UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA EN INFORMÁTICA

Inteligencia Artificial e Ingeniería del Conocimiento

Práctica 2: Búsqueda entre adversarios

Jose Domingo López López
josed.lopez1@alu.uclm.es

28 de Mayo de 2009

Este documento fue compuesto con LaTeX. Imágenes generadas con The GIMP.

Índice general

Capítulo 1

Introducción

Para comenzar con este documento se explicará en qué consiste el problema que abordaremos y cuáles son las decisiones de diseño planteadas, así como las estructuras de datos que se utilizarán para la resolución del mismo.

1.1. Definición del problema

El problema consiste en realizar un bot (agente racional) que pueda jugar de forma autónoma contra un adversario en **El Juego de la Bandera**, permitiendo la posibilidad de realizar una pequeña competición. La estrategia de juego será dirigida por un algoritmo Mini-Max con poda alfa-beta.

Se puede encontrar una especificación más detallada de en qué consiste **El Juego de la Bandera** en el enunciado de la primera práctica de *Inteligencia Artificial e Ingeniería del Conocimiento* del curso académico 2008-2009.

1.2. Decisiones de diseño

Al abordar este problema se plantearon una serie de conflictos (distancias mínimas reales, heurísticas, etc) cuya resolución no fue trivial y, dada su complejidad, se explicarán en apartados posteriores dedicados expresamente a ellos. No obstante, a este nivel ya se puede hablar de tres decisiones de diseño de gran importancia:

- El temporizador.
- El método de búsqueda.
- Control de estados repetidos.

1.2.1. Temporizador

Como se define en el enunciado del problema, el bot debe jugar de forma autónoma contra un adversario y existe un parámetro fijado al crear una partida que indica el tiempo que tiene cada jugador para "pensar" su jugada. Este parámetro acotará la profundidad a la que puede llegar el algoritmo Mini-Max y es crucial mandar un movimiento al servidor dentro de este límite de tiempo.

Este problema se resolverá por medio de **hilos**. El programa principal pondrá a "falso" una variable compartida que hace la función de semáforo y creará un hilo. El hilo lanzará el algoritmo de búsqueda y el programa principal dormirá una cantidad de tiempo cercana a la duración del turno. Al despertar, pondrá a "verdadero" el semáforo y el hilo que lanzó el algoritmo de búsqueda parará su ejecución. Para que todo esto funcione correctamente, el algoritmo de búsqueda debe ir guardando en todo momento el mejor nodo que ha ido encontrando.

1.2.2. Método de búsqueda

Generalmente los algoritmos Mini-Max hacen una búsqueda clásica en profundidad, pero cuando el árbol de búsqueda es muy extenso y estamos limitados por tiempo, este método puede hacerse inviable ya que podemos dejar sin explorar zonas del árbol muy prometedoras. Para una búsqueda más exhaustiva, emplearemos un método de búsqueda de **profundidad iterativa**. De este modo nos aseguramos el ir explorando cada nivel por completo y nos es posible guardar la mejor solución encontrada hasta el momento.

1.2.3. Control de estados repetidos

Como se verá a lo largo de este documento, el algoritmo Mini-Max tiene una función muy importante llamada *expandir* que se encarga de generar todos los sucesores de un determinado nodo. Es posible que los sucesores generados ya hayan sido generados en algún otro momento y estén en espera para ser evaluados o expandidos, o incluso que ya hayan sido evaluados y expandidos.

Para evitar este crecimiento innecesario del árbol de búsqueda se tratará de controlar la creación de estados repetidos mediante dos estructuras de datos:

1. Lista de estados repetidos.

2. Diccionario de estados repetidos, simulado una tabla hash.

1.3. Estructuras de datos

En la figura 1.1 se muestra el diagrama de clases a partir del cual afrontaremos la programación de la práctica. Posteriormente, se mostrarán cada una de las clases individualmente y se dará una breve explicació de sus métodos y atributos.

1.3.1. Clase Casilla

Esta clase (ver figura 1.2) almacena la información relativa a una casilla dada.

- **Atributos**: identificador y tipo.

- **Métodos**: *convertirHierba()* y *cavar()* únicamente actualizan el atributo *tipo* del objeto. La operación *coste()* devuelve un entero que indica cuánta vida la cuesta a un jugador moverse a dicha casilla, teniendo en cuenta las unidades de hacha (para bosques) y barca (para agua) de ese jugador.

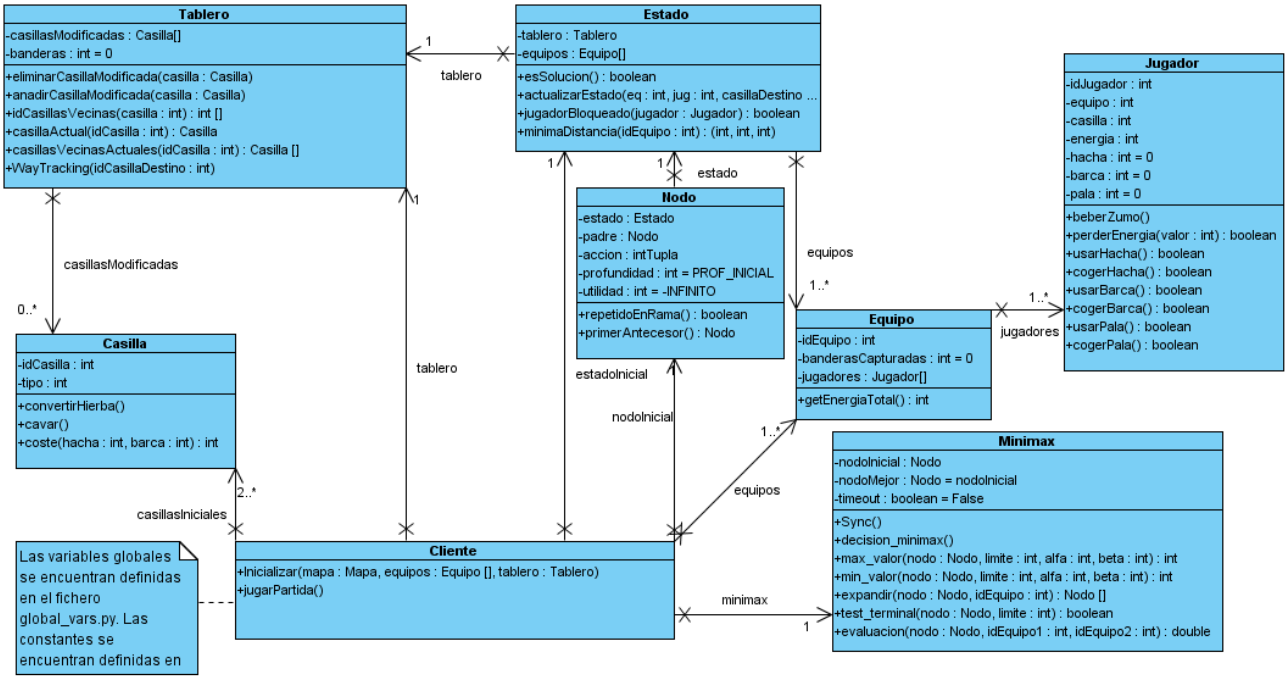

Figura 1.1: Diagrama de clases

1.3.2. Clase Tablero

Esta clase (ver figura 1.3) almacena los cambios que han sucedido en el tablero desde el inicio hasta un momento dado. De este modo, podemos calcular el estado actual del tablero sin tener que almacenar información que no ha cambiado.

- **Atributos**: lista de casillas modificadas y número de banderas que hay actualmente en el tablero.
- **Métodos**.
  - *idCasillasVecinas()* devuelve una lista de identificadores de las seis casillas vecinas a la casilla indicada.
  - *casillaActual()* devuelve un objeto Casilla dado un identificador de casilla. Este objeto contiene el tipo actual de la casilla en ese momento.
  - *casillasVecinasActuales()* devuelve una lista que contiene las seis casillas vecinas al identificador dado. Estos objetos Casilla contienen su tipo actual.

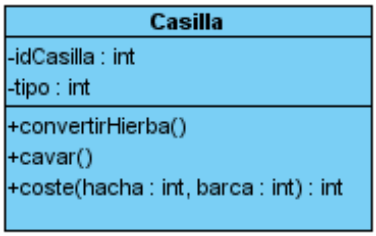

Figura 1.2: Clase Casilla

- *WayTracking()* es un algoritmo que calcula las distancias reales desde cualquier casilla del tablero a una casilla dada. Más adelante se dará una descripción más detallada de este algoritmo.

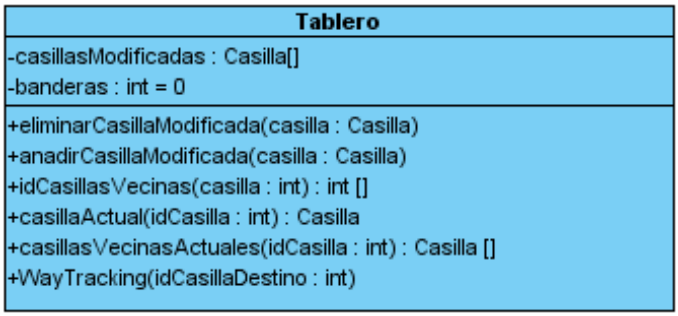

Figura 1.3: Clase Tablero

1.3.3. Clase Jugador

Esta clase (ver figura 1.4) almacena la información relativa a cada jugador tal y como se indica en el enunciado de la práctica.

- **Atributos**: identificador del jugador, identificador del equipo, casilla actual, energia y las unidades de los objetos que posee.
- **Métodos**: Como se puede observar, prácticamente todos los métodos devuelven un *boolean*. Se trata de los métodos que implican coger un objeto o utilizarlo, por lo que esta variable booleana indica si el jugador ha cogido un objeto o lo ha utilizado. La

importancia de esta decisión de diseño radica en poder saber si el jugador se ha movido o no, de este modo únicamente se generarán los sucesores que han generado algún cambio en el estado de la partida.

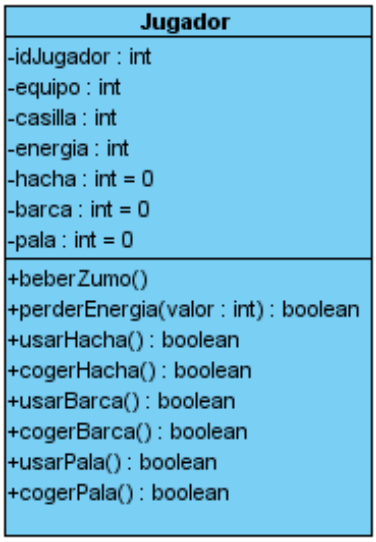

Figura 1.4: Clase Jugador

1.3.4. Clase Equipo

Esta clase (ver figura 1.5) almacena la información relativa a un equipo. En nuestro problema únicamente existirán dos instancias de esta clase: una para el equipo MIN y otra para el equipo MAX.

- **Atributos**: identificador del equipo, banderas capturadas por el equipo y la lista de jugadores que pertenecen al equipo.
- **Métodos**: *getEnergiaTotal()* suma las energias de todos los jugadores del equipo. Este método es importante para la heurística que valore lo buena o mala que sea la cantidad de energía de un equipo.

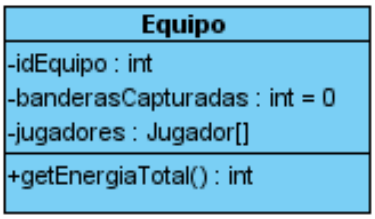

Figura 1.5: Clase Equipo

1.3.5. Clase Estado

Esta clase (ver figura 1.6) almacena la información relevante de un estado.

- **Atributos**: una instancia de la clase Tablero que indica las modificaciones que se han hecho en el tablero desde el inicio hasta este estado, y la lista de equipos que juegan la partida con el estado de cada uno de sus jugadores.
- **Métodos**.
  - *esSolucion()* devuelve "verdadero" si el número de banderas que quedan en el tablero es 0. En caso contrario devuelve "falso".
  - *actualizarEstado()* actualiza la información del jugador con el que se realiza una acción, la información de su equipo y la información de la casilla a la que se mueve. Devuelve "verdadero" en caso de que se haya modificado el estado y "falso" en caso contrario. Esta decisión es importante para únicamente generar nodos que tengan modificaciones en su estado.

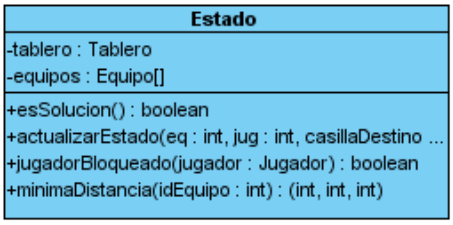

Figura 1.6: Clase Estado

1.3.6. Clase Nodo

Esta clase (ver figura 1.7) almacena la información perteneciente a cada nodo. Esta información está reflejada en sus atributos.

- **Atributos**: el estado de este nodo, el nodo padre, la acción que se ha ejecutado para llegar a este nodo, la profundidad del nodo y su valor de utilidad (inicialmente valorado a -INFINITO).
- **Métodos**.
  - *repetidoEnRama()* que comprueba si este nodo contiene un estado repetido en alguno de sus antecesores.
  - *primerAntecesor()* que devuelve el nodo con profundidad inicial que nos ha llevado a generar este nodo.

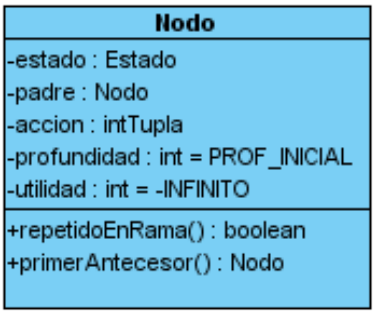

Figura 1.7: Clase Nodo

1.3.7. Clase Minimax

Esta clase (ver figura 1.8) es la clase principal que maniobra con todas las estructuras en busca de una jugada.

- **Atributos**: el nodo inicial a partir del cual comenzaremos la búsqueda, el mejor nodo encontrado hasta el momento y una variable de control que nos indica si podemos seguir buscando o no.

- **Métodos**.

  - *Sync()* reinicia la variable compartida *timeout* a "falso" y duerme durante X segundos, donde X es el tiempo por turno con el que se configura la partida menos un breve intervalo de tiempo de seguridad.
  - *decision-minimax()* inicia la búsqueda en profundidad iterativa desde el nodo inicial expandiéndolo como MAX.
  - *max-valor()* y *min-valor()* son los métodos utilizados en el algoritmo minimax para la búsqueda de los mejores y peores sucesores. Más adelante se explicará con detalle.
  - *expandir()* expande un nodo generando únicamente los sucesores válidos que han generado algún cambio en el estado.
  - *test-terminal()* devuelve "verdadero" si el nodo contiene un estado objetivo o si se ha llegado a la profundidad de corte.
  - *evaluacion()* obtiene un valor para el nodo dado. Esta función será explicada con detalle más adelante.

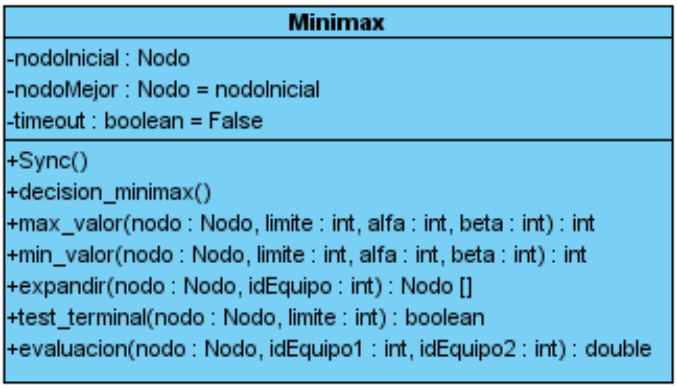

Figura 1.8: Clase Minimax

Capítulo 2

Soluciones Teóricas

En un algoritmo Mini-Max se consideran dos jugadores a los que llamaremos Max y Min ya que, una vez que evaluamos un nodo, se supone que los valores altos son buenos para Max y los malos para Min. De esto se deduce que, dado que queremos que nuestro equipo consiga las mejores puntuaciones, nosotros seremos el equipo Max y nuestro rival será el equipo Min.

Considerando un árbol de juegos, la estrategia óptima puede determinarse examinando el **valor minimax** de cada nodo (si es un estado terminal es solamente su utilidad). Un algoritmo minimax calcula la decisión minimax del estado actual. La recursión avanza hacia las hojas del árbol, y entonces los valores minimax retroceden por el árbol cuando la recursión se va deshaciendo. Como se puede deducir, un algoritmo minimax realiza una **búsqueda primero en profundidad** completa del árbol de juegos, es decir, si la profundidad máxima de árbol es *m*, y hay *b* movimientos legales en cada punto, entonces la complejidad en tiempo del algoritmo minimax es $O(b^m)$ y la complejidad en espacio es $O(bm)$.

Como hemos visto, el problema de la búsqueda minimax es que el número de estados que tiene que examinar es exponencial en el número de movimientos. Lamentablemente no podemos eliminar el exponente, pero podemos dividirlo, con eficacia, en la mitad. La jugada es que es posile calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juegos aplicando la ténica de **poda alfa-beta**. Esta poda puede aplicarse en árboles de cualquier profundidad y, a menudo, es posible podar subárboles enteros. Además, es necesario tener en cuenta que los estados repetidos en el árbol de búsqueda pueden causar un aumento exponencial del coste de búsqueda. Esto se debe a permutaciones diferentes de la secuencia

de movimientos que terminan en la misma posición. Para resolver este problema se utiliza una **tabla de transposición**, que tradicionalmente es una *tabla hash* idéntica a la *lista cerrada* que utilizamos en el algoritmo A* de la primera práctica. La forma de trabajar con dicha tabla es guardando la evaluación de cada posición la primera vez que se encuentre, de modo que no tenemos que volver a calcularla las siguientes veces.

Por último, hay que tener en cuenta que la **función de utilidad** (que comprueba si un estado es objetivo y le asigna un valor de utilidad) hay que modificarla ligeramente si no podemos explorar completamente el árbol de búsqueda (y no podremos ya que los turnos están limitados en tiempo), ya que no podemos explorar las ramas hasta las hojas y debemos cortar a una determinada profundidad. Esta nueva función recibe el nombre de **función de evaluación**, que devuelve una estimación de la utilidad esperada de una posición dada. En este momento, podemos plantearnos la posibilidad de implementar nuestro algoritmo mediante **profundidad iterativa** para que cuando se agote el tiempo, el programa nos devuelva el movimiento seleccionado por la búsqueda completa más profunda.

2.1. Conceptos básicos

Para la realización de esta práctica es necesario tener en cuenta los siguientes conceptos:

- **Estado**. En la primera práctica un estado estaba compuesto por una lista de jugadores (ya que únicamente jugaba un equipo) y una lista de casillas del tablero que se habían ido modificando a lo largo de la partida. En esta segunda práctica, en la que juegan juegan dos equipos (el nuestro y el rival), es necesario hacer un pequeño ajuste para que el estado esté compuesto por una lista de equipos (cada equipo tendrá una lista con sus jugadores) y una lista de casillas modificadas. Pero, como hemos visto en la sección anterior, la complejidad en espacio de un algoritmo minimax es $O(bm)$. Dicho esto, puede ser interesante plantearse la posibilidad de almacenar el estado del tablero completo para no tener que recalcularlo cada vez que lo necesitemos aplicando los cambios indicados por la lista de casillas modificadas.

- **Función sucesor**. Esta función es similar a la que utilizamos en la primera práctica,

solo que añadiremos un séptimo movimiento para poder cavar. De modo que serán dos bucles anidados en los que el primero recorrerá la lista de jugadores y el segundo hará cada uno de los siete posibles movimientos, generando así todos los posibles sucesores de un nodo.

- **Test terminal**. Esta función determina si un nodo contiene un estado objetivo. Un estado será objetivo cuando se hayan capturado todas las banderas del equipo contrario o no nos quede ningún jugador con vida.

- **Función de utilidad**. Nuestro juego es un **juego de suma no cero**. Esta función evaluará un nodo terminal y le asignará un valor de utilidad en función del número de banderas capturadas y la energía total de cada equipo, tratando así de maximizar las banderas que captura nuestro equipo y su energía; y de minimizar las banderas que captura el equipo rival y su energía. En esta función no es necesario tener en cuenta las distancais de los jugadores a las banderas restantes ya que estamos hablando de **nodos terminales**.

- **Función de evaluación**. Convierte los nodos no terminales en hojas terminales. Es necesario sustituir test terminal por un **test-límite** que decide cuando aplicar la función de evaluación (que sustituye a la función de utilidad), que nos devolverá una estimación de la utilidad esperada de una posición dada.

2.2. Algoritmo Minimax

A continuación se muestra el pseudocódigo del algoritmo Mini-Max que se implementará. Este pseudocódigo está diseñado para una *búsqueda primero en profundidad* por lo que habrá que hacer una pequeña modificación en la función *decisión-minimax()* para que la búsqueda sea de *profundidad iterativa*. Esta modificación consiste en pasar a *valor-max()* y *valor-min()* un argumento *límite* que se irá incrementando unidad a unidad mediante una estructura iterativa de tipo *for*. Las funciones *valor-max()* y *valor-min()* pasarán a su vez este parámetro al *test-terminal()* que comprobará cuándo se llega a la profundidad de corte.

La documentación utilizada a la hora de diseñar el algoritmo Mini-Max e incorporarle la poda Alfa-Beta se puede encontrar en [3]

Nótese que a este nivel, la función *utilidad()* es una mera aproximación y será revisada con más detenimiento en la sección 2.4.

```
# nodoMejor es una variabe global o un atributo de clase que se inicializa
#con su valoracion a -INFINITO

# El nodo raiz es de tipo MAX
function decision-minimax(nodo)
    v=-INFINITO
    max-valor(nodo)


function max-valor(nodo):
    if test-terminal(nodo.estado):
        return utilidad(nodo.estado,MAX,MIN)
    else:
        v=-INFINITO
        accion=(-1,-1)
        # Expando el nodo MAX
        sucesores=expandir(nodo,MAX)
        for s in sucesores:
            v=max(v, min-valor(s))
            if v>nodoMejor.v:
                nodoMejor=primerAntecesor(s)
                nodoMejor.v=v
    return v


function min-valor(nodo):
    if test-terminal(nodo.estado):
        return utilidad(nodo.estado,MAX,MIN)
    else:
        v=INFINITO
        accion=(-1,-1)
        sucesores=expandir(nodo.estado,MIN)
        for s in sucesores:
            v=min(v, max-valor(s))
    return (accion,v)


function test-terminal(estado,equipo):
```

```
    terminal=True
    # Si se han capturado todas las banderas, es terminal
    if estado.equipos[equipo].banderasCapturadas==banderasTotales:
        pass
    else:
        # Sino, comprobamos si hay algun jugador vivo
        # Si queda alguno vivo, no es terminal
        for jug in estado.equipos[equipo].jugadores:
            if jug.energia<=0:
                termnal=False
                break
    return terminal


function utilidad(estado,equipo):
    eq_aux=estado.equipos[equipo]
    utilidad=0
    utilidad+=valoracionEnergia(eq_aux,energiaTotal)
    utilidad+=valoracionCapturas(eq_aux.banderasCapturadas,banderasTotales)
    return utilidad
```

2.3. Poda Alfa-Beta

A continuación se muestra el pseudocódigo del algoritmo minimax de la sección 2.2 con la poda alfa-beta incorporada.

Su funcionamiento es sencillo. Consiste en dos variables locales *alfa* y *beta* que indican el valor de la mejor alternativa para MAX y el valor de la mejor alternativa para MIN a lo largo del camino respectivamente. Inicialmente, *alfa* es tiene un valor muy malo para que éste pueda ser mejorado (-INFINITO), y *beta* tiene un valor muy bueno para que éste pueda ser empeorado (INFINITO), ya que a MAX le interesa que el contrario obtenga el menor beneficio. Estas variables se van pasando de una llamada a otra y sus valores van siendo actualizados. De este modo podemos detectar cuando hay subárboles que no son prometedores y podemos "podarlos" sin compromiso alguno.

```
# alfa: valor de la mejor alternativa (el valor mas alto) encontrada
#hasta el momento para un nodo MAX
# beta: valor de la mejor alternativa (el valor mas bajo) encontrada
```

```
#hasta el momento para un nodo MIN
# accion es una tupla de enteros (jugador,movimiento)


function busqueda-alfa-beta(nodo)
    v=-INFINITO
    max-valor(nodo,-INFINITO,INFINITO)


function max-valor(nodo,alfa,beta):
    if test-terminal(nodo.estado):
        return utilidad(nodo.estado,MAX,MIN)
    else:
        v=-INFINITO
        accion=(-1,-1)
        sucesores=expandir(nodo,MAX)
        for s in sucesores:
            v=max(v, min-valor(s,alfa,beta))
      # poda
            if v>=beta:
         return(accion,v)
      # si no se poda...
            alfa=max(alfa,v)
      if v>nodoMejor.v:
         nodoMejor=primerAntecesor(s)
         nodoMejor.v=v
    return v


function min-valor(nodo,alfa,beta):
    if test-terminal(nodo.estado):
        return utilidad(nodo.estado,MAX,MIN)
    else:
        v=INFINITO
        accion=(-1,-1)
        sucesores=expandir(nodo,MIN)
        for s in sucesores:
            (accion_aux,v_aux)=max-valor(s,alfa,beta)
      # poda
            if v<=alfa:
         return(accion,v)
            beta=min(beta,v)
    return (accion,v)
```

```
function test-terminal(estado,equipo):
    terminal=True
    # Si se han capturado todas las banderas, es terminal
    if estado.equipos[equipo].banderasCapturadas==banderasTotales:
        pass
    else:
        # Sino, comprobamos si hay algun jugador vivo
        # Si queda alguno vivo, no es terminal
        for jug in estado.equipos[equipo].jugadores:
            if jug.energia<=0:
                termnal=False
                break
    return terminal


function utilidad(estado,equipo):
    eq_aux=estado.equipos[equipo]
    utilidad=0
    utilidad+=valoracionEnergia(eq_aux,energiaTotal)
    utilidad+=valoracionCapturas(eq_aux.banderasCapturadas,banderasTotales)
    return utilidad
```

2.4. Heuristicas Diseñadas

En realidad sólo se ha diseñado una heurística, pero ésta se ha obtenido a lo largo de cuatro iteraciones bien definidas. En cada iteración se trataba de lograr un objetivo, es decir, un comportamiento por parte de los jugadores

2.4.1. Algoritmo WayTracking

Probablemente el aspecto más crucial a la hora de desarrollar la heurística sea el cálculo de distancias reales desde los jugadores a las banderas. Este problema ya se planteó en la primera práctica cuando se desarrolló el algoritmo de *búsqueda informada A** y se resolvió mediante distancias euclídieas o distancias en línea recta, una forma muy sencilla y válida en caso de tratarse de un entorno **parcialmente observable**, pero como en este caso el entorno es **totalmente observable**, se trata de un cálculo poco eficiente ya que no tiene en cuenta las restricciones que pueda imponer éste (ver figura 2.1). Por esta razón, se pro-

pone un algoritmo desarrollado por el autor de este documento y que ha sido denominado **WayTracking**. WayTracking es un algoritmo de tipo *backtracking* que calcula las distancias reales desde una casilla a todas las demás casillas teniendo en cuenta las murallas del tablero. Se trata de un algoritmo muy eficiente y se ha conseguido calcular las distancias en mapas de hasta $100x100$ casillas en menos de $0{,}5$ segundos.

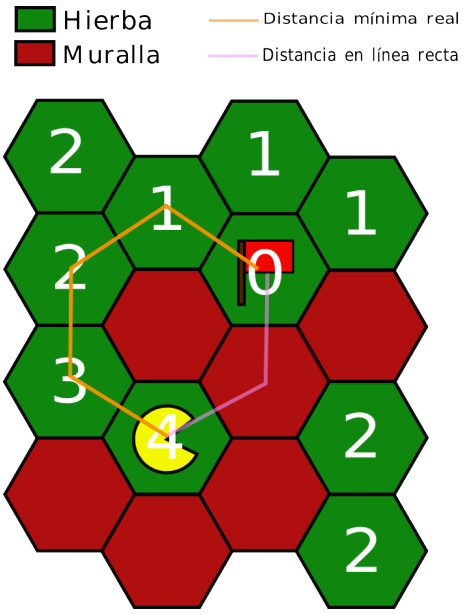

Figura 2.1: Cálculo de distancias reales y en línea recta

WayTracking consta de dos fases:

1. La primera es una inicialización en la que se etiqueta la casilla destino donde se encuentra la bandera con un cero, indicando distancia cero. A continuación, se etiquetan todas las casillas del tablero a una distancia máxima de la casilla destino, por ejemplo INFINITO. El siguiente paso es etiquetar todas las casillas que son de tipo *muralla* a distancia -1. De este modo se indica que esas casillas son inalcanzables. Dada esta fase de inicialización, WayTracking sólo puede utilizarse en entornos totalmente observables.

2. La segunda fase son una serie de llamadas recursivas al algoritmo de WayTracking que se encarga de calcular las distancias mínimas desde la casilla destino a cada una de las casillas etiquetadas como INFINITO del tablero.

El algoritmo se planteó siguiendo los siguientes razonamientos:

- En la **etapa** k etiquetamos las casillas adyacentes a las casillas a distancia k de la bandera supuesto que hemos etiquetado las $k-1$ casillas anteriores, de modo que las casillas etiquetadas con un valor inferior a k ya están etiquetadas con su distancia mínima.

- La **generación de descendientes** consiste en recorrer con un bucle *for* todas las casillas adyacentes a aquellas que tienen distancia k.

- El **test de solución** es cuando hemos etiquetado todas las casillas ($filas * columnas$)

- El **test de fracaso** es intentar etiquetar con una distancia mayor una casilla etiquetada o etiquetar una muralla. Como las murallas se inicializaron con distancia -1, no podemos etiquetarla en ningún momento porque k comienza en cero.

- Buscamos minimizar la distancia, así que las **inicializaciones** se harán a INFINITO. No se trata de un problema de optimización de soluciones, por lo que no habrá que comparar distintas soluciones.

- La **solución** será una lista que indica la distancia real de cada casilla a una casilla dada.

Dicho esto, podemos observar la importancia de la información que nos genera WayTracking obsevando la figura 2.1, donde podemos ver que la distancia en línea recta es de 2 casillas mientras que WayTracking indica que son 4 casillas. Gracias a esto, podemos saber realmente si nuestros jugadores están más cerca de las banderas que los jugadores contrarios.

Como WayTracking sólo calcula las distancias desde una casilla hasta todas las demás casillas del tablero, deberemos hacer una ejecución del algoritmo por cada bandera. Así, podremos hacer un diccionario de distancias donde podremos encontrar las distancias reales desde cualquier casilla hasta cualquier bandera. Además, como el tablero es completamente obsevable antes de que comienze la partida (justo al conectar al servidor) y las banderas siempre están en posiciones fijas, podemos lanzar las ejecuciones de WayTracking antes de consumir tiempo de nuestro turno y realizar los cálculos una única vez, reutilizándolos a lo largo de toda la partida.

La implementación de WayTracking puede verse en el código fuente de la clase Tablero (sección 6.6).

2.4.2. Primera aproximación

En una primera iteración del diseño de la heurística, se ha tratado que los jugadores vayan a coger las banderas que se encuentran a menor distancia sin tener en cuenta la posición de los jugadores contrarios.

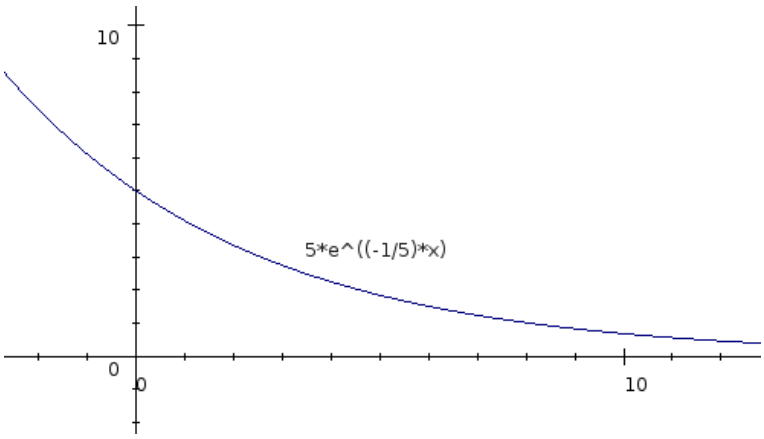

Figura 2.2: $b * e^{(-1/b)*x}$, siendo $b = 5$

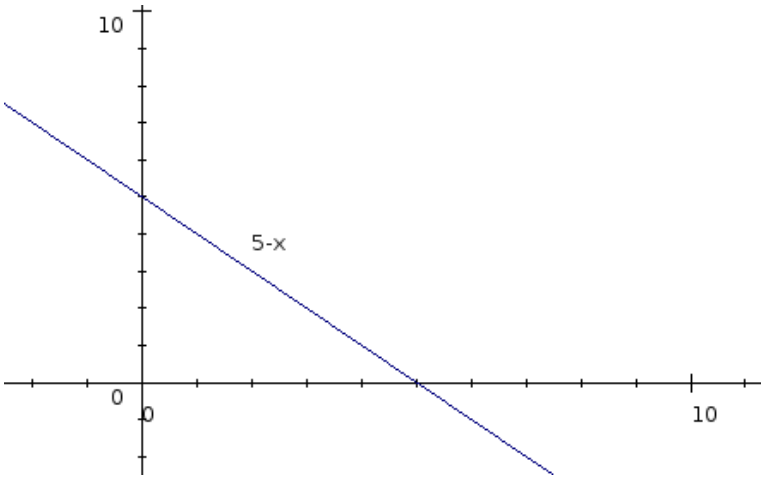

Figura 2.3: $b - x$, siendo $b = 5$

Es aquí cuando se plantea un problema a la hora de valorar las bondades de las distancias y las banderas porque, cuantas más banderas capturemos mejor, pero cuanto mayores sean las distancias mínimas peor es nuestra posición de juego. Con esto se intenta ver que hay que dar mayor valor a las distancias bajas y para ello necesitamos alguna función que nos invierta este concepto. Se proponen dos funciones como caso de estudio: $b * e^{(-1/b)*x}$ (ver figura 2.2) y $b - x$ (ver figura 2.3).

Vamos a tratar primero el tema de las banderas capturadas ya que es más sencillo. En esta primera interación se obtenían las banderas capturadas por el equipo MAX y las banderas capturadas por el equipo MIN para hacer la diferencia que se multiplicaba por una constante.

Para tratar el tema de las distancias es necesario entender las gráficas mostradas en 2.2 y 2.3: el eje de abcisas representa la distancia que estamos midiendo, y el eje de ordenadas representa el valor que vamos a darle a esa distancia. De este modo podemos obtener el cuadro 2.1.

Las ventajas de 2.3 sobre 2.2 es que con un mismo parámetro, no tiende hacia un valor 0 para distancias muy grandes y hace que las valoraciones sean más significativas porque da valores enteros y no valores con decimales que hacen que éstos sean muy próximos entre sí.

Distancia	**Función 2.2**	**Función 2.3**
5	1.83	0.00
4	2.24	1.00
3	2.74	2.00
2	3.35	3.00
1	4.09	4.00
0	5.00	5.00

Cuadro 2.1: Valoraciones de las funciones 2.2 y 2.3 para $b = 5$

Una vez que tenemos las valoraciones de las distancias mínimas de los jugadores MAX y MIN, las multiplicamos por una constante, calculamos la diferencia y lo sumamos a la evaluación que habíamos obtenido de las banderas.

Hasta ahora hemos evaluado los dos equipos del mismo modo y la evaluación resultante es la diferencia de ambas evaluaciones, pero existe el problema de las constantes utilizadas

al evaluar las banderas y las distancias que deberá ser resuelto en la segunda iteración (ver sección 2.4.3).

Por último, otro factor a tener en cuenta es que no midamos la distancia mínima desde un jugador que no está muerto pero que no se puede mover (p.e. un jugador rodeado de agua con 2 unidades de energía). Esto se resuelve mediante la función *jugadorBloqueado()* de la *Clase Estado*, que nos indica si el jugador se puede mover o no: si puede moverse lo tendremos en cuenta para el cálculo de distancias mínimas; en caso contrario, se considerará que está muerto.

2.4.3. Segunda aproximación

En esta iteración definiremos cuál será el parámetro b que tanto condiciona nuestra función 2.3. Después de una serie de pruebas, se ha aproximado que este parámetro tendrá un valor igual a $\frac{columnas+filas}{2} * \frac{3}{4}$, dado que es una aproximación a la máxima distancia que puede haber en cualquier tipo de tablero.

Ahora, dado que la bondad de las distancias es variable, tenemos que asegurarnos que las distancias no obtengan mejores valoraciones que la captura de banderas. Esto se consigue sustituyendo la constante que indica la bondad de las banderas por otro valor que varie en función del tablero. Se propone utilizar un valor igual a $filas * columnas$.

2.4.4. Tercera aproximación

Hasta ahora nuestros jugadores capturan correctamente las banderas cualesquiera que sea la dimensión del tablero y siempre y cuando nuestro contricante no coja las banderas antes que nosotros. Esto plantea el problema de que nuestros jugadores van a por la bandera con distancia más corta si pensar que el contrario va también a por esa misma bandera, dado que también es la que tiene a mínima distancia.

Aquí es donde entra el concepto de **blacklist** o **lista negra**. Consiste en que una vez que se han calculado las distancias mínimas de ambos equipos, se comprueban si ambas distancias se refieren a la misma bandera. En caso afirmativo, si la distancia del equipo contrario es menor que la de nuestro equipo, esa bandera se añadirá a la lista negra y se volverá a calcular la

distancia mínima para nuestro equipo sin tenerla en cuenta. Este proceso se hará hasta que se encuentre una bandera a la que realmente podamos optar a capturar o, si se da el caso en que el jugador contrario está mejor situado que nosotros con respecto a todas las banderas y todas ellas se han añadido a la lista negra, se escogerá una bandera aleatoriamente y será removida de la lista negra para dirigirnos hacia ella.

Si obsevamos la figura 2.4, se aprecia que la distancia mínima del *Jugador 1* es 1 y se refiere a la *Bandera 1*, y que la distancia mínima del *Jugador 2* es 2 y también se refiere a la *Bandera 1*. Como el *Jugador 2* no puede optar a coger la *Bandera 1*, la añadirá a su *blacklist* y volverá a calcular su distancia mínima, siendo en este caso 3 refiriéndose a la *Bandera 2*. Como esta bandera no está amenazada, irá a capturarla dejando que el *Jugador 1* capture la *Bandera 1*.

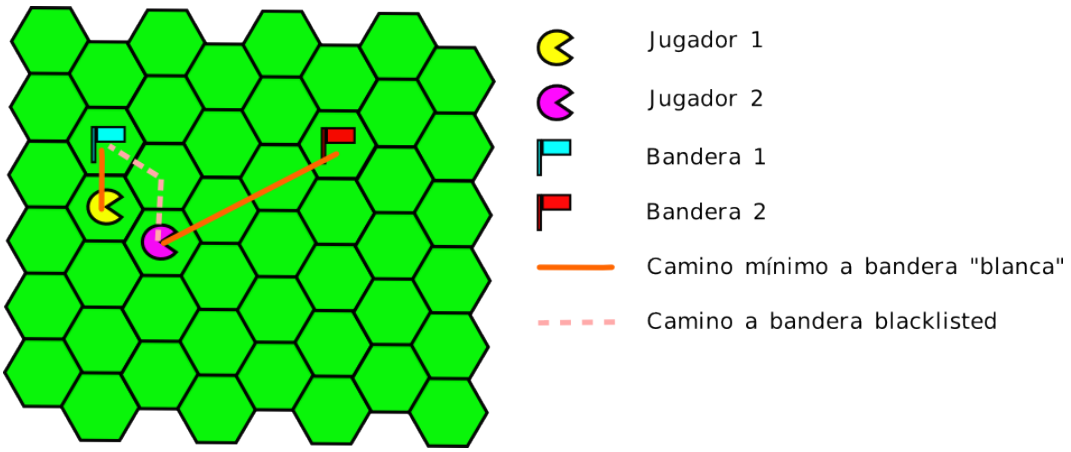

Figura 2.4: Ejemplo de *lista negra*

2.4.5. Cuarta aproximacion

Se llevó a cabo una cuarta iteración en la que se trataba de ahorrar la energía consumida. Se consiguió este comentido pero si no se llegaba en la búsqueda a la profundidad necesaria, podía provocar que perdiésemos alguna bandera o incluso la partida, por lo que fue deshechada.

Si observamos la figura 2.5 podemos ver un claro ejemplo en el que ahorrar energía nos podría suponer la pérdida de una bandera (suponiendo que mueva primero el *Jugador 1*).

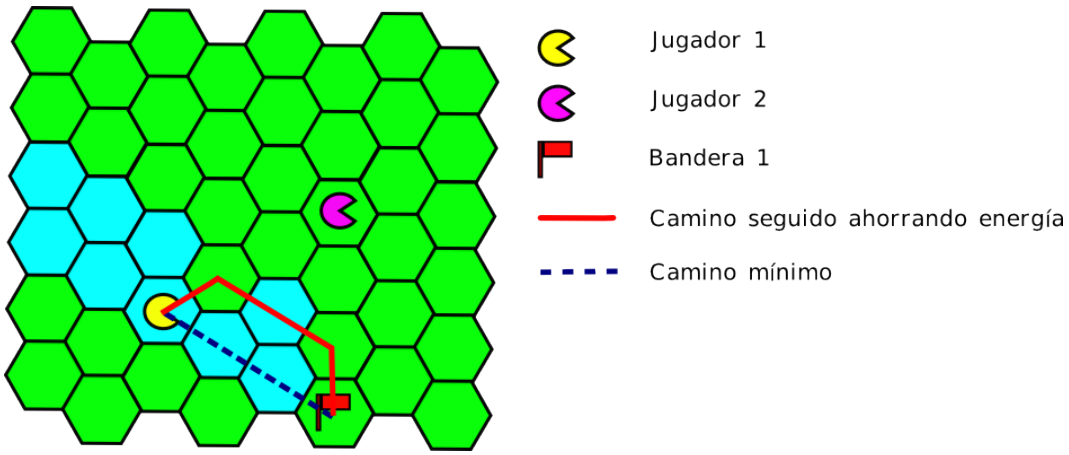

Figura 2.5: Ejemplo de ahorro de energía

Para evitar este tipo de situaciones, ya que los zumos creaban aún más conflicto, esta característica ha sido suprimida de la heurística.

2.4.6. Heurística final

Después de estas cuatro iteraciones, la heurística que hemos obtenido tiene las siguientes características:

- Siempre trabaja con distancias reales independientemente de las restricciones que imponga el trablero gracias al algoritmo **WayTracking**. Como ya se ha explicado, trabajar con distancias reales en lugar de distancias estimadas en línea recta es una solución mucho más eficiente (ver sección 2.4.1).

- No sólo se ignoran jugadores muertos en el cálculo de distancias mínimas, si no que también se ignoran jugadores que no pueden moverse por la razón que sea (ver sección 2.4.2).

- Evalúa el número de banderas y las distancias mínimas a las que se encuentran los jugadores de cada equipo utilizando bondades que varían en función del tamaño del tablero (ver sección 2.4.3).

- Tiene en cuenta que las banderas a las que se refieren las distancias mínimas de cada

equipo no sean las mismas, para no tratar de coger una bandera que sabemos que capturará el otro equipo antes que nosotros. Esto es gracias al uso de la **lista negra** (ver sección 2.4.4).

Capítulo 3

Pruebas

3.1. Detección de estados repetidos

Como ya se comentó en la sección 1.2.3 se ha tratado de realizar un control sobre los estados repetidos. Se trata de una característica importante porque en este tipo de problemas el número de estados a expandir y examinar crece exponencialmente y, si evitamos situaciones iguales podemos reducir drásticamente este espacio de búsqueda.

Mantener una estructura de datos que almacene los nodos que ya hemos expandido o examinado no sólo es costoso en términos de memoria si no que también lo es en términos de tiempo. No obstante, dado que los turnos están limitados por tiempo, la memoria no será un problema a no ser que los turnos sean lo suficientemente largos como para generar una grandísima cantidad de nodos. Por ello, nos hemos centrado en la optimización de tiempo por medio de dos técnicas:

- Una lista.
- Un diccionario simulando una tabla hash.

Inicialmente y sabiendo que era la peor solución, se trató de resolver el problema mediante una lista de estados. La peor parte de esta solución viene dada cuando la lista contiene miles de elementos y queremos comprobar si un estado ya está contenido en la lista. La comparación de estos elementos uno a uno es muy tediosa ya que interviene un gran número de atributos (de casillas, jugadores, equipos...) y hay que realizar esta operación miles de veces. Tras realizar

las pruebas pertinentes, se comprobó que no obteníamos mejores niveles de profundidad y que en ocasiones éstos empeoraban.

Por último, se trató de implementar una tabla hash por medio de un diccionario. Esta solución parece más práctica puesto que dado un identificador, podemos acceder a ese elemento directamente. El problema está en qué identificador utilizar. Se ha tratado de identificar cada estado mediante una máscara formada por las características del estado, es decir, una cadena de dígitos que representan las casillas, los jugadores, los equipos, etc. pero, a parte de ser un método que puede llevar a confusión en la identificación de estados, no ha dado los resultados esperados en cuanto a los niveles de profundidad alcanzados en la búsqueda. Éstos son similares a los obtenidos sin controlar los estados repetidos dada la poca duración de los turnos.

En conclusión, esta característica ha sido contemplada, estudiada y rechazada.

3.2. Resultados prácticos

A continuación se comentarán los resultados prácticos obtenidos durante la ejecución de las pruebas del programa en las distintas fases de desarrollo de la heurística. Los escenarios de prueba han sido los mapas *Mapas de pruebas mejor* y *Mapa*, disponibles en la página web de la asignatura.

3.2.1. Primera iteración

Esta primera iteración corresponde con la heurística desarrollada en la primera aproximación (ver sección 2.4.2).

Tras una serie de tests y ajustes, el comportamiento del bot era el esperado en mapas de dimensiones reducidas: los jugadores se aproximaban a las banderas hasta capturarlas. En mapas de dimensiones algo mayores, los jugadores hacían movimientos sin control debido a que los parámetros que condicionan las funciones 2.2 y 2.3 no habían sido ajustados. La consecuencia de esto es que un estado con una distancia mínima de 6 obtenía la misma valoración que otro estado con una distancia mínima de 10, cuando éste segundo estado es claramente peor.

3.2.2. Segunda iteración

En esta segunda iteración, que corresponde con la segunda aproximación de la heurística (ver sección 2.4.3), se hicieron un cálculos sencillos para determinar los valores que pueden tomar los parámetros que condicionan las funciones 2.2 y 2.3 para que éstas valoren bien las distancias cualesquiera que sean las dimensiones del tablero.

Los resultados obtenidos fueron los esperados: se ha resuelto en problema que se planteaba en la primera iteración cuando las dimensiones del mapa eran lo suficientemente grandes. En este momento, los jugadores ya se dirigen a las banderas y las capturan, independientemente de las dimensiones del tablero.

3.2.3. Tercera iteración

Esta tercera iteración, que corresponde con la tercera aproximación de la heurística (ver sección 2.4.4), ha sido la más compleja de testear y depurar para obtener los resultados esperados. No obstante, se ha logrado dotar a los jugadores con algo más de inteligencia para que éstos solo traten de capturar banderas que, en la medida de lo posible, tienen probabilidades de capturar, es decir, no compiten por banderas que es seguro que no van a poder capturar porque el contrincante las capturará antes.

3.2.4. Cuarta iteración

Esta iteración se correponde con la cuarta aproximación de la heurística (ver sección 2.4.5). Los resultados obtenidos eran los esperados si el nivel de profundidad al que se llegaba en la búsqueda era el adecuado ya que los jugadores se movían por los caminos menos costosos, pero si no se alcanzaba la profundidad adecuada, ahorrar energía podía suponer aumentar la longitud del camino y perder una bandera y, en el peor de los casos, perder la partida. Para evitar este tipo de situaciones, esta característica ha sido suprimda de la heurística.

Capítulo 4

Manual de Usuario

En esta sección se darán unas breves nociones de cuál es el contenido de la práctica, qué es necesario instalar en el sistema para poder ejecutarla y cómo ejecutarla.

4.1. Organización de directorios

La práctica se distribuye en un fichero con extensión *.tar.gz* que contiene cinco directorios:

- **config**. Contiene un fichero *config.client* empleado por el middleware de comunicaciones ICE. Se recomienda no modificar este fichero.
- **doc**. Contiene la documentación de la práctica así como las imágenes utilizadas.
- **exec**. Contiene dos scripts bash *run* y *kill*.
    - **run**. Admite un parámetro que es un número entero en el cual se indica el número de jugadores que van a competir (1 si es una partida individual y 2 si se trata de una competición). Si no se suministra este parámetro, se mostará un mensaje de ayuda. Ejemplo de uso: *./run 1*
    - **kill**. Su ejecución no admite ningún parámetro (*./kill*). Consiste en una tubería compuesta mediante las utilidades *ps*, *grep* y *awk* que obtiene los *PID* de los clientes en ejecución y los mata. Es un script muy cómodo en el proceso de depuración de clientes.

- **slice**. Contiene un fichero *Practica.ice* que indica los tipos de datos e interfaces públicos del servidor ICE de la práctica.
- **src**. Contiene todos los ficheros que forman parte del código fuente de la práctica. Se puede observar un fichero para cada clase y dos ficheros (global-vars.py y config.py) que definen las variables globales y las constantes.

4.2. Instalación

Antes de ejecutar la práctica hay que instalar tres componentes en el sistema:

- **python** [2]. Es el intérprete de python.
- **psyco** [1]. Es un compilador en tiempo de ejecución especializado para python. Psyco puede acelerar notablemente aplicaciones que hacen un uso intensivo de la CPU. El rendimiento actual depende de forma importante de la aplicación y pueden aumentarse hasta 40 veces, aunque la mejora de rendimiento media es aproximadamente de 4x. Puede descargarlo de
- **ZeroC Ice** [5]. Runtime del middleware de comunicaciones empleado por el cliente y el servidor de la práctica.

Si se goza de una distribución GNU/Linux basada en debian simplemente hay que instalar los metapaquetes **python**, **python-psyco** y **zeroc-ice33**

4.3. Ejecución

Para poder ver como juega el *bot* es necesario configurar la partida en la web de la asignatura [4] siguiendo los siguientes pasos:

1. Abrir la página de la asignatura [4].
2. Introducir en la parte superior derecha nuestro login (DNI) y password para realizar la autenticación e iniciar una sesión.

3. Clickar en **Partida Activa** para acceder al panel de configuración de partidas (ver figura 4.1) si no hay ninguna partida creada, o visualizar la partida en curso.

Figura 4.1: Panel de Administración

4. Si queremos crear una partida: elegimos el mapa, el tiempo por turno y el modo de juego (individual o competición); si por el contrario queremos unirnos a una partida ya creada (modo competición), elegimos a qué partida deseamos unirnos.

Una vez que clickemos en **Aceptar** y ya tenemos configurada nuestra partida. Ahora tenemos que ejecutar el cliente para conectarnos al servidor. Para ello seguimos la siguiente secuencia de pasos:

1. Descomprimimos la práctica con el comando *tar xvfz fichero-practica.tar.gz*.

2. Nos situamos en el directorio *exec*.

3. Si vamos a jugar una partida invididual ejecutamos el script *run* con el argumento *1*. Abrimos otro terminal y repetimos este mismo paso; Si por el contrario vamos a jugar una partida en modo competición, ejecutamos el script *run* con el argumento *2*.

4. El programa conectará con el servidor y cuando ambos jugadores estén conectados lanzarán sus algoritmos minimax y comenzarán la partida.

Capítulo 5

Conclusiones

Después de la realización de esta práctica se ha comprendido cómo funciona un algoritmo Mini-Max y las ventajas que aporta la incorporación de la poda Alfa-Beta. Además, se han reutilizado los conceptos aprendidos durante la realización de la primera práctica para la elección del método de búsqueda: *primero en profundidad* o *profundidad iterativa*.

Como se ha podido comprobar a lo largo de las pruebas que se han hecho con el cliente, los papeles más importantes a la hora de decidir la mejor acción que se puede realizar en cada momento los juegan la heurística y la profundidad a la que llegamos explorando. Sin duda, el comportamiento del bot viene determinado por la heurística (p.e. ignorar banderas que sabemos que no vamos a coger) pero la profundidad a la que llegamos en la búsqueda es tan importante como la heurística ya que podemos obtener comportamientos inesperados debido a situaciones que no son del todo estables (p.e. perder una bandera por tratar de ahorrar energía se podría haber evitado si el nivel de profundidad al que llegamos explorando incluye la captura de esa bandera). Este último aspecto hace todavía más importante la incorporación de la poda Alfa-Beta al algoritmo Mini-Max, ya que nos permite la poda de sub-árboles completos y, por consiguiente, invertir el tiempo que invertiríamos en explorar esos sub-árboles en explorar otras ramas a mayores niveles de profundidad.

Por último, y en concepto de resumen, en este documento se han tratado los siguientes temas:

- En qué consiste el problema propuesto, las decisiones de diseño que se han tomado y

las estructuras de datos que se han empleado para su resolución.

- Las soluciones teóricas propuestas, cómo son llevadas a la práctica, y cómo realizar una buena heurística en varias iteraciones bien definidas, dividiendo el problema y resolviéndolo por partes.

- Los comportamientos, tanto esperados como inesperados, que se han obtenido por parte del bot en la ejecución de las pruebas.

- Y un manual de usuario para saber qué software es necesario instalar en el sistema y cómo se debe ejecutar la práctica para verla en funcionamiento.

Dejando de lado aspectos de los que ya se hablan en otras secciones de este documento y refiriéndome a experiencias personales, tengo que decir que esta es la primera vez que escribo una documentación en LaTeX y, aunque he tardado más tiempo en escribirla que con otros procesadores de textos como *OpenOffice.org Writer*, considero que la presentación y calidad del documento merecen la pena.

Por último, quiero agradecer a los profesores y desarrolladores del servidor la inclusión de distintos clientes en diferentes lenguajes de programación ya que nos sirven de ejemplo para futuros usos del interfaz de comunicaciones Ice.

Capítulo 6

Código Fuente

A continuación se mostrará el código fuente de cada uno de los ficheros que forman parte de la práctica.

6.1. Variables Globales

```
idUsuario = -1
casillasIniciales = []
banderasObjetivo = [] # ID de la casilla
distanciaBanderas = {}
filasTablero = 0
columnasTablero = 0
MIN = -1
MAX = -1
deadline = 8
```

6.2. Constantes

```
# Constantes que indican el tipo de cada casilla
T_HIERBA = 1
T_AGUA = 2
T_BARRO = 3
T_MURALLA = 4
```

```
T_HOYO = 5
T_ZANJA = 6
T_BANDERA = 7
T_BARCA = 8
T_HACHA = 9
T_ZUMO = 10
T_PALA = 11
T_BOSQUE = 12

INFINITO = 99999
PROF_INICIAL = 0
MAX_ACCIONES = 7

TIME_INCREMENT = 0.4
SECURITY_RANGE = TIME_INCREMENT
```

6.3. Clase Casilla

```
from config import *
import psyco

psyco.full()

class Casilla:
    def __init__ (self, idCasilla, tipo):
        self.idCasilla = idCasilla
        self.tipo = tipo


    def setIdCasilla (self, idCasilla):
        self.idCasilla = idCasilla


    def getIdCasilla (self):
        return self.idCasilla


    def setTipo (self, tipo):
        self.tipo = tipo
```

```
    def getTipo (self):
        return self.tipo


    def convertirHierba (self):
        self.setTipo (T_HIERBA)


    def cavar (self):
        if self.getTipo() == T_HOYO: self.setTipo (T_ZANJA)
        else:
            if self.getTipo() <> T_ZANJA: self.setTipo (T_HOYO)


    def coste (self, hacha, barca):
        tieneHacha = False
        tieneBarca = False
        if hacha > 0: tieneHacha = True
        if barca > 0: tieneBarca = True
        if self.tipo == T_HIERBA or self.tipo == T_BANDERA or self.tipo == T_PALA or self.
            tipo == T_BARCA or self.tipo == T_HACHA or self.tipo == T_ZUMO:
            return 1
        elif self.tipo == T_AGUA:
            if tieneBarca: return 3
            else: return 6
        elif self.tipo == T_BOSQUE:
            if tieneHacha: return 4
            else: return 8
        elif self.tipo == T_BARRO: return 2
        elif self.tipo == T_HOYO: return 4
        elif self.tipo == T_ZANJA: return 6


    def __eq__(self, other):
        return self.idCasilla==other.idCasilla


    def __str__(self):
        cadena = "CASILLA::: Id: %d\n" % self.getIdCasilla()
        cadena += "CASILLA::: Tipo: %d\n" % self.getTipo()
        return cadena
```

6.4. Clase Jugador

```
import global_vars, config
import psyco

psyco.full()

class Jugador:
    def __init__ (self, idJugador, equipo, casilla, energia, hacha = 0, barca = 0, pala = 0,
         banderas = 0, zumo = 0):
        self.idJugador = idJugador
        self.equipo = equipo
        self.casilla = casilla # Identificador (int) de la casilla actual del jugador
        self.energia = energia
        self.hacha = hacha
        self.barca = barca
        self.pala = pala


    def getIdJugador (self):
        return self.idJugador


    def getEquipo (self):
        return self.equipo


    def getCasilla (self):
        return self.casilla


    def setCasilla (self, valor):
        self.casilla = valor


    def getEnergia (self):
        return self.energia


    def setEnergia (self, valor):
        self.energia = valor


    def beberZumo (self):
```

```
        mover = self.perderEnergia(1)
        if mover:
            self.energia = self.energia + 20
        return mover


    def perderEnergia (self, valor):
        energiaActual = self.energia - valor
        if energiaActual >= 0:
            self.energia = energiaActual
            return True
        else:
            return False


    def getHacha (self):
        return self.hacha


    def setHacha (self, valor):
        self.hacha = valor


    def usarHacha (self):
        """ Cuando se entra en una casilla de bosque se utiliza el hacha siempre
        que se tenga. No hay eleccion a utilizarla o no.
            DEVUELVE True si ha utilizado el hacha y False si no la ha utilizado
        """
        if self.getHacha() > 0:
            if self.perderEnergia(4):
                self.setHacha(self.getHacha() - 1)
                return (True, True) # Se mueve y usa el hacha
            else:
                return (False, False) # No se puede mover asi que no usa el hacha
        else:
            if self.perderEnergia(8):
                return (True, False) # Se mueve y no usa el hacha
            else:
                return (False, False) # No se puede mover y no tiene hacha


    def cogerHacha (self):
        mover = self.perderEnergia(1)
        if mover:
            self.setHacha(self.getHacha() + 20)
```

```
        return mover


    def getBarca (self):
        return self.barca


    def setBarca (self, valor):
        self.barca = valor


    def usarBarca (self):
        """ Cuando se entra en una casilla de agua se utiliza la barca siempre
        que se tenga. No hay eleccion a utilizarla o no.
        """
    # Devolvemos si se ha podido mover o no
        mover = True
        if self.getBarca() > 0:
            mover = self.perderEnergia(3)
            # Si tiene energia, gasta barca
            if mover:
                self.setBarca(self.getBarca() - 1)
        else:
            mover = self.perderEnergia(6)
        return mover


    def cogerBarca (self):
        mover = self.perderEnergia(1)
        if mover:
            self.setBarca(self.getBarca() + 20)
        return mover


    def getPala (self):
        return self.pala


    def setPala (self, valor):
        self.pala = valor


    def usarPala (self):
        if self.getPala() > 0:
            self.setPala(self.getPala() - 2)
```

```
            return True
        else:
            return False


    def cogerPala (self):
        mover = self.perderEnergia(1)
        if mover:
            self.setPala(self.getPala() + 10)
        return mover


    def __str__ (self):
        cadena = "JUGADOR::: ID Jugador: %d\n" % self.getIdJugador()
        cadena += "JUGADOR::: Equipo: %d\n" % self.getEquipo()
        cadena += "JUGADOR::: Casilla: %d\n" % self.getCasilla()
        cadena += "JUGADOR::: Energia: %d\n" % self.getEnergia()
        cadena += "JUGADOR::: Hacha: %d\n" % self.getHacha()
        cadena += "JUGADOR::: Barca: %d\n" % self.getBarca()
        cadena += "JUGADOR::: Pala: %d\n" % self.getPala()
        return cadena
```

6.5. Clase Equipo

```
from jugador import *
import psyco

psyco.full()


class Equipo:
    def __init__ (self, idEquipo, jugadores_ice, banderasCapturadas = 0):
        self.idEquipo = idEquipo
        self.banderasCapturadas = banderasCapturadas
        self.jugadores = []
        for jug in jugadores_ice:
            #jugadores_ice es una estructura de tipo Jugador definida en Practica.ice
            if jug.equipo == self.idEquipo:
                jugAux = Jugador(jug.idJugador, jug.equipo, jug.casilla, jug.energia)
                self.jugadores.append(jugAux)
```

```
    def getIdEquipo (self):
        return self.idEquipo


    def getBanderasCapturadas (self):
        return self.banderasCapturadas


    def setBanderasCapturadas (self, valor):
        self.banderasCapturadas = valor


    def capturarBandera(self):
        self.setBanderasCapturadas(self.getBanderasCapturadas() + 1)


    def getJugadores(self):
        return self.jugadores


    def getEnergiaTotal(self):
        energiaTotal = 0
        for i in self.jugadores:
            energiaTotal += i.energia
        return energiaTotal


    def __str__ (self):
        cadena = "EQUIPO::: ID Equipo: %d\n" % self.getIdEquipo()
        cadena += "EQUIPO::: Banderas capturadas: %d\n" % self.getBanderasCapturadas()
        for i in self.jugadores: cadena += str(i)
        return cadena
```

6.6. Clase Tablero

```
from casilla import *
import global_vars, copy
import psyco

psyco.full()
```

```
class Tablero:
    def __init__ (self, casillas=[], banderas=0):
        self.casillasModificadas = casillas[:]
        self.banderas=banderas


    def getCasillasModificadas(self):
        return self.casillasModificadas


    def setCasillasModificadas(self, seq):
        self.casillasModificadas.extend(seq)


    def eliminarCasillaModificada(self, casilla):
        self.casillasModificadas.remove(casilla)


    def anadirCasillaModificada(self, casilla):
        self.casillasModificadas.append(casilla)


    def idCasillasVecinas (self, casilla):
        """ RECIBE un entero que es el identificador de la casilla para la cual
        queremos encontrar las vecinas.
            DEVUELVE una lista con los identificadores de las casillas vecinas
        dada una casilla en un tablero de casillas hexagonales
        """

        casillasVecinas = [-1, -1, -1, -1, -1, -1]
        columnas = global_vars.columnasTablero
        filas = global_vars.filasTablero
        cruzarAncho = filas*columnas-columnas
        if casilla > 0 and casilla <= columnas: # Primera fila
            casillasVecinas[0] = casilla + cruzarAncho
            if (casilla%2) != 0: # Primera fila e impar
                casillasVecinas[1] = casilla + cruzarAncho + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla + columnas
                if casilla == 1: # Primera fila y primera casilla
                    casillasVecinas[4] = columnas
                    casillasVecinas[5] = columnas*filas
                else: # Primera fila y resto de casillas impares
```

```
                casillasVecinas[4] = casilla - 1
                casillasVecinas[5] = casilla + cruzarAncho - 1
            else: # Primera fila y par
                if casilla == columnas: # Primera fila y ultima casilla(par)
                    casillasVecinas[1] = 1
                    casillasVecinas[2] = casilla + 1
                else: # Primera fila y resto de casillas pares
                    casillasVecinas[1] = casilla + 1
                    casillasVecinas[2] = casilla + columnas + 1
                casillasVecinas[3] = casilla + columnas
                casillasVecinas[4] = casilla + columnas - 1
                casillasVecinas[5] = casilla - 1
        elif casilla > cruzarAncho and casilla <= columnas*filas: # Ultima fila
            casillasVecinas[0] = casilla - columnas
            if (casilla%2) != 0: # Ultima fila e impar
                casillasVecinas[1] = casilla - columnas + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla - cruzarAncho
                if casilla == cruzarAncho + 1: # Ultima fila y primera casilla (impar)
                    casillasVecinas[4] = columnas*filas
                    casillasVecinas[5] = casilla - 1
                else: # Ultima fila y resto de casillas impares
                    casillasVecinas[4] = casilla - 1
                    casillasVecinas[5] = casilla - columnas - 1
            else: # Ultima fila y par
                if casilla == columnas*filas: # Ultima fila y ultima casilla (par)
                    casillasVecinas[1] = casilla - columnas + 1
                    casillasVecinas[2] = 1
                else: # Ultima fila y resto de casillas pares
                    casillasVecinas[1] = casilla + 1
                    casillasVecinas[2] = casilla - cruzarAncho + 1
                casillasVecinas[3] = casilla - cruzarAncho
                casillasVecinas[4] = casilla - cruzarAncho - 1
                casillasVecinas[5] = casilla - 1
        else: # No es ni primera ni ultima fila
            casillasVecinas[0] = casilla - columnas
            if (casilla%2) != 0: # Impar
                casillasVecinas[1] = casilla - columnas + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla + columnas
                if ((casilla-1)%columnas == 0): # Borde izquierdo
                    casillasVecinas[4] = casilla + columnas - 1
                    casillasVecinas[5] = casilla - 1
                else:
                    casillasVecinas[4] = casilla - 1
```

```
                casillasVecinas[5] = casilla - columnas - 1
        else: # Par
            if (casilla%columnas) == 0: # Borde derecho
                casillasVecinas[1] = casilla - columnas + 1
                casillasVecinas[2] = casilla + 1
            else:
                casillasVecinas[1] = casilla + 1
                casillasVecinas[2] = casilla + columnas + 1
            casillasVecinas[3] = casilla + columnas
            casillasVecinas[4] = casilla + columnas - 1
            casillasVecinas[5] = casilla - 1
    return casillasVecinas


def casillaActual (self, idCasilla):
    """ RECIBE un identificador de casilla
        DEVUELVE un objeto casilla con su identificador y tipo actual
    """
    cas = copy.deepcopy(global_vars.casillasIniciales[idCasilla-1])
    # Si la casilla NO es una muralla, miramos si ha sido modificada
    if cas.getTipo() != T_MURALLA:
        if cas in self.casillasModificadas:
            indice = self.casillasModificadas.index(cas)
            cas.setTipo(self.casillasModificadas[indice].getTipo())
    return cas


def casillasVecinasActuales (self, idCasilla):
    """ RECIBE el identificador de la casilla sobre la cual queremos calcular
    las casillas vecinas actuales.
        DEVUELVE las 6 casllas vecinas actuales
    """
    # Obtenemos los identificadores de las casillas vecinas
    idVecinas = self.idCasillasVecinas(idCasilla)
    # Construimos las casillas vecinas actuales
    casillasVecinas = []
    for i in idVecinas:
        cas = Casilla (i, global_vars.casillasIniciales[i-1].getTipo())
        if cas in self.casillasModificadas:
            indice = self.casillasModificadas.index(cas)
            cas.setTipo(self.casillasModificadas[indice].getTipo())
        casillasVecinas.append(cas)
    return casillasVecinas
```

```
    def __WayTrackingBack (self, k, idCasillaDestino, calculadas, filas, columnas,
        casillasTablero, distancias):
        if calculadas == filas*columnas: return
        else:
            for i in range(len(distancias)):
                if distancias[i] == k:
                    adyacentes = self.casillasVecinasActuales(i+1)
                    for j in adyacentes:
                        if j.tipo != T_MURALLA and distancias[j.idCasilla - 1] > k+1:
                            distancias[j.idCasilla - 1] = k+1
                            calculadas += 1
            self.__WayTrackingBack (k+1, idCasillaDestino, calculadas, filas, columnas,
                casillasTablero, distancias)


    def WayTracking (self, idCasillaDestino):
        # Este algoritmo solo es util en un medio completamente observable
        columnas = global_vars.columnasTablero
        filas = global_vars.filasTablero
        casillas = global_vars.casillasIniciales[:]
        # Inicializo la lista de distancias a INFINITO
        distancias = []
        for i in range(filas*columnas):
            distancias.append(INFINITO)
        # Etiqueto la casilla de la bandera con distancia 0
        calculadas = 1
        distancias[idCasillaDestino-1] = 0
        # Etiqueto las murallas con distancia -1
        for i in casillas:
            if i.tipo == T_MURALLA:
                distancias[i.idCasilla-1] = -1
                calculadas += 1
        self.__WayTrackingBack(0, idCasillaDestino, calculadas, filas, columnas, casillas,
            distancias)
        return distancias


    def __contains__(self, element):
        return element in self.casillasModificadas


    def __eq__(self, other):
        return self.casillasModificadas==other.casillasModificadas
```

6.7. Clase Estado

```
from config import *
import global_vars
import psyco

psyco.full()


class Estado:


    def __init__ (self, tablero, equipos):
        self.tablero = tablero
        self.equipos = equipos


    def esSolucion (self):
        if self.tablero.banderas==0: return True
        else: return False


    def actualizarEstado (self, eq, jug, casillaDestino):
        """ Actualiza las componentes del estado: tablero y jugadores.
        Del tablero se actualizan las casillas modificadas (si se ha modificado
        alguna) y de los jugadores se actualizan la vida y los objetos.
            RECIBE "jug" que es el indice del jugador en la lista de jugadores
        (normalmente idJugador-1) y la casilla a la que se mueve (tipo Casilla)
            DEVUELVE nada.
        """
        equipoActor = self.equipos[eq]
        jugadorActor = self.equipos[eq].jugadores[jug]
        accionEjecutada = True
        if jugadorActor.getCasilla() <> casillaDestino.getIdCasilla(): # El jugador se mueve
            self.equipos[eq].jugadores[jug].setCasilla(casillaDestino.getIdCasilla())
            # Dirty indica si es necesario actualizar las casillas modificadas del tablero
            dirty = 0
            if casillaDestino.getTipo() == T_HIERBA:
                accionEjecutada = jugadorActor.perderEnergia(1)
            elif casillaDestino.getTipo() == T_AGUA:
                accionEjecutada = jugadorActor.usarBarca()
            elif casillaDestino.getTipo() == T_BARRO:
                accionEjecutada = jugadorActor.perderEnergia(2)
            elif casillaDestino.getTipo() == T_HOYO:
```

```
                accionEjecutada = jugadorActor.perderEnergia(4)
            elif casillaDestino.getTipo() == T_ZANJA:
                accionEjecutada = jugadorActor.perderEnergia(6)
            elif casillaDestino.getTipo() == T_BANDERA:
                accionEjecutada = jugadorActor.perderEnergia(1)
                if accionEjecutada:
                    dirty = 1
                    equipoActor.capturarBandera()
                    self.tablero.banderas -= 1
            elif casillaDestino.getTipo() == T_BARCA:
                dirty = 1
                accionEjecutada = jugadorActor.cogerBarca()
            elif casillaDestino.getTipo() == T_HACHA:
                dirty = 1
                accionEjecutada = jugadorActor.cogerHacha()
            elif casillaDestino.getTipo() == T_ZUMO:
                dirty = 1
                accionEjecutada = jugadorActor.beberZumo()
            elif casillaDestino.getTipo() == T_PALA:
                dirty = 1
                accionEjecutada = jugadorActor.cogerPala()
            elif casillaDestino.getTipo() == T_BOSQUE:
                (accionEjecutada, hacha) = jugadorActor.usarHacha()
                if accionEjecutada and hacha: dirty = 1
            # Si el dirty bit vale 1, actualizamos la casilla
            if dirty == 1 and accionEjecutada:
                 casillaDestino.convertirHierba()
                 self.tablero.anadirCasillaModificada(casillaDestino)
        else: # El jugador no se mueve utiliza la pala
            if casillaDestino.getTipo() == T_HIERBA or casillaDestino.getTipo() == T_ZANJA
                or casillaDestino.getTipo() == T_HOYO:
                if jugadorActor.usarPala(): # El jugador tiene pala para usarla
                    satisfactoriamente
                    if casillaDestino in self.tablero.getCasillasModificadas():
                        self.tablero.eliminarCasillaModificada(casillaDestino)
                    casillaDestino.cavar()
                    self.tablero.anadirCasillaModificada(casillaDestino)
                else: # El jugador NO tiene pala, por lo que no puede ejecutar la accion
                    accionEjecutada = False
            else: # No se puede cavar en la casilla
                accionEjecutada = False
        return accionEjecutada


    def jugadorBloqueado (self, jugador):
```

```
        casillasVecinas = self.tablero.casillasVecinasActuales(jugador.getCasilla())
        bloqueado = True
        for i in casillasVecinas:
            if i.tipo != T_MURALLA:
                if jugador.energia >= i.coste(jugador.hacha, jugador.barca):
                    bloqueado = False
                    break
        return bloqueado


    def minimaDistancia (self, eq):
        """ RECIBE el identificador de equipo que hay que examinar
            DEVUELVE una tupla (idJugador, bandera, distancia)
        """
        equipo = self.equipos[eq]
        minimaDistancia = INFINITO
        jugador = -1
        bandera = -1
        for band in global_vars.banderasObjetivo:
            if self.tablero.casillaActual(band).getTipo() == T_BANDERA:
                distancias = global_vars.distanciaBanderas[band] #diccionario de listas de
                    distancias
                for jug in equipo.jugadores:
                    if not self.jugadorBloqueado(jug):
                        dist = distancias[jug.casilla-1]
                        if dist < minimaDistancia:
                            minimaDistancia = dist
                            jugador = jug.idJugador
                            bandera = band
        return (jugador, bandera, minimaDistancia)


    def __eq__(self, other):
        return self.tablero==other.tablero and self.jugadores==other.jugadores
```

6.8. Clase Nodo

```
from config import *
import psyco

psyco.full()
```

```
class Nodo:


    def __init__ (self, estado, padre=None, accion=(-1,-1), profundidad=PROF_INICIAL,
        utilidad=-INFINITO):
        self.estado = estado # Tipo estado
        self.padre = padre # Tipo nodo
        self.accion = accion # Tupla (jugador, accion)
        self.profundidad = profundidad # Tipo entero
        self.utilidad = utilidad


    def repetidoEnRama (self):
        """ Comprueba si el estado del nodo esta repetido en alguna de sus nodos
        antecesores (solo comprueba en su rama).
            DEVUELVE True si esta repetido; False si no esta repetido
        """
        repetido = False
        actual = self
        while not repetido and actual.profundidad > PROF_INICIAL:
            if self == actual.padre: repetido = True
            actual = actual.padre
        return repetido


    def primerAntecesor(self):
        actual = self
        while actual.profundidad > PROF_INICIAL+1:
            actual = actual.padre
        return actual


    def __cmp__(self, other):
        return cmp(self.utilidad, other.utilidad)


    def __eq__(self, other):
        return self.estado==other.estado


    def __str__(self):
        cad = ""
        for i in self.estado.jugadores:
            cad += "Jugador " + str(i.idJugador) + "\tCasilla: " + str(i.casilla) + "\
```

```
                tEnergia: " + str(i.energia) + "\n"
        cad += "Accion: " + str(self.accion) + "\tCoste: " + str(self.g) + "\tProfundidad: "
             + str(self.profundidad)
        return cad
```

6.9. Clase Minimax

```
import copy, psyco, time, global_vars, math, random
from config import *
from nodo import *


psyco.full()


class Minimax:
    def __init__(self, nodo_inicial):
        self.nodoInicial = nodo_inicial
        self.nodoMejor = copy.deepcopy(nodo_inicial)
        self.timeout = False


    def Sync (self):
        tiempo = 0
        self.timeout = False
        while tiempo < global_vars.deadline - SECURITY_RANGE:
            time.sleep(TIME_INCREMENT)
            tiempo += TIME_INCREMENT
        self.timeout = True
        print "Tiempo: %f" % (tiempo)


    def decision_minimax(self):
        limite = 1
        while not self.timeout:
            self.max_valor(self.nodoInicial, limite, -INFINITO, INFINITO)
            limite += 1
        print "Profundidad alcanzada: ", limite


    def max_valor(self, nodo, limite, alfa, beta):
```

```
        if self.test_terminal (nodo, limite):
            return self.evaluacion(nodo, global_vars.MAX, global_vars.MIN)

        v = -INFINITO
        sucesores = self.expandir (nodo, global_vars.MAX)

        i = 0
        while not self.timeout and i < len(sucesores):
            v = max(v, self.min_valor (sucesores[i], limite, alfa, beta))
            # Poda
            if v >= beta:
                return v
            alfa = max (alfa, v)
            # Si no hemos podado, comprobamos la utilidad del nodo y
            # si es el mejor nodo que hemos encontrado hasta ahora
            # seleccionamos su antecesor con profundidad 1
            if v > self.nodoMejor.utilidad:
                self.nodoMejor = copy.deepcopy(sucesores[i].primerAntecesor())
                self.nodoMejor.utilidad = v
            i += 1
        return v


    def min_valor(self, nodo, limite, alfa, beta):
        if self.test_terminal (nodo, limite):
            return self.evaluacion(nodo, global_vars.MAX, global_vars.MIN)

        v = INFINITO
        sucesores = self.expandir (nodo, global_vars.MIN)

        i = 0
        while not self.timeout and i < len(sucesores):
            v = min(v, self.max_valor (sucesores[i], limite, alfa, beta))
            if v <= alfa:
                return v
            beta = min (beta, v)
            i += 1
        return v



    def expandir(self, nodo, equipo):
        sucesores = []
        numero_jugadores = len(nodo.estado.equipos[equipo].jugadores)
        for jug in range(numero_jugadores):
```

```
            if nodo.estado.equipos[equipo].jugadores[jug].getEnergia() > 0:
                casillasVecinas = nodo.estado.tablero.casillasVecinasActuales(nodo.estado.
                    equipos[equipo].jugadores[jug].getCasilla())
                # Si el jugador tiene pala, anadimos la casilla actual para hacer hoyos
                # Esto solo se lleva a cabo si la heuristica tiene en cuenta la energia del
                    equipo contrario
                if nodo.estado.equipos[equipo].jugadores[jug].pala > 0:
                    casillasVecinas.append(nodo.estado.tablero.casillaActual(nodo.estado.
                        equipos[equipo].jugadores[jug].getCasilla()))
                for mov in range (len(casillasVecinas)):
                    # Comprobamos que la casilla destino NO sea una muralla
                    if casillasVecinas[mov].getTipo() != T_MURALLA:
                        estadoAux = copy.deepcopy(nodo.estado)
                        # actualizarEstado devuelve True o False en funcion de si se ha
                            podido ejecutar la accion o no
                        if estadoAux.actualizarEstado (equipo, jug, casillasVecinas[mov]):
                            # Si se ha podido ejecutar la accion, construimos un nuevo
                                sucesor
                            accion = (estadoAux.equipos[equipo].jugadores[jug].getIdJugador
                                (), mov+1)
                            nuevoNodo = Nodo (estadoAux, nodo, accion, nodo.profundidad+1)
                            sucesores.append(nuevoNodo)
        return sucesores


    def test_terminal (self, nodo, limite):
        terminal = False
        # El test-terminal solo comprueba si quedan banderas en el tabero
        # o si se ha llegado a la profundidad de corte (prof. iterativa)
        if nodo.estado.esSolucion() or nodo.profundidad==limite:
            terminal = True
        # No se comprueban energias ya que expandir() se encarga de ello
        return terminal


    def evaluacion(self, nodo, equipo1, equipo2):
        estado = nodo.estado
        totalBanderas = len(global_vars.banderasObjetivo)
        filas = global_vars.filasTablero
        columnas = global_vars.columnasTablero

        bandEq1 = estado.equipos[equipo1].banderasCapturadas
        bandEq2 = estado.equipos[equipo2].banderasCapturadas

        # Se toma como bondadBanderas el producto de filas*columnas para evitar
```

```
        #que en un tablero muy grande las distancias sean mas importantes que
        #las banderas.
        bondadBanderas = filas*columnas
        ratioBandEq1 = bandEq1*totalBanderas
        ratioBandEq2 = bandEq2*totalBanderas
        ratioBand = bondadBanderas * (ratioBandEq1 - ratioBandEq2)

        maxDist = ((columnas+filas)/2)*(3/4)
        done = False
        blacklist = []
        (jug1,band1,distEq1) = estado.minimaDistancia(equipo1)
        (jug2,band2,distEq2) = estado.minimaDistancia(equipo2)

        # Si el estado no es solucion y aun quedan banderas por coger
        if not estado.esSolucion():
            while not done:
                # Si la distancia mas corta del equipo MAX y MIN se refieren
                #a la misma bandera, y MIN esta mas cerca que MAX, buscamos otra
                #bandera para MAX que pueda coger sin que MIN se la quite
                if (band1 == band2) and (distEq1 > distEq2):
                    blacklist.append(band1)
                    # Actualizamos el estado eliminando la bandera que hemos metido en
                        blacklist
                    cas = estado.tablero.casillaActual(band1)
                    cas.convertirHierba()
                    estado.tablero.anadirCasillaModificada(cas)
                    estado.tablero.banderas -= 1
                    # Si todas las banderas estan en la blacklist, quitamos una aleatoria
                    if estado.tablero.banderas == 0:
                        if len(blacklist) <= 1: white = 0
                        else: white = random.randint(1, len(blacklist)-1)
                        toDel = blacklist.pop(white)
                        casDel = estado.tablero.casillaActual(toDel)
                        estado.tablero.eliminarCasillaModificada(casDel)
                        estado.tablero.banderas += 1
                        done = True
                    # Medimos la distancia minima del equipo MAX
                    (jug1,band1,distEq1) = estado.minimaDistancia(equipo1)
                else:
                    done = True
        # Vaciamos la blacklist y ponemos el estado como estaba
        if len(blacklist) > 0:
            for i in range(len(blacklist)):
                toDel = blacklist.pop(0)
                casDel = estado.tablero.casillaActual(toDel)
```

```
                estado.tablero.eliminarCasillaModificada(casDel)
                estado.tablero.banderas += 1

        ratioDistEq1 = maxDist - distEq1
        ratioDistEq2 = maxDist - distEq2
        ratioDist = ratioDistEq1 - ratioDistEq2

        ###############
        # ENERGIA
        ###############
        #(jug,accion) = nodo.accion
        #equipoExpande = (jug-1)//len(nodo.estado.equipos[0].jugadores)
        #energiaEq1 = nodo.estado.equipos[equipoExpande].getEnergiaTotal()
        #ratioEnergiaEq1 = energiaEq1/100

        #evalEq1 = ratioBandEq1 + ratioDistEq1 # + ratioEnergiaEq1
        #evalEq2 = ratioBandEq2 + ratioDistEq2
        return ratioBand + ratioDist
```

6.10. Clase Cliente

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import Ice, sys, time, threading, psyco, datetime

Ice.loadSlice('-I/usr/share/slice ../slice/Practica.ice')

import Practica

from tablero import *
from equipo import *
from estado import *
from nodo import *
from minimax import *

psyco.full()


def Inicializar(mapa, equipos, tablero):
    global_vars.casillasIniciales = []
```

```
    global_vars.banderasObjetivo = []
    global_vars.distanciaBanderas = {}
    # Tiempo del turno
    global_vars.deadline = mapa.tiempo
    # ID de usuario, MIN y MAX
    global_vars.idUsuario = ((mapa.idUsuario - 1) %2)+1
    print "idUsuario %d" % (global_vars.idUsuario)
    global_vars.MAX = global_vars.idUsuario - 1
    print "MAX %d" % (global_vars.MAX)
    global_vars.MIN = (global_vars.MAX + 1) %2
    print "MIN %d" % (global_vars.MIN)
    # Filas y columnas del tablero
    global_vars.filasTablero = mapa.dimy
    global_vars.columnasTablero = mapa.dimx
    # Inicializamos las casillas iniciales que nos da el servidor
    for index,casilla in enumerate(mapa.casillas):
        cas = Casilla(index+1,casilla)
        global_vars.casillasIniciales.append(cas)
        if casilla == T_BANDERA:
            tablero.banderas += 1
            global_vars.banderasObjetivo.append(index+1)
    # Inicializamos las distancias a cada bandera
    for i in global_vars.banderasObjetivo:
        distancias = []
        distancias = tablero.WayTracking(i)
        global_vars.distanciaBanderas[i] = distancias[:]
    # Inicializamos los equipos. Cada equipo inicializa sus jugadores en el constructor
    idEquipo = 1
    for i in mapa.jugadores:
        if idEquipo == i.equipo:
            eqAux = Equipo (idEquipo, mapa.jugadores)
            equipos.append(eqAux)
            idEquipo += 1


class Client (Ice.Application):

    def printHelp(self):
        print "\nEjecucion del cliente para las practicas de IA: \n"
        print "- Para jugar una partida individual ejecutar dos instancias del programa con
            la linea: ./Client 1 --Ice.Config=(Ruta Archivo .config)\n"
        print "- Para jugar una competicion ejecutar una instancia del programa con la linea
            : ./Client 2 --Ice.Config=(Ruta Archivo .config)\n  Esperar a que el oponente se
             una a la partida.\n\n"
```

```
    def jugarPartida(self, dni, partida):
        jugando = True;
        mapa = partida.obtenerMapa(dni)
        time.sleep(2)
        equipos = []
        tablero = Tablero()
        Inicializar (mapa, equipos, tablero)
        estado_actual = Estado (tablero, equipos)
        nodo_actual = Nodo (estado_actual)
        minimax = Minimax(nodo_actual)
        #print mapa
        print "Tiempo de los turnos configurado en %d segundos." % (global_vars.deadline)
        print "Esperando a que se una otro jugador..."

        while jugando:
            moviendo = True
            infoJugada = partida.pedirTurno(mapa.idUsuario)

            print "Movimiento realizado por el oponente, resultado y token para realizar el
                movimiento:"
            print infoJugada
            # Esta condicion es para corregir desde el cliente un bug del servidor
            if infoJugada.mov.idJugador == -1 and infoJugada.mov.mov == -1:
                infoJugada.resultado = 0

            if infoJugada.resultado != 0:
                if infoJugada.resultado == 1:
                    print "Has GANADO!"
                    jugando = False
                    moviendo = False
                else:
                    print "Has PERDIDO!"
                    jugando = False
                    moviendo = False
                break

            while moviendo:
                if infoJugada.mov.idJugador <> -1 and infoJugada.mov.mov <> -1:
                    # Actualizamos el estado con los cambios que ha provocado en el tablero
                        la accion del oponente
                    jug = (infoJugada.mov.idJugador-1) %(len(mapa.jugadores)/2)
                    casillas = estado_actual.tablero.idCasillasVecinas(equipos[global_vars.
                        MIN].jugadores[jug].casilla)
                    nodo_actual.estado.actualizarEstado(global_vars.MIN, jug, tablero.
                        casillaActual(casillas[infoJugada.mov.mov-1]))
```

```
                minimax.nodoMejor = copy.deepcopy(nodo_actual)
                print str(estado_actual.equipos[global_vars.MIN])

            d1 = datetime.datetime.now()
            # Creamos un hilo que lanzara la busqueda minimax
            minimax.timeout = False
            buscador = threading.Thread(target=minimax.decision_minimax, args=())
            buscador.start()
            # Ejecutamos la funcion durmiente que indicara cuando parar la busqueda
            minimax.Sync()

            (jugador, movimiento) = minimax.nodoMejor.accion
            print datetime.datetime.now()-d1
            if jugador == 0:
                break

            try:
                # Mandamos el movimiento al servidor
                devuelto = partida.jugada(mapa.idUsuario, jugador, movimiento,
                    infoJugada.token)
                print "Movimiento realizado: " + str(devuelto)
                print "Jugador %d\nAccion %d" % (jugador,movimiento)
                moviendo = False;
                # Actualizamos el estado con los cambios que ha provocado nuestro
                    movimiento
                jug = (jugador-1)%(len(mapa.jugadores)/2)
                casillas = estado_actual.tablero.idCasillasVecinas(equipos[global_vars.
                    MAX].jugadores[jug].casilla)
                nodo_actual.estado.actualizarEstado(global_vars.MAX, jug, tablero.
                    casillaActual(casillas[movimiento-1]))
                minimax.nodoMejor = copy.deepcopy(nodo_actual)
                print str(estado_actual.equipos[global_vars.MAX])

            except Practica.TokenIncorrectoError, e:
                print e.reason
                print "El temporizador ha expirado. Has PERDIDO la partida."
                jugando = False
                moviendo = False

            except Practica.MovimientoIncorrectoError, e:
                print e.reason
                print "Ha introducido un movimiento no valido. Introduzca de nuevo el
                    movimiento."

    def run(self, argv):
```

```
        dni = "71219116"
        password = "asdfqwer"

        if len(argv) == 1 or len(argv) > 2:
            self.printHelp()

        else:
            base = self.communicator().stringToProxy("AutenticacionObject")
            autenticacion = Practica.AutenticacionPrx.checkedCast(base)
            partida = None
            if not autenticacion:
                print "ERROR"

            try:
                partida = autenticacion.login(dni, password)
                print str(partida)

            except Practica.PasswordIncorrectaError, e:
                print e.reason
            except Practica.UsuarioIncorrectoError, e:
                print e.reason
            except Practica.NoExisteContrincanteError, e:
                print e.reason
            except Practica.NoExistePartidaError, e:
                print e.reason

            if argv[1] == "1" and partida != None:
                self.jugarPartida(dni, partida)
                try:
                    autenticacion.finalizarPartida(dni, password)
                except Practica.UnknownException:
                    pass

            elif argv[1] == "2" and partida != None:
                self.jugarPartida(dni, partida)
                print "RECUPERANDO SEGUNDA PARTIDA"
                time.sleep(5)
                self.jugarPartida(dni, partida)
                try:
                    autenticacion.finalizarPartida(dni, password)
                except Practica.UnknownException:
                    pass

            elif partida != None:
                print "ERROR: El argumento introducido no es correcto\n"
```

```
                self.printHelp()

Client().main(sys.argv)
```

Bibliografía

[1] Psyco Home Page. http://psyco.sourceforge.net/.

[2] Python Programming Language - Official Website. http://www.python.org/.

[3] S.J. Russell and P.Ñorvig. *Inteligencia Artificial. Un Enfoque Moderno, 2a ed.* Pearson Prentice Hall, 2003.

[4] Inteligencia Artificial - web de la asignatura. http://obi-wan.inf-cr.uclm.es:44080/parte2/.

[5] Zeroc, the Home of Ice. http://www.zeroc.com/.