 0 | 0 | 0 |
| 9 | Maligne | 0 | 0 | 0 | 0 |

CUADRO 5.4: Resultados de las pruebas clasificatorias GGP Competition 2009

Como se puede ver nuestro agente (Hex) no ha obtenido una gran puntuación y además un gran porcentaje de las ejecuciones han tenido fallos. Los fallos en la competición se deben a que el GameMaster de la universidad de Dresde, donde se realizo la competición, no sigue el estándar definido en esta memoria. Como se puede ver incluso un jugador como FluxPlayer ganador de una de las competiciones previas ha tenido un 0.2 de fallos. Además la mayor parte de esos fallos son problemas de conexión debido a que si no llega el mensaje a tiempo al servidor este ejecuta un movimiento al azar, pero lo notifica como fallo. No obstante como se ha demostrado en las pruebas, además del uso que se hizo de

la aplicación en el servidor de la universidad de Stanford http://games.stanford.edu/ han demostrado que la aplicación funciona correctamente.

En la competición final el ganador ha sido **ary** seguido de **FluxPlayer**. Como se puede ver en los resultados de la clasificación si bien CadiaPlayer ha obtenido bastante puntuación, como se esperaba por la ronda clasificatoria, no ha sido suficiente para conseguir una victoria.

| Posición | Nombre | Puntuación |
|---|---|---|
| 1 | ary-distant | 339 |
| 2 | Fluxplayer | 256 |
| 3 | Maligne | 255 |
| 4 | Centurio | 153 |
| 5 | Ethos | 318 |
| 6 | CadiaPlayer | 306 |
| 7 | Gamer | 261 |
| 8 | TurboTurtle | 143 |

CUADRO 5.5: Resultados de la GGP Competition 2009

# Capítulo 6

# Conclusiones

En este capítulo se analizarán las ventajas y desventajas encontradas a la hora de realizar este proyecto. Primero se realizará un repaso de los objetivos, comprobando en qué medida se han cumplido. Después se indicarán algunas conclusiones adicionales.

## 6.1. Objetivos realizados

A continuación se realiza un seguimiento de los objetivos vistos en la sección 3, página 33 y de las conclusiones obtenidas de los mismos una vez realizado el proyecto.

### 6.1.1. Desarrollar un sistema GGP

El objetivo del proyecto era desarrollar un sistema para jugar en la *General Game Playing Competition*. Además, este sistema tenía que servir de punto de partida para futuras ampliaciones. Lo primero que se realizó en el proyecto fue estudiar las aproximaciones que se habían realizado al problema de generar un sistema GGP. Después de ello se estudiaron técnicas y métodos para resolver el problema. Una vez recopilada suficiente información sobre cómo hacer un agente, se ha explicado detalladamente la estructura del mismo. Sabiendo como es la estructura de un agente, se puede usar como punto de partida para desarrollar otro. Además gracias al manual de referencia, se puede realizar una ampliación del agente desarrollado, con mucho menos esfuerzo.

### 6.1.2. Obtener una visión general de la competición

Además de estudiar competiciones previas, analizando a sus ganadores, se ha intentado participar en la competición de GGP. Se han estudiado competiciones pasadas, comprobando las formas de ganar de los ganadores. Además se ha investigado sobre diferentes formas de crear el agente para participar en la competición. Se creó un agente funcional, tal y como demuestran las pruebas, capaz de competir. Además se ha participado en las rondas clasificatorias de la competición 2009. El resultado ha sido bastante malo, pero se ha obtenido información sobre los ganadores de la misma, y experiencia para poder participar en futuras competiciones.

Además viendo los resultados de la competición podemos ver que cada año el ganador de la misma va siendo diferente, por lo que podemos deducir que no existe una única solución para resolver el problema.

### 6.1.3. Elección de algoritmo

Uno de los problemas principales a la hora de desarrollar el algoritmo ha sido la elección del algoritmo de búsqueda usado. Para ello se realizó un estudio de los algoritmos de búsqueda para usar en el agente. Después de elegir esos algoritmos, se explicó su elección. Además para cada algoritmo se intentó mejorar usando técnicas como reconocimiento de patrones en el caso de LRTA* y de simulaciones para los números conspiratorios. Además este último algoritmo ha demostrado en las pruebas, que aún siendo una perspectiva diferente a la hora de abordar el problema, también era válido. Además la suposición de que en sistemas con baja información, es mejor tener robustez en la solución, merecen un mayor enfoque para futuras investigaciones.

### 6.1.4. Participar en la competición con un agente competitivo

Como se ha ido explicando en los puntos anteriores se ha creado un agente lo suficientemente competitivo, como para participar en la competición GGP. Además si bien los resultados de las rondas clasificatorias fueron malos y no se pudo participar en la competición por problemas técnicos, se ha demostrado que se puede crear un sistema capaz de participar en dicha competición. El sistema creado tiene la posibilidad de ser expandido y mejorado en el futuro, por lo que se podrá participar de forma más competitiva en futuras competiciones.

### 6.1.5. Critica al lenguaje GDL

No sólo se estudió la competición y el lenguaje GDL, sino que además se criticaron todos los fallos existentes a la hora de formalizar todos los tipos de juegos. Para cada una de las críticas se encontraron mejoras, lo suficientemente buenas y realizables, que sería interesante ver realizadas. Incluso el problema de representar juegos de información incompleta, ha quedado resuelto mediante el uso de una nueva relación. Gracias a estas mejoras, es más probable que la GGP sea capaz de abarcar una mayor cantidad de juegos, cumpliendo con el propósito fundacional de dicha competición.

## 6.2. Conclusiones Finales

La realización de este agente ha servido de base para futuras mejoras y desarrollos de agentes. Además se ha ampliado el lenguaje GDL, de forma que se han creado las condiciones adecuadas para la creación de una competición de sistemas GGP que abarque una mayor variedad de juegos. El objetivo de desarrollar un agente competitivo, quizá era demasiado ambicioso para un único proyecto, pero este proyecto ha servido de base para conocer la estructura de un agente y sus carencias.

# Capítulo 7

# Lineas Futuras

Este proyecto sirve como punto de partida para crear un agente para la General Game Playing Competition. Además también sirve como punto de inicio y fuente de técnicas de inteligencia artificial genéricas. A continuación pasamos a detallar algunas de las posibles técnicas y líneas futuras que pueden partir del trabajo desarrollado.

## 7.1. Creación de una función heurística general

Uno de los problemas que nos hemos encontrado a la hora de usar algoritmos de búsqueda, es tener una buena función heurística que guíe la búsqueda. El problema de crear una función heurística para un sistema GGP, es que debe valer para cualquier posible situación y juego. Y ese es un gran problema, debido a que la técnica más usada para crear funciones heurísticas es la experiencia y crear técnicas especificas *ad hoc*.

La obtención de parámetros que se ha realizado en el agente 4.1.2.2 es un primer paso para obtener la función heurística, pero precisa de un estudio en profundidad. Este estudio podría consistir en encontrar relaciones entre dichos parámetros, nuevos patrones relevantes o el uso de técnicas de minería de datos. La técnica de reconocimiento de patrones, como la usada en el agente, puede ser útil también si se realiza un estudio en profundidad de las posibles estructuras existentes en las reglas de juego.

En definitiva, existe una gran cantidad de técnicas disponibles para obtener mejores funciones heurísticas que deben ser investigadas y perfeccionadas para poder ser usadas en un sistema GGP.

## 7.2. Ampliación del lenguaje GDL y la competición GGP

Como se comento en el apartado de la memoria dedicado a criticar el lenguaje GDL 2.2.2, el lenguaje y la competición no recogen toda la variedad de juegos disponibles. Si bien esto puede mejorarse mediante las soluciones aportadas en esta memoria, todavía quedan muchas mejoras por hacer. Existen juegos con un número infinito de acciones que no pueden ser representados en GDL. Si bien estas limitaciones pueden deberse a prioridades por parte de los creadores de la competición, es preciso no limitar el concepto de sistema GGP por la competición.

## 7.3. Estudio de funciones de búsqueda

Al ser la competición GGP tan abierta, permite el uso de infinidad de técnicas diferentes para encontrar solución al problema planteado. El problema radica en que no sabemos qué técnicas son las mejores para cada tipo de juego. Los juegos de suma nula o los simétricos pueden necesitar algoritmos de búsqueda diferentes que otra clase de juegos.

En nuestra solución se usarón los números conspiratorios como técnica de búsqueda. Aunque con está técnica no se han encontrado mejoras respecto a otros algoritmos usados, tampoco existen técnicas que funcionen correctamente cuando la función heurística da poca información. Por eso, un punto de investigación sería encontrar algoritmos que funcionen bien cuando la información que tenemos se refiere más a la estructura del problema y menos al problema en sí. En definitiva, se trata de encontrar algoritmos que funcionen correctamente incluso con poca información, o con información errónea.

# Apéndice A

# Pseudocódigos

## A.1. A*

En esta sección se expone de forma detallada el algoritmo A* 2.3.1.

**Algorithm 1** Algoritmo A*

```
1: procedure ASTAR(estado,meta)                          ▷ El estado actual y la meta
2:     listaCerrada                                      ▷ Nodos ya evaluados
3:     listaAbierta                                      ▷ Nodos a evaluar
4:     add(listaAbierta,estado)          ▷ Se añade el estado inicial a la lista abierta
5:     while len(listaAbierta) >0 do
6:         mejorNodo:=obtenerMejorNodo(listaAbierta,meta)
7:         if llegaANodoMeta(mejorNodo,meta) then        ▷ El algoritmo acaba si se
   puede llegar al nodo meta
8:             return generarCamino(mejorNodo,meta)
9:         end ifremove(listaAbierta,mejorNodo)
10:        add(listaCerrada,mejorNodo)
11:        nodos := getNodosDesde(mejorNodo)             ▷ Se exploran todos los nodos
12:        for all n ∈ nodos do
13:            valorG:=getDif(n,meta) + getGValue(n)
14:            if exist(listaAbierta,n) AND valorG <getValorActual() then
15:                remove(listaAbierta,n)
16:                add(listaCerrada,n)
17:            else
18:                if !exist(listaCerrada,n) AND !exist(listaAbierta,n) then
19:                    add(listaAbierta,n)
20:                    setGValue(n,valorG)
21:                end if
22:            end if
23:        end for
24:    end while
25: end procedure
```

## A.2. RTA*

En este apartado se detalla el pseudocódigo del algoritmo RTA* 2.3.2.El método `doPath` reconstruye haciendo retropropagación, la ruta para el algoritmo.

**Algorithm 2** Algoritmo RTA*

```
1: procedure RTA(estado, searchState)          ▷ El estado actual y el estado meta
2:     acciones                                                ▷ Array de acciones
3:     while !isEndTime() do
4:         if estado = searchState then
5:             return doPath(estado, searchState)
6:         end if
7:         states := getStates(estado)
8:         evs          ▷ Lista con los valores de la función de evaluación de cada estado
9:         for all s ∈ states do
10:            add(s,getHeuristic(s) + getCost(s, searchState))
11:        end for
12:        setHeuristic(estado, getHeuristic(getSecondBest(evs,states)) +
   getCost(getSecondBest(evs,states), searchState))
13:        RTA(getBest(evs,states))
14:    end while
15:    return getActionWithMinEvaluationValue(estado) ▷ Se toma el movimiento
   con menor valor
16: end procedure
```

## A.3. LRTA*

En esta sección se expone de forma detallada el algoritmo LRTA* 2.3.3. La función *getActionWithMinEvaluationValue* devuelve el menor valor posible.

**Algorithm 3** Algoritmo LRTA*

```
1: procedure LRTA(estado)                                   ▷ El estado actual
2:     acciones                                             ▷ Array de acciones
3:     states                                               ▷ Array de estados
4:     searchState,aux                                      ▷ Estados
5:     value := 0
6:     searchState := estado
7:     while !isEndTime() do
8:         acciones := getAllActions(estado)
9:         states := doAction(searchState)
10:        value = getHeuristic(state[0]) + getCost(state[0], searchState)
11:        for all s ∈ states do
12:            if getHeuristic(s) + getCost(s, searchState) <= value then
13:                aux := s
14:                value := getHeuristic(s) + getCost(s, searchState)
15:            end if
16:        end for
17:    end while
18:    return getActionWithMinEvaluationValue(estado)  ▷ Se toma el movimiento
   con menor valor
19: end procedure
```

## A.4. Minimax

En este apartado se detalla el pseudocódigo del algoritmo minimax 2.3.3.

**Algorithm 4** Algoritmo Minimax

```
1: procedure MINIMAX(estado)                                  ▷ El estado actual
2:     i, w := 0
3:     m, t := 0,0
4:     if isTerminal(estado) then
5:         return funcionEvaluacion(estado)
6:     end if
7:     w := generar(estado)
8:     m := −∞
9:     for i = 1 dow
10:        t := minimax(p_i)
11:        if (t > m)and(tipo(p) == alfa) then
12:            m := t
13:        end if
14:        if (t < m)and(tipo(p) == beta) then
15:            m := t
16:        end if
17:    end for
18:    return m
19: end procedure
```

## A.5. Algoritmo Alfa-Beta

Para esta implementación del algoritmo Alfa-Beta 2.3.4 se ha usado como base el algoritmo minimax.

**Algorithm 5** Algoritmo Alfa-Beta

```
1: procedure MINIMAX(estado)                                  ▷ El estado actual
2:     i, w := 0
3:     m, t := 0,0
4:     if isTerminal(estado) then
5:         return funcionEvaluacion(estado)
6:     end if
7:     w := generar(estado)
8:     m := −∞
9:     for i = 1 dow
10:        t := alfabeta(p_i, α, β)
11:        if (t > m)and(tipo(p) == alfa) then
12:            m := t
13:        end if
14:        if (t < m)and(tipo(p) == beta) then
15:            m := t
16:        end if
17:        if (m >= β) then
18:            return m
19:        end if
20:    end for
21:    return m
22: end procedure
```

## A.6. CN-Search

El algoritmo de los números conspiratorios que se explica a continuación viene detallado en la sección 2.3.6. Esta versión de algoritmo viene más detallada en (Schaeffer, 1990)

**Algorithm 6** Algoritmo CN

```
1: procedure CNSearch(estado,umbralMin,umbralMax,umbral)  ▷ El estado inicial, y los umbrales de partida
2:     while umbralMin <umbralMax do
3:         CNS = 0
4:         if esNodoHoja(estado) then                     ▷ El estado es nodo Hoja
5:             if esTerminal(estado) then
6:                 updateTerminal(CNS, calcularValorMiniMax(CN))  ▷ En el nodo terminal, todos los valores valen infinito, salvo el del nodo que es 0
7:             else
8:                 updateHoja(CNS, calcularValorMiniMax(CN))  ▷ Si es un nodo hoja los valores que se pueden tomar son: 0 si el valor es igual al umbral o 1 si no lo es
9:             end if
10:        else
11:            for all hijo ∈ estado do
12:                CNSearch(hijo,umbralMin,umbralMax,umbral)
13:            end for
14:        end if
15:        umbralMin = min(getDownSet(CNS,umbral))  ▷ Se actualizan los umbrales
16:        umbralMin = max(getUpSet(CNS,umbral))
17:    end while
18: end procedure
```

Las funciones auxiliares `getDownSet` y `getUpSet` indican que se obtiene del conjunto de números conspiratorios de un nodo los subconjuntos UP y DOWN.

# Apéndice B

# Reglas de Juegos

En esta sección se encuentra las reglas del Tic-Tac-Toe a modo de ejemplo de fichero GDL. Este código ha sido obtenido de la base de datos de juegos, `http://www.general-game-playing.de/game_db/doku.php`

## B.1. TicTacToe

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Tictactoe
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (role xplayer)
  (role oplayer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (init (cell 1 1 b))
  (init (cell 1 2 b))
  (init (cell 1 3 b))
  (init (cell 2 1 b))
```

```
  (init (cell 2 2 b))
  (init (cell 2 3 b))
  (init (cell 3 1 b))
  (init (cell 3 2 b))
  (init (cell 3 3 b))
  (init (control xplayer))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Cell

  (<= (next (cell ?m ?n x))
      (does xplayer (mark ?m ?n))
      (true (cell ?m ?n b)))

  (<= (next (cell ?m ?n o))
      (does oplayer (mark ?m ?n))
      (true (cell ?m ?n b)))

  (<= (next (cell ?m ?n ?w))
      (true (cell ?m ?n ?w))
      (distinct ?w b))

  (<= (next (cell ?m ?n b))
      (does ?w (mark ?j ?k))
      (true (cell ?m ?n b))
      (or (distinct ?m ?j) (distinct ?n ?k)))

  (<= (next (control xplayer))
      (true (control oplayer)))

  (<= (next (control oplayer))
      (true (control xplayer)))


  (<= (row ?m ?x)
      (true (cell ?m 1 ?x))
```

```
      (true (cell ?m 2 ?x))
      (true (cell ?m 3 ?x)))

  (<= (column ?n ?x)
      (true (cell 1 ?n ?x))
      (true (cell 2 ?n ?x))
      (true (cell 3 ?n ?x)))

  (<= (diagonal ?x)
      (true (cell 1 1 ?x))
      (true (cell 2 2 ?x))
      (true (cell 3 3 ?x)))

  (<= (diagonal ?x)
      (true (cell 1 3 ?x))
      (true (cell 2 2 ?x))
      (true (cell 3 1 ?x)))


  (<= (line ?x) (row ?m ?x))
  (<= (line ?x) (column ?m ?x))
  (<= (line ?x) (diagonal ?x))


  (<= open
      (true (cell ?m ?n b)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (legal ?w (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control ?w)))

  (<= (legal xplayer noop)
      (true (control oplayer)))

  (<= (legal oplayer noop)
      (true (control xplayer)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (goal xplayer 100)
      (line x))

  (<= (goal xplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal xplayer 0)
      (line o))

  (<= (goal oplayer 100)
      (line o))

  (<= (goal oplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal oplayer 0)
      (line x))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= terminal
      (line x))

  (<= terminal
      (line o))

  (<= terminal
      (not open))
```

# Bibliografía


Michael Genesereth and Nathaniel Love. General game playing: Overview of the aaai competition. *AI Magazine*, March 2005. URL http://games.stanford.edu/competition/misc/aaai.pdf.

Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. *AAAI*, March 2008.

Michel Goossens, Frank Mittlebach, and Alexander Samarin. Checkers is solved. *Science*, pages 1518–1522, July 2007.

The duel: Man vs machine (www.rag.de/microsite$_c hess_c om$). 2006.

Michael Buro. *IEEE Intell. Sys. J. 14, 12*, 1999.

David M. Kaiser. The design and implementation of a successful general game playing agent. *AAAI*, 2007.

Judea Pearl. *Heuristics*. Addison-Wesley, Reading MA, 1984.

Holger H.Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, USA, 2005.

Stuart Russel and Peter Norving. *Artificial Intelligence: A modern approach*. Pearson Education International, 1995.

Richard E. Korf. Heuristic search. January 2007.

E. Balas. A note on the branch-and-bound principle. *Operational Research*, 16(886):442–444, 1968.

David Allen McAllester. Conspiracy numbers for min-max search. 35:287–310, 1988.

Norbert Klingbeil and Jonathan Schaeffer. Empirical results with conspiracy numbers. *Computational Intelligence*, 6:1–11, 1990.

Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, November 1989.

Clune J. Heuristic evaluation functions for general game playing. *In Proceedings of the Twenty-Second AAAI Conference on Articial Intelligence*, pages 1134—-1139, 2007.

Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. *Student Paper*, 2007.

Hilmar Finnson. Cadia-player: A general game playing agent. Master's thesis, Reykjavík University - School of Computer Science, December 2007.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. 2006.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. 27: 97–109, 1985.

J. D. Williams. Kriegsspiel rules. 1950.

Jonathan Schaeffer. Conspiracy numbers. *ai*, 43:67–84, 1990.