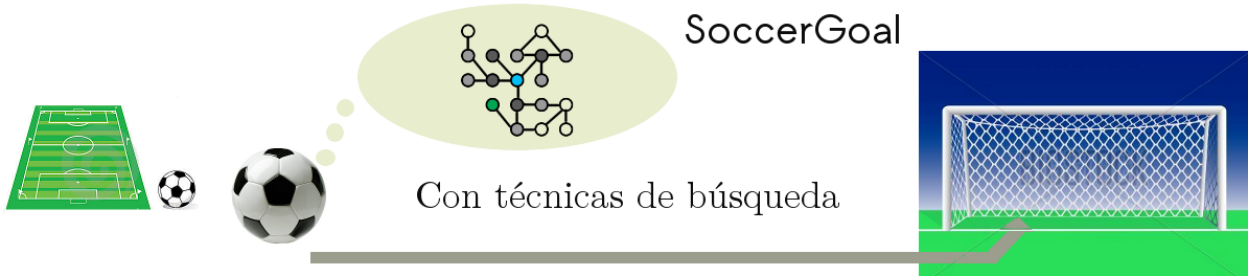
Sistemas Inteligentes - Curso 2014/2015

PRÁCTICA 1.-TÉCNICAS DE BÚSQUEDA



1. Motivación y objetivos de la práctica

Los temas 2, 3 y 4 de la asignatura están dedicados a estudiar agentes y métodos de búsqueda (no informada, informada y búsqueda en juegos). Si bien estas técnicas se aplican a una gran variedad de problemas reales, es necesario que como primer paso podamos formular el problema con uno de búsqueda en espacio de estados. Esto ya constituye un ejercicio interesante, al que podemos unir los de definir las funciones heurísticas y el planteamiento de las estrategias a seguir en el caso de búsquedas con adversario.

En esta práctica trataremos de resolver un problema real como es el comportamiento de unos "laberintos" un tanto particulares, dentro de un juego bautizado como SoccerGoal, que hemos diseñado nosotros. Se trata de un juego muy sencillo donde un agente jugador, agent, representado por un balón de fútbol: ⚽ , tendrá que encontrar el camino a la portería dentro de un campo, que presentará obstáculos para sortear, y además hoyos, por los que se puede pasar pero a coste de sufrir una penalización (por estar ese terreno desnivelado y sin césped). Este sencillo juego ha sido elaborado con fines didácticos, y persigue que se pueda usar para estudiar, conocer y programar diferentes tipos de técnicas de **Sistemas Inteligentes**[1]. En este caso nos centraremos en los algoritmos de búsqueda (no informada e informada), vistos en clase de teoría.

Como ya sabéis, es prerrequisito de esta asignatura tener fluidez en la programacin con lenguajes de alto nivel de Orientación a Objetos (como es nuestro caso, Java), así como haber adquirido los conocimientos necesarios en estructuras de datos y algorítmica.

Al ser esta la primera práctica donde vais a usar el entorno SoccerGoal, antes que nada necesitaremos presentar su funcionamiento básico y cómo podréis trabajar con él. Lo primero que tendréis que hacer, por tanto es:

- Familiarizaros con la estructura del juego SoccerGoal.
- Conocer la arquitectura del código del software que usaremos como plataforma para programar vuestros agentes de búsqueda.
- Entender cómo se programa un agente

[1] Por supuesto, con las pertinentes modificaciones para adaptarlo a cada caso.

Los objetivos finales que se persiguen con esta práctica, aparte de adquirir las capacidades anteriormente listadas, son:

1. Ver en detalle un ejemplo particular que transforma un problema "real" en una posible abstracción.
2. Formalizar el problema de resolver un juego, en particular mediante técnicas de búsqueda.
3. Realizar implementaciones en `Java` de algunos de los algoritmos de búsqueda (informada y heurística) vistos en clase, de modo que se entiendan todavía mejor y en detalle.
4. Comprobar el comportamiento y complejidad (en tiempo y espacio) de estos algoritmos, enlazándolo con los conceptos vistos en clase.
5. Ser capaces de realizar, teniendo en cuenta los factores anteriores, y considerando además la complejidad del problema (dimensiones del problema y del espacio de búsqueda), una comparativa entre los distintos algoritmos.

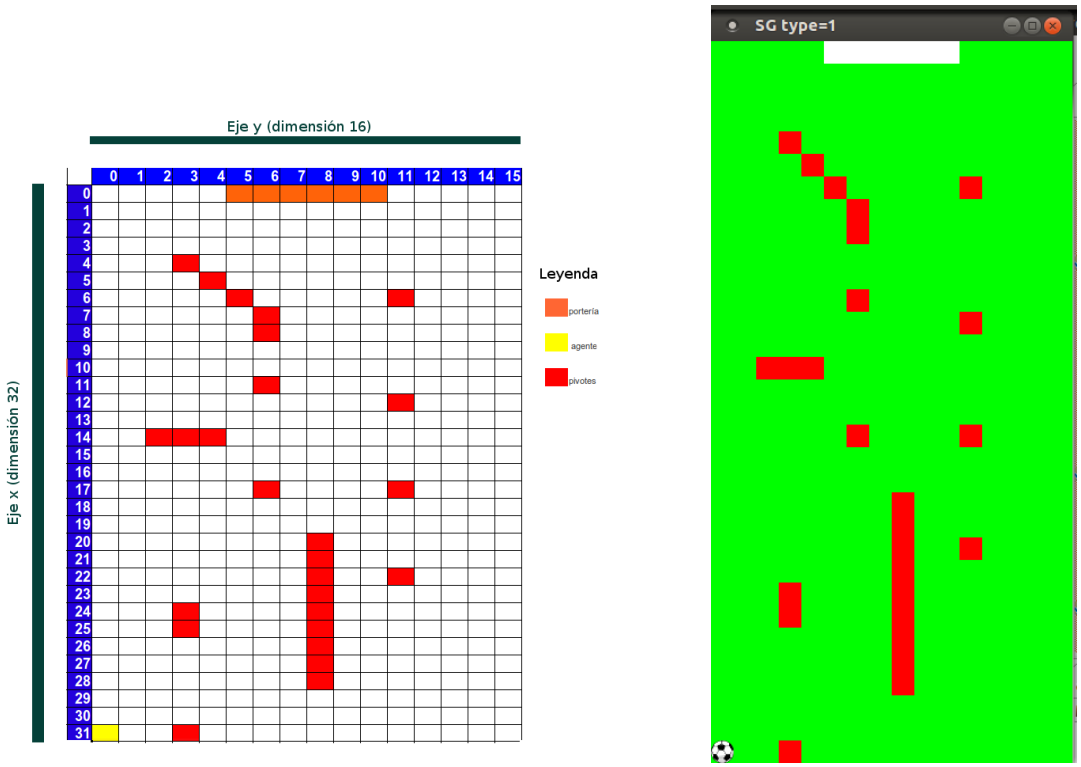
2. Funcionamiento del juego

El funcionamiento del juego es bastante simple, aunque merece la pena primeramente comentar en detalle algunos aspectos del diseño. El escenario es un campo (de fútbol) que tendrá dos dimensiones, en el eje horizontal (o de las x) y en el eje vertical (o de las y). Estas dimensiones son fijas durante una partida, pero son "configurables" al inicio de un juego. La única condición, en principio (en la implementación será más restrictivo), es que han de ser dimensiones con valores enteros. Veamos, un escenario concreto como ejemplo: Figura 1.(a).

A lo largo del campo se distribuirán un conjunto de obstáculos, que serán siempre cuadrados con una unidad de longitud en cada lado, y que estarán, por motivos de simplicidad, ocupando siempre posiciones completas. Es decir, un obstáculo que está en la posición $< i, j >$, en realidad ocupará las posiciones x que van hasta $(i+1, j+1)$. Por ejemplo, en la figura 1.(a), hay un obstáculo (mostrados en rojo) en la posición $(31, 3)$, lo cual significa que su esquina superior izquierda es exactamente esa posición en reales, la posición $(31, 4)$ sería su esquina superior derecha, y las inferiores serían $(32, 3)$ y $(32, 4)$. Es importante notar que el eje de las x va creciendo de izquierda a derecha, pero el de las y de arriba abajo, por compatibilidad con la forma de trabajar normalmente con gráficos en programación. Por otro lado, los hoyos se ubican de manera análoga en posiciones unitarias y 'enteras'.

La pelota puede partir de cualquier posición del campo, y en cada instante del juego ocupará posiciones que sí son reales/continuas, por ejemplo el agente podría estar en la posición $[4.75, 16.12]$. Si miramos la figura 2, esta posición estaría *aproximadamente* representada por el punto amarillo, y la situación esquemática del agente como un cuadrado (rectángulo por la forma de dibujar el esquema) gris. En este ejemplo, la pelota nunca podría encontrarse en la posición $[5.75, 16.12]$, pues chocaría con un obstáculo, aunque eso lo veremos en las reglas del juego.

Respecto a los hoyos, consideraremos que el agente es penalizado por atravesar un hoyo, cuando está posicionado de tal forma que el centro de la pelota se encuentra en el área de desnivel u hoyo. Es decir, podrá suceder que el agente tocara mínimamente sin ser penalizado (y sin considerar que pasa por él), pues se considera que esta circunstancia no afecta a la dificultad de 'moverse'. Nótese que, para simplificar, las figuras 1, 2 y 3 no presentan hoyos, ya que su mecanismo de ubicación es



(a) (b)

Figura 1: (a)Esquema de un juego SoccerGoal con dimensiones x=16 e y=32. (b) Mismo juego que en la figura de la izquierda, pero en el modo GUI que presentará el programa.

similar al de los obstáculos. En las posteriores figuras (y en la GUI del programa) aparecerán en un color 'arena'.

Queda un último detalle del agente a tener en cuenta, las figuras "esquemáticas" que hemos mostrado lo representan como un cuadrado, pero sin embargo en el juego será circular (con diámetro 1), tomando la forma de un balón (en 2D, claro), como muestra la figura 3.(a). Esto será importante de cara a las colisiones. Sin embargo, la posición de la pelota dentro del juego vendrá determinada por la esquina superior izquierda del cuadrado que lo circunscribe (ver Fig. 3.(b)). En realidad estos detalles del juego van a ser prácticamente "transparentes" para el jugador/agente, salvo, quizás, para pequeños detalles, por lo que hemos considerado de utilidad indicarlos.

Antes de pasar a las reglas del juego, nos queda un último elemento por comentar: la portería. Estará compuesta por varios "cuadrados", consecutivos en las x, es decir, siempre en horizontal. En todos nuestros escenarios suele estar colocada en la posición 0 (entre el 0 y el 1 en números continuos) y centrada en las x, pero nada impide que pudiera estar a mitad del campo, por ejemplo, o más hacia la izquierda o la derecha. La única restricción para que un campo sea válido, es que la portería, al menos, debería ocupar un cuadrado. Por tanto, debe existir al menos una posición "completa" $< g_x, g_y >$, o si la longitud de la portería es mayor que uno, serían todas las que van desde $< g_x^{left}, g_y >$ a $< g_x^{right}, g_y >$

Reglas del juego

1. SITUACIÓN INICIAL y determinación del campo

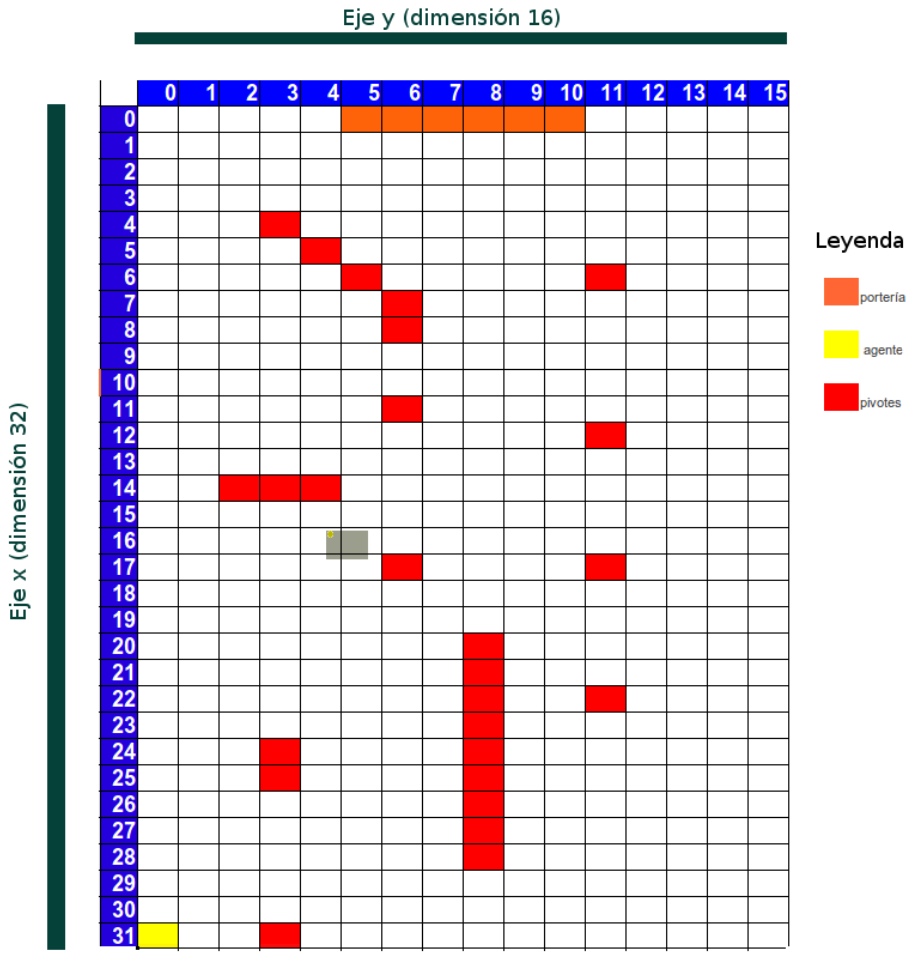
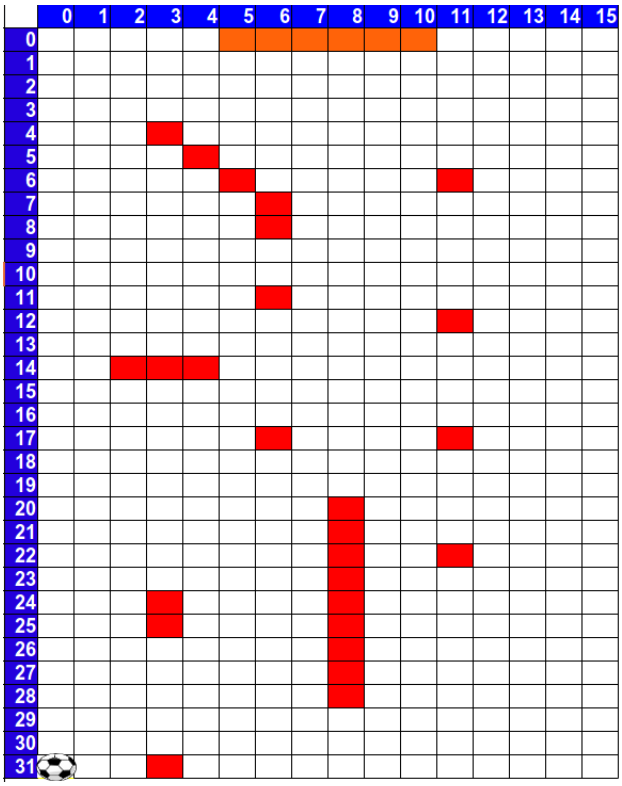


Figura 2: (a) Ejemplo de agente en la posición $[4.75, 16.12]$.

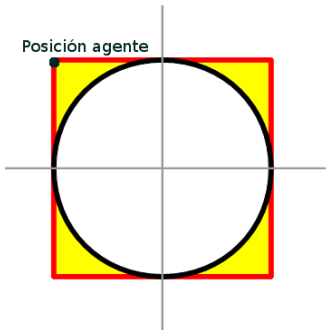
- Al inicio de cada juego estará completamente definido el campo, con los siguientes elementos:
  - *a*) Dimensiones en ambos ejes.
  - *b*) Posiciones *completas* de la portería, al menos una, y si son varias siempre ubicadas de manera consecutiva y en horizontal.
  - *c*) Número de obstáculos y sus posiciones *completas*.

  NOTA: Los obstáculos nunca invadirán la portería (sí podrían rodearla) ni chocar el agente en su posición inicial.
  - *d*) Número de hoyos y sus posiciones *completas*, así como la penalización numérica por pasar por cada uno de ellos.

  Estos 4 elementos serán invariables mientras se desarrolle la partida en cuestión.
- Al inicio de cada juego el agente estará ubicado en una posición $[x_a^0, y_a^0]$ conocida y asignada por el juego.
- El agente es el único elemento dentro del juego que podrá ir variando su posición a cada paso (iteración) del juego.



(a)



(b)

Figura 3: (a) Esquema, pero incluyendo el agente en su forma "circular" (pelota). (b) Cuadrado que circunscribe el círculo que representa al agente.

2. OBJETIVO DEL JUEGO y causas de finalización

   - Para que el agente finalice el juego con éxito, su punto central $[x_a+0.5, y_a+0.5]$ tendrá que encontrarse dentro de la portería.

NOTA: Podría suceder que el agente esté bloqueado por los obstáculos y no haya forma de llegar a la portería, el juego no asegura que sean resolubles los escenarios, pero el agente puede percatarse de dicha situación y "comunicarlo" con un abandono.

   - Si el agente supera un número máximo de iteraciones predefinido, el juego será finalizado sin éxito para el agente.
   - En la versión actual del juego, realizar un movimiento inválido significa automáticamente la finalización fallida del mismo.

NOTA: Está permitido pasar por un hoyo, pero eso supondrá un coste adicional, cuya cuantía numérica (penalización) se conoce en la definición del juego y es fijo a lo largo de una 'partida'. En ningún caso, esta circunstancia supondrá, por tanto, la finalización del juego.

3. DESARROLLO DEL JUEGO y acciones posibles

   - El agente puede moverse, siempre en línea recta, en cualquier dirección/ángulo que desee, y en principio, cualquier número de unidades, en valor real. Cada movimiento realizado por el agente se considerará una iteración. El coste de ese movimiento, una

vez realizado, será igual al de la distancia recorrida. Si la posición a la que se llega en un movimiento se considera dentro de un hoyo, se sumará a ese coste, además, la penalización correspondiente.

- Para que el movimiento por el agente sea válido tiene que cumplirse que:
  1) No se sale de las dimensiones del campo (excepto cuando entra en la portería) y
  2) Que en su trayectoria no choca con ningún obstáculo. En este caso, se considera siempre el agente como un círculo y los obstáculos como cuadrados.

NOTA: Se da una cierta flexibilidad para que sí pueda "rozar" ligeramente a un obstáculo sin considerarlo colisión.

4. RESUMEN DEL JUEGO

- Por tanto el juego se compondrá de n iteraciones o pasos (variables, según el caso, hasta que finalice el juego) limitados por un número máximo.
- En cada iteración i el agente ocupará la posición $[x_a^i, y_a^i]$. La posición al inicio ($[x_a^0, y_a^0]$) será impuesta por el juego. En las posteriores iteraciones $i > 0$, el agente indicará al juego qué movimiento/acción quiere realizar. El agente podría indicar también que abandona el juego.
- Si dicha acción (elegida por el agente) es válida, entonces en la iteración siguiente $i+1$, el agente ocupará en el campo la posición correspondiente a moverse en la dirección (ángulo en radianes) y cantidad de unidades de espacio indicadas, $[x_a^{i+1}, y_a^{i+1}]$. Además si esta nueva posición le lleva a que su centro de masas alcance la portería, $[x_a^f, y_a^f]$, habrá finalizado el juego con éxito. Y esa resolución llevará asociada un coste = distancia recorrida + (número de hoyos * coste_{hoyo}). Evidentemente, cuanto menor sea el coste mejor será la solución.
- En otro caso, si el movimiento del agente es no válido o superó el número máximo de iteraciones, el juego finalizará con error o fallo.

3. El software a utilizar

Para realizar esta práctica necesitarás descargarte de moodle el fichero soccergoal1.zip y un compilador de Java versión 1.7 o posterior (en el caso de que la versión 1.8 diera problemas, indicad 'compatibilidad con 1.7'). Cada alumno podrá usar el IDE (Eclipse, NetBeans, etc.) que considere oportuno para el desarrollo de la práctica, o incluso un editor de textos y compilar/ejecutar por línea de comandos.

3.1. Primeros pasos

Tras descargar y descomprimir el fichero `soccergoal1.zip` debes encontrar dos carpetas: `src` y `doc`. No os damos los ficheros `.class` obtenidos tras compilar los fuentes, porque dependiendo de vuestra opción para trabajar os vendrá mejor de una manera u otra (varía el nombre de las carpetas que lo contienen y/o su ubicación), y teniendo los fuentes los podréis generar vosotros sin problemas.

El directorio `src` contiene los fuentes estructurados en dos subdirectorios que corresponden a los paquetes Java. El directorio `doc` contiene la documentación html generada a partir de los

comentarios incluidos en los ficheros fuente mediante `Javadoc`. Lo podrás emplear como ayuda/guía si te resulta de utilidad.

Veamos cómo ejecutar el juego:

1. Primero asegúrate que la versión de Java (JVM) es la requerida.
2. Si vas a usar un IDE simplemente asegúrate de poner la versión buena del Java. Deberás montar un proyecto con *código fuente existente* y seleccionar la carpeta **src**, puesto que es la que contiene los fuentes (.java). Normalmente también es posible crear un nuevo proyecto vacío y luego copiar los fuentes en las carpetas ya creadas por el sistema. Si usas esta posibilidad, recuerda actualizar o refrescar. Puedes usar cualquier IDE de tu elección, p.e. Eclipse o NetBeans, pero asegúrate de dominarlo, ya que el manejo del entorno de programación no corresponde a esta asignatura.
3. Si prefieres línea de comandos, simplemente recuerda que tendrás que configurar manualmente el `Classpath`.
4. Una vez que hayas montado el proyecto software, con la versión correcta (1.7), todo debería estar OK. Si ejecutas, por ejemplo

   ```
   java game.Main -agent search.GreedyAgentRd -type 1
   ```

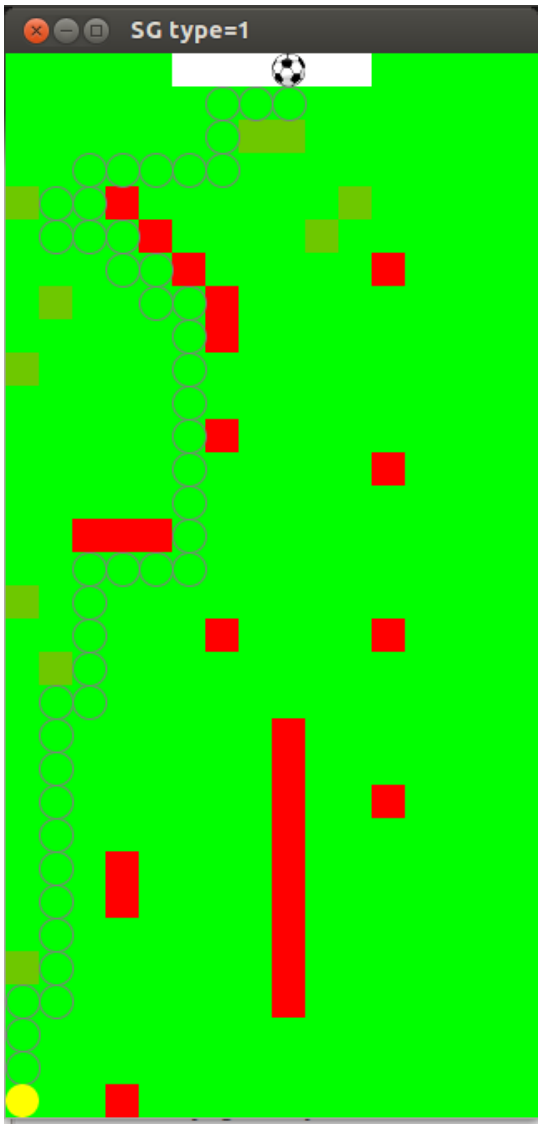
   la salida correspondiente se ve en la figura 4. Fíjate que conforme se va ejecutando el juego podemos ver la *estela* de su recorrido, que son los círculos grises que se ven a la izquierda de la figura.
5. Si lo ejecutas desde un IDE debes poner como clase principal `game.Main`, y configurar los argumentos, p.e. `-batch`. Con esta opción `-batch`, ya no se usa la salida GUI. Si no empleas al menos la opción `-agent`, el programa dará error, pues necesita saber cuál es el comportamiento del agente. También es necesario indicar el tipo de escenario o las dimensiones o el modo práctica, sólo una de ellas, ya que son las tres variantes posibles.
6. Todas las posibles opciones se muestran en la figura 5.

3.2. Estructura del código

Aunque se puede obtener mucha información estudiando los ficheros de ayuda incluidos en el directorio doc (abrir `....soccergoal1/doc/index.html`), veamos para empezar una breve descripción de cómo está estructurado el código. Básicamente podemos ver dos paquetes/subdirectorios en soccergoal1/src/:

- `game`: Contiene las clases e interfaces que forman el juego.
  IMPORTANTE: **¡¡¡ No modifiques este paquete!!!** [2].
- `search`: Contiene las implementaciones de los agentes específicos básados en búsqueda. Es en este directorio donde debes poner el código que desarrolles. Puedes estudiar los dos agentes de ejemplo GreedyAgentNaive y GreedyAgentRd (basado en el anterior, pero que intenta salir muy rudimentariamente de los *atascos*). Es **importante** indicar que estos ejemplos NO son agentes basados en búsqueda.

[2] La evaluación de la práctica será con el código original, los agentes programados que den errores con este código se considerarán inválidos y llevarán a un suspenso.




```
-------------------------------------------
 SETUP summary:
 - GUI mode
 - Using predefined field number 1.
 - AGENT PLAYER: search.GreedyAgentRd
--> No numAngles used, default value (= 4).
- unit = 1.0
-------------------------------------------

 Starting running SoccerGoal.....


***********************************
Congratulations, you reached the goal!
Info:
- iterations = 47
- holes  = 0
- thinking time = 0,0260000000 sec.
-----------------------------------
Search Details:
Num of expanded nodes: 47.0
Num of explored nodes: 48.0
Num of generated nodes: 170.0
-----------------------------------
Simulated nodes: 170.0
***********************************

path found by search.GreedyAgentRd is :
{1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,
0,0,0,1,1,1,1,1,1,1,1,2,1,2,1,2,2,1,
0,1,0,0,0,0,1,1,0,0,1}
```

Figura 4: Ejemplo de ejecución: output gráfico final (izquierda); salida en texto (derecha).

Aunque no podemos modificar el paquete `game`, es el núcleo del juego y, por tanto, tendremos que usar varias de sus clases/métodos. Las principales clases e interfaces que contiene son:

- `Main`.- Es el programa que vais a ejecutar para jugar, ya sea en modo sencillo (una única partida) o en el modo evaluable **práctica** (15 partidas variadas donde además se recopilan estadísticas).
  Contiene un método `main()` que es el que invocamos para lanzar el juego. Puedes ver sus opciones en la figura 5, o ejecutando `java game.Main` sin parámetros.

- `Game`.- Contiene todo lo relativo al control del juego, tanto la definición del campo (de tipo `Field`) con sus dimensiones, obstáculos, hoyos y portería, como la posición actual del agente, si la partida es en modo `GUI` o `batch`, etc., y es el encargado de velar porque se cumplan las reglas del juego. El método principal es `play(boolean gui)`. Puedes ver más información en javadoc. Con respecto a lo indicado en las reglas generales (en la sección) hay que indicar

```
java game.Main [arguments]

Possible arguments:
-------------------------------------------------------------------------------------------

   -agent agent_file. Use the .class specified as search agent
   (it must extend SearchAgent and it must be in package search, so: search.FileName)
   -numAngles num. It indicates how many possible angles the agent can use for movements
   (if not specified, default = 4)
   -batch. It doesn't use GUI. Results are shown only in text output. (default = GUI mode)
   -seed number. Use number as a fixed seed in order to keep the same fields
   in every program run (for debugging, professors will use any seed when assessment).
   -unit number. number must be positive and equal or less than 1.0 (default = 1.0)
     NOTE: -unit option has no effect in 'practica' mode

   Also, include ONLY ONE of the three following options:

     -type value. There are four simple predefined fields for debugging (see assignment).
     value must be in {1,2,3,4,5}
     -xDim numX and/or -yDim numY.
     These are also for debugging purposes, numX and numY must be integer numbers.
     (If only one dimesion is specified, the field will be square,
     taking the same length for both dimensions)
     -practica.  This will show a sequence of fields that must be all solved by the agent
     (or indicating there was no found solution), in increasing complexity.
     (IMPORTANT: Students will be assessed using this 'practica' mode.)
-------------------------------------------------------------------------------------------
```

Figura 5: Argumentos de llamada al programa principal: `game.Main`

que la actual implementación no permite que el agente se mueva a "saltos" mayores a 1.0, pero sí permite valores en el intervalo real (0.0,1.0].

- `Field`.- Esta clase tiene una doble finalidad: (1) por un lado especificar las características del campo, que pueden ser predeterminadas (4 escenarios predefinidos) o generadas con un método aleatorio; y (2) servir de herramienta para visualizar el juego, si se ha elegido el modo `GUI`. `Field` hace uso de las clases `BarrierPost` y `HolePost` para especificar los obstáculos y hoyos, respectivamente.

- `GameStep`.- Es una clase muy sencilla que captura la información necesaria de un instante determinado del juego. Básicamente guarda la posición del agente en cada iteración, de forma que el juego pueda llevar el historial de jugadas.

- `DoublePosition`.- Usada para especificar una posición real (en valores `double`), normalmente para la posición del agente. Dentro de la clase se incluyen algunas utilidades, como la distancia euclídea entre dos objetos del tipo `DoublePosition` o la transformación del objeto actual, con diferentes modalidades matemáticas, a un objeto `Position`.

  ¡Ojo! Para asegurarnos un número de estados de búsqueda finito, e igual para todos, en los agentes de búsqueda deberemos emplear el método estático `getDiscrN`, dentro de `SearchAgent`. Los detalles se verán cuando expliquemos los agentes de tipo búsqueda.

- `Position`.- Muy similar a `DoublePosition` pero con coordenadas **enteras**, ver su significado y uso en la sección 3.2.
- `Movement`.- Nos permite realizar un movimiento del agente, indicando la posición inicial, la distancia a recorrer y el ángulo en radianes. Su método `performMove()` nos dirá cuál sería la hipotética posición final. La clase `Game` lo usa para comprobar la validez de los movimientos. Ojo, vuestros agentes de búsqueda nunca llamarán directamente a este método sino a través de `simulateMovement` en `Game`.
- `AgentPlayer`.- Es la interfaz que tiene que implementar cualquier agente jugador de SoccerGoal. Para los agentes de búsqueda de esta práctica, habrá que implementar una subclase de `SearchAgent` que implementa esta interfaz y presenta algunas particularidades que describimos en la siguiente sección.

Es importante indicar que para la correcta visualización del juego tendréis que tener el fichero de imagen `ballPeq20.png` en el mismo directorio del paquete `game`. En caso contrario la pelota se visualizará en modo GUI como un círculo relleno de color negro.

3.3. Construcción de un agente

Como se ha dicho anteriormente los agentes ⚽ deben implementar la interfaz `AgentPlayer`, cuyo contenido se copia a continuación:

```
package game;

/**
 * An interface for classes that can choose agents's moves
 * @author ssii
 *
 */

public interface AgentPlayer {

  /**
   * Chooses a Movement for the agent player in the Game.
   * @param game
   * @return a Movement
   * @throws NoPathFoundException
   */
  Movement move(Game game) throws NoPathFoundException;

}
```

Es decir, cualquier clase que lo implemente deberá tener un método `move` que reciba el juego actual como entrada, y que devuelva el movimiento elegido como siguiente jugada. Esta es una interfaz genérica destinada a poder emplear cualquier tipo de agente, que puede estar basado en diferentes enfoques de Sistemas Inteligentes. Como en esta práctica vamos a emplear los agentes de búsqueda, hemos creado una clase abstracta `SearchAgent` que presenta elementos comunes, y vuestro agente `MySearchAgent` en esta práctica **tendrá que ser subclase de la misma**.

Estas son las principales características de la clase `SearchAgent` que debéis conocer:

- Necesita un valor para su variable `numAngles` que servirá para determinar cuántos ángulos puede usar el agente de búsqueda en su movimiento. Será el factor de ramificación (b) que veíamos en clase. Es importante que el constructor de vuestra clase llame a `super(n)` con un valor para n, porque sino no funcionará correctamente.

- Entre esos ángulos posibles, que se almacenan en el array `possibleAngles`, nosotros trabajaremos con sus índices. El de índice 0 será siempre el de 0 radianes = 0 grados, que es moverse a la derecha. El siguiente será $\frac{2\pi}{numAngles}$ radianes más en el sentido antihorario, y así hasta el índice `numAngles-1`. La figura 6 explica la correspondencia entre los ángulos y el valor de `numAngles`. Por ejemplo, en 6.(b), el índice 0 serían 0 radianes, el 1 corresponde a $\pi/3 = 60$ grados, el 2 $2\pi/3 = 120$ grados, y así, hasta el último índice (5), que correspondería a $5\pi/3 = 300$ grados.

  Es importante notar que este ángulo se mira con respecto al campo tal cual se ve en la interfaz (ver figura 1.(b)), por lo que moverse hacia arriba va a suponer siempre decrecer la coordenada y, por ejemplo. De esto se encarga la clase `Movement`, es transparente para el agente.

- La superclase `SearchAgent` ya escribe el método `Movement move(Game game)` que necesita implementarse para la interfaz, y devuelve el movimiento correcto al juego. Vuestros agentes **sólo tendrán que devolver el índice** del ángulo (acción posible), tal cual lo hemos explicado en el punto anterior, mediante el método `public abstract int nextMove(Game game)` que nos exige la clase `SearchAgent`.
  [**IMPORTANTE**] Si nuestro algoritmo no encuentra el camino, tendrá que devolver un -1 y ya se encargan otras clases de gestionar la *excepción* correspondiente.

- La superclase `SearchAgent` mantiene tres variables, `numExpandedNodes`, `numExploredNodes` y `numGeneratedNodes`, que guardarán el número de nodos generados, explorados y expandidos (respectivamente). Ella se encarga de comunicárselo al juego, pero vuestras clases se encargarán de pasar ese valor mediante los métodos del tipo `setNumXXXXX(numero)`. Si no lo hacéis correctamente, estas estadísticas tendrán el valor (incorrecto) de -1.

- La superclase `SearchAgent` es la que define de qué longitud u (`unit`) van a ser los "pasos" del agente (en esta práctica: $0 < u \leq 1.0$). En el modo 'práctica' de `Main.main` esto no es configurable, pero se usan varias posibilidades. Esta longitud de paso será la misma durante una partida.

- En `SearchAgent` existe una variable, denominada `discrN`, que en esta práctica será fijada para todos los casos igual (40). Cuando queráis usar la posición del agente como estado, para por ejemplo, comprobar que no está ya en cerrados, el tema se *complica* al estar usando para las posiciones del agente un sistema de coordenadas "real". Por ello, necesitamos una *abstracción* que simplifique el espacio de búsqueda, ya que teorícamente podrían existir infinitas posiciones reales. Por ello, se os proporciona el método estático `DoublePosition getNDiscrStateFor(DoublePosition value)`.

  El número `discrN` significa cuántas divisiones (en ambos ejes) hacemos en un unidad cuadrada, por ejemplo `discrN=2`, tendría 2 divisiones en cada coordenada, y daría cuatro posibles estados en una unidad cuadrada. Lo hemos fijado a 40, que es un número alto, para asegurarnos que interfiere lo menos posible en la solución *esperada*, al darle una granularidad más fina. En realidad, estudiar el valor adecuado de este parámetro en conjunción con el número de ángulos en los que se puede mover un agente, darían para un estudio interesante, que vamos a omitir en esta práctica.

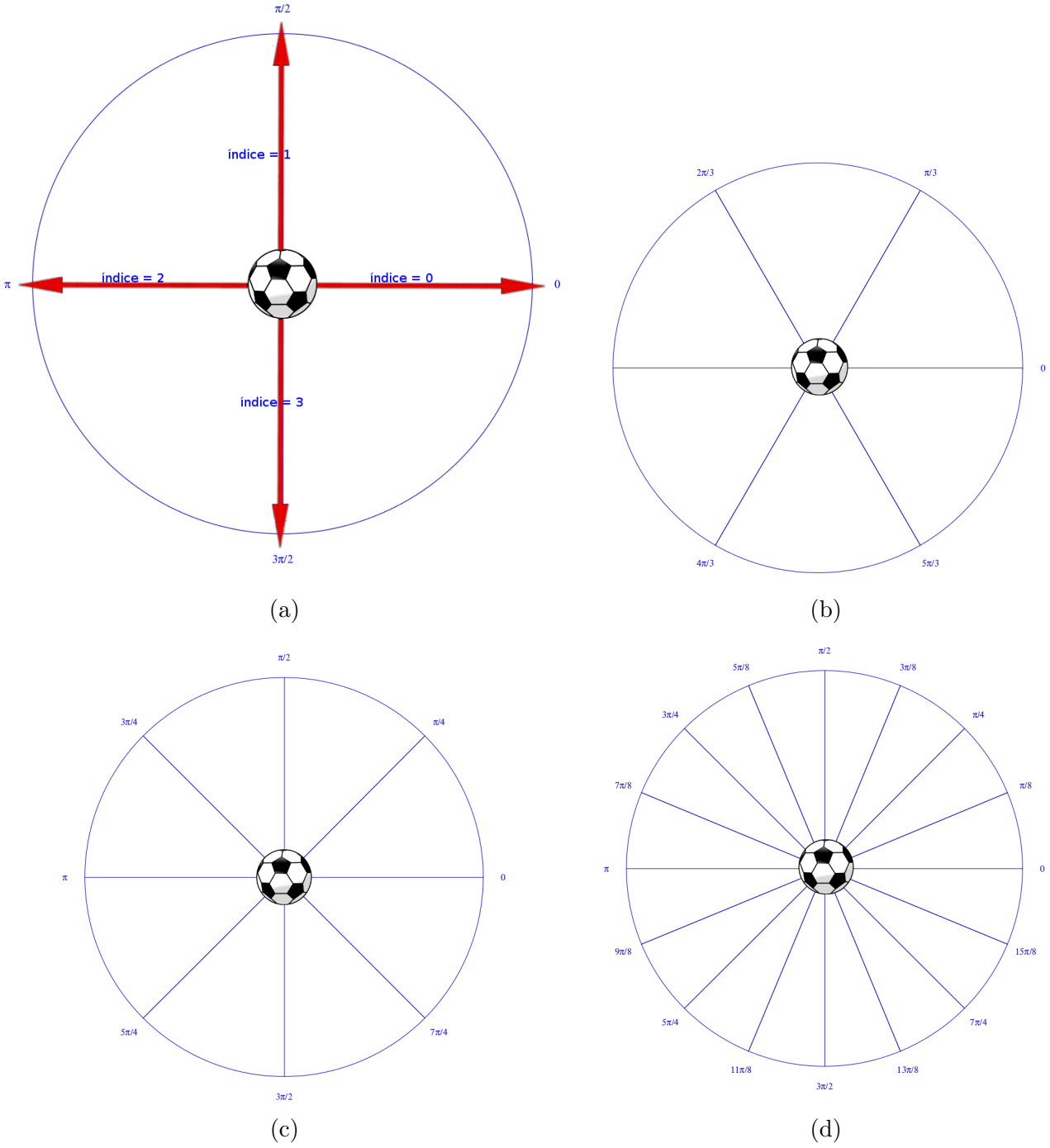
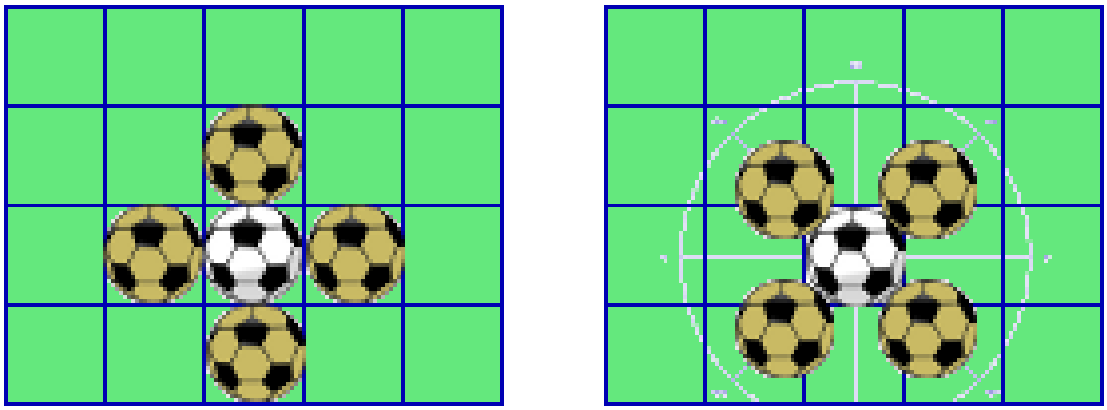


Figura 6: (a) Ejemplo con 4 ángulos `numAngles` = 4 indicando las direcciones con flechas. (b) Ejemplo con `numAngles` = 6; (c) Ejemplo con `numAngles` = 8 ; (d) Ejemplo con `numAngles` = 16.

En cualquier caso, si el agente se limitara a moverse en los cuatro ángulos básicos 1 unidad, no tendríamos ningún problema, pues los estados por los que puede pasar son siempre *enteros*, ver figura 7.(a). Pero la idea es que exploréis con una mayor capacidad de *movilidad*, para comprobar qué sucede cuándo más acciones (sucesores) son posibles.



(a) (b)

Figura 7: La pelota indica la posición inicial antes de un movimiento, y en amarillo las posibles situaciones finales después de un movimiento con un paso de longitud 1 (a) Con cuatro ángulos y pasos de 1.0, siempre va a moverse de una posición a otra. (b) Con 8 ángulos, obviando las 4 direcciones (derecha, arriba, izquierda y abajo) que ya se ven en la figura (a), los movimientos ya nos llevan a posiciones diferentes a cuadrados enteros.

3.4. Ejemplos de agentes jugadores con `SearchAgent`

A modo de ejemplo en el paquete **`search`** os damos dos agentes muy sencillos, que pasamos a comentar en esta sección. De nuevo, incidimos en que si bien son subclases de **`SearchAgent`**, para que veáis una base de cómo crear vuestros agentes, éstos no están desarrollando búsqueda, tal cual lo hemos visto en clase, y tal cual van a tener que comportarse vuestros agentes.

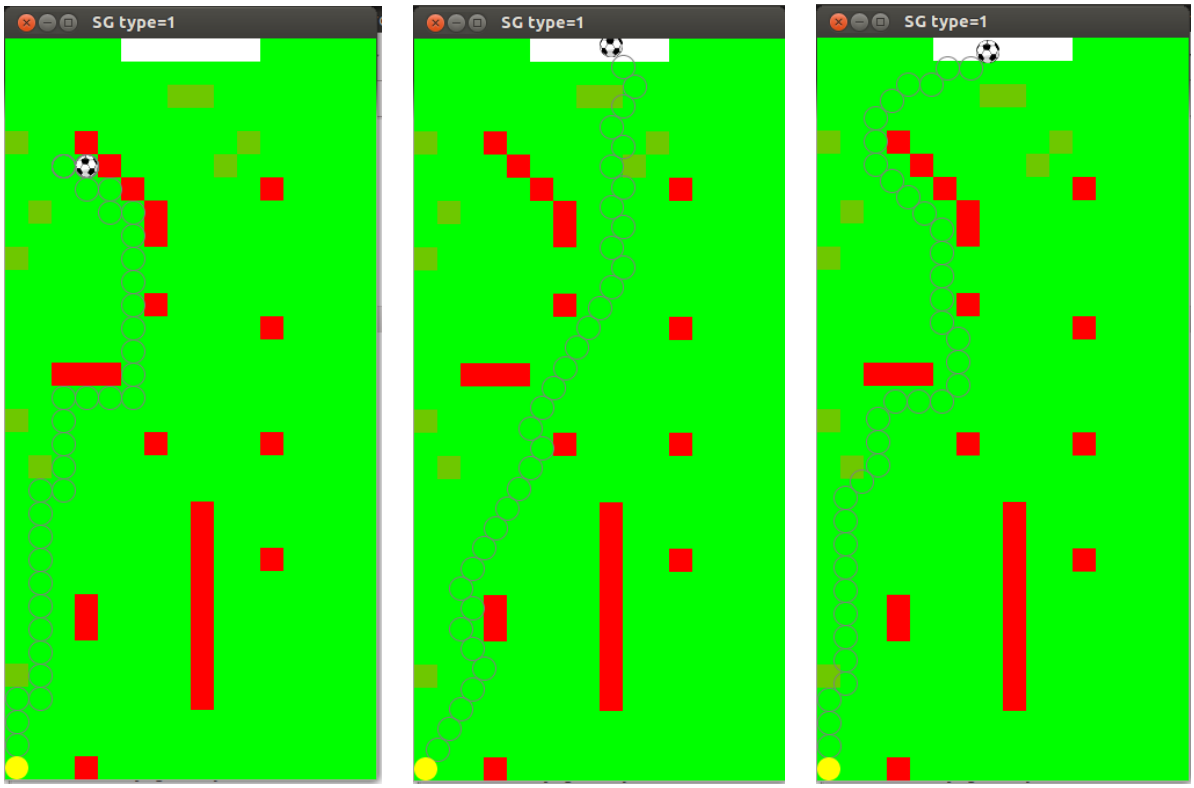
Ejemplo 1: Algoritmo *greedy* o voraz básico Copiamos, a continuación, el código eliminando los comentarios `javadoc`, que ya encontraréis en la carpeta `doc` correspondiente.

```
package search;

import game.*;

public class GreedyAgentNaive extends SearchAgent {

  boolean debug = false;

  public GreedyAgentNaive(int num) {
    super(num);
  }

  public int nextMove(Game game) {
    int i;

    // num will keep the maximum number of possible angles
    int num = possibleAngles.length;

    DoublePosition iPosition;

    double[] distances = new double[num];
```

```
        // loop for any hypothetical possible action
        for (i = 0; i < num; i++) {

            // the game simulates what happens if this agent moves from the current
                agent position using angle of index i
            iPosition =
                game.simulateMovement(game.field.agentPosition,possibleAngles[i]);

             /* if previously simulateMovement returns null means the this movement is
                 not valid, otherwise iPosition will
                contain the DoublePosition that correspond to the final position,
                    given that the movement would have been performed */
            if (iPosition != null) {
                // iPosition is the final position of a valid Movement, so we compute
                    the distance to midGoal
                distances[i] = DoublePosition.euclideanDistance(iPosition,
                    game.field.getMidGoal().toDoublePosition());
                // Include the cost of going through a hole
                if(game.isInHole(iPosition))
                    distances[i]+=game.holeCost;
                if (debug) {
                    System.out.println("Distance of moving in direction-angle i=" + i
                        + " is " + distances[i]);
                }
             } else
               // we put in distance[i] a very large value ('emulating' infinite)
               distances[i] = Double.MAX_VALUE;

        }

        /* Code for computing the index of minimum distance */
        double min = Double.MAX_VALUE;
        int minInd = -1;
        for (i = 0; i < num; i++) {
            if (distances[i] < min) {
              min = distances[i];
              minInd = i;
            }
        }

        if (debug)
            System.out.println("I chose i=" + minInd + "!!!");

        // minInd is now the chosen index
        return minInd;
    }
}
```

Pues bien, este agente, a cada paso del juego, devuelve mediante el método `nextMove` el índice de aquel ángulo (movimiento) que sitúa la pelota más cerca (en términos de coste) de la posición (de tipo "entero") media de la portería. En la figura 8 podéis ver tres ejecuciones en el mismo escenario con `numAngles` 4, 6 y 8. Es evidente que este algoritmo no controla estados repetidos, por lo que

se puede *atascar* en su camino. De hecho, en el caso de la figura 8.(a) efectivamente se atasca y nunca encontrará el camino.



(a) (b) (c)

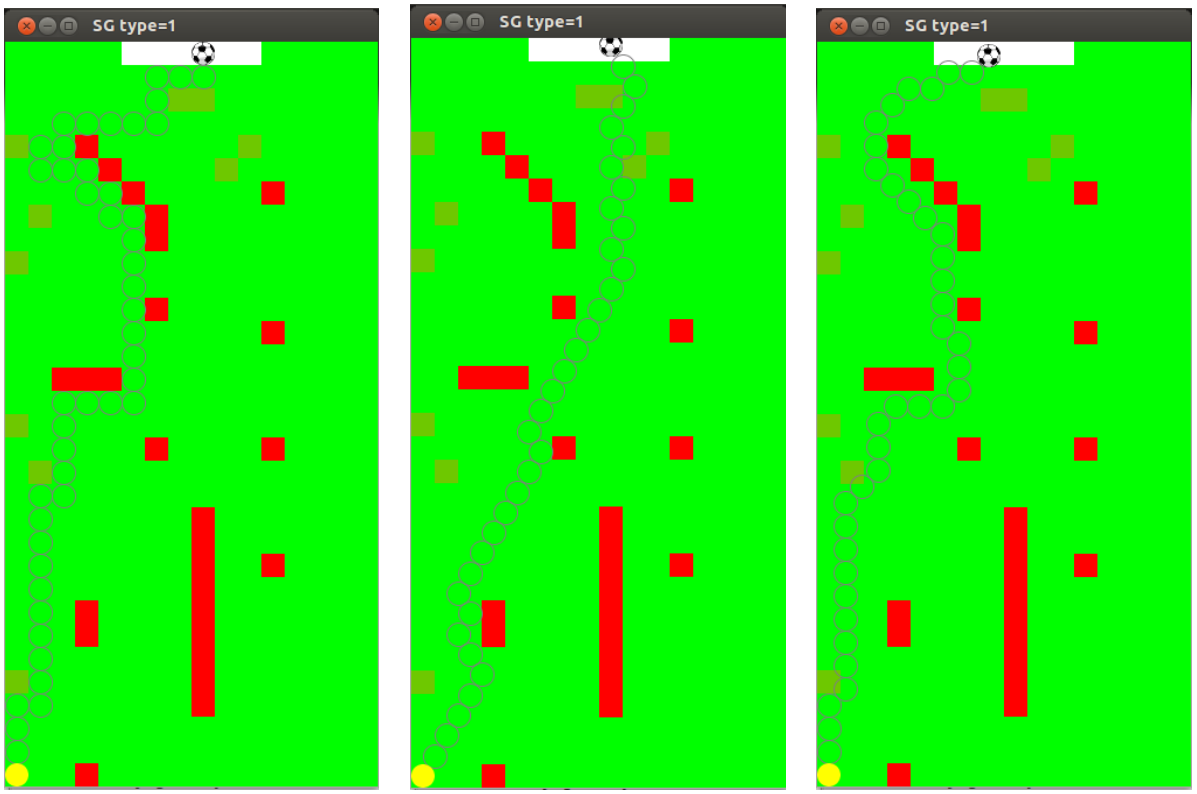
Figura 8: Ejemplos del Greedy básico: (a) 4 ángulos; (b) 6 ángulos; (c) 8 ángulos.

Ejemplo 2: Algoritmo *greedy* o voraz "mejorado" Para solventar este problema, de una manera bastante *rudimentaria*, os dejamos además una segunda versión del mismo agente que hemos denominado `GreedyAgentRd`. La figura 9 muestra los mismos escenarios que en la 8, pero con este nuevo agente modificado. Fijaos que en los casos (b) y (c) ambos agentes se comportan igual, porque no han encontrado estados repetidos, que es lo que los diferencian.

El `Rd` del nombre hace mención a su naturaleza aleatoria (*random*) pues cuando detecta un ciclo, es decir, que ya ha pasado por una posición, en lugar de escoger el movimiento que le acerca más a la portería escoge cualquiera de los movimientos posibles, de forma aleatoria. No entra en ciclos como antes, podría ciclar por otros motivos. En cualquier caso, se comporta *torpemente* y puede que se pierda en explorar exhaustivamente una zona del espacio sin necesidad. Incluso, en campos grandes, puede perder la partida por superar el número máximo de iteraciones. Se trata de que tengáis un ejemplo de programación, no un agente que funcione con un comportamiento excepcional. Y además, este agente, sin ser de búsqueda, ya usa algunos de los elementos que os serán necesarios a vosotros.

No vamos a copiar todo el código como antes (lo tenéis en el fichero `java` correspondiente), sólo comentaremos las partes más significativas:

- Usamos una lista `visited` para poder guardar por qué estados hemos pasado. Para ello empleamos una estructura de tipo `Hash` para que los accesos sean más rápidos.



(a) (b) (c)

Figura 9: Ejemplos del Greedy *random*: (a) 4 ángulos; (b) 6 ángulos; (c) 8 ángulos.

- Cuando un estado ya es visitado distinguimos entre la posición real del agente y el estado que nuestra *abstracción* (discretización) reconoce. El estado real, por ejemplo `DoublePosition dp`, lo necesitamos conocer en cada momento (y en cada nodo de búsqueda en vuestro caso) para conocer dónde estamos exactamente. Y el estado abstracto que se corresponde con él, será el que resulte de la discretización (con una granularidad muy fina, `discrN=40`). Para conseguir ese estado discretizado tendremos que usar lo que nos devuelva `SearchAgent.getNDiscrStateFor(dp)`.

- Necesitamos llevar registro del número de nodos generados, explorados y expandidos. En este caso, al no ser exactamente una búsqueda lo hemos *adaptado* así: generado será cualquier posición a la que llegue con mi algoritmo; expandido será cualquier posición de la que hemos recorrido sus sucesores para encontrar el de distancia mínima; y explorado serán los expandidos, más la posición a la que le preguntamos si es el objetivo. Por eso siempre la diferencia es de 1.

  No es necesario que entendáis esta invención que hemos hecho, sólo que veáis un ejemplo de cómo lo tendréis que implementar vosotros, con dos ideas principales:

  1. Tendréis que actualizar el número de nodos generados, explorados y expandidos conforme se desarrolle vuestra búsqueda.
  2. Tendréis que aseguraros de que antes de que finalice el agente de jugar, éste comunique al juego sus estadísticas. La finalización se puede dar por diversos motivos: porque ha llegado al objetivo, porque no ha encontrado un camino – y devuelve -1 – o porque

supera el número máximo de iteraciones, etc.

Finalmente, también podéis ver un agente muy sencillo: `SearchReadArrayAgent`. Este agente parte de conocer la solución, con lo cual en la práctica no resuelve nada. Os podrá valer con finalidades de depuración, cuando queráis comprobar en el GUI la solución devuelta por algún algoritmo, y no queráis esperar de nuevo a realizar la búsqueda (que en algunos casos será lenta). Por ejemplo, tomemos el output en texto de la figura 4. Y copiamos la solución encontrada (vector de índices) directamente en esta clase:

```
package search;

import game.*;

/**
 * This agent just read the solution from an array. So we must be sure we have the
     correct game and the correct solution, or
 * errors will probably happen.
 * @author ssii
 */
public class ReadArrayAgent extends SearchAgent {

    public int[] myPath =
        {1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,2,1,2,1,2,2,1,0,1,0,0,0,0,1,1,0,0,1}
    //This exact path works ok when playing -type 1 scenary

    int step = -1;

    public ReadArrayAgent(int num) {
        super(num);
    }

    public int nextMove(Game game) {
        step++;
        if (step==0){
          this.setNumExpandedNodes(0.0);
          this.setNumExploredNodes(0.0);
          this.setNumGeneratedNodes(0.0);
        }
        return myPath[step];
    }
}
```

Si ejecutamos

```
java Main -type 1 -agent search.ReadArrayAgent
```

la salida debería ser la misma que en 4 (izda) [3].

[3] como el número de ángulos es el valor por defecto (4), no hace falta

3.5. Funciones a tener en cuenta

Cuando implementéis vuestros algoritmos de búsqueda, tendréis, como vimos en clase, que pensar primero una solución, y ya obtenida, ir diciéndole al juego a cada paso cuál es la acción elegida. Es decir, al principio no se juega, sino que se simula que se juega para conocer por dónde se puede ir. En definitiva, gran parte del trabajo se basa en *imaginar* cómo queda el estado del juego tras uno o varios hipotéticos movimientos, pero sin alterar el estado real del juego. Los siguientes métodos están incluidos en `game.Game` y entre ellos están los que necesitaréis emplear:

- `simulateMovement((DoublePosition startingPos, double angle)`.- Este método recibe la `DoublePosition` de partida, y el ángulo (`possibleAngles[indice]`) con el que pretendemos simular el movimiento, y devolverá `null` si ese movimiento no es válido o un `DoublePosition` con la posición final, en otro caso.

NOTA: Existe también el mismo método pasándole directamente el movimiento: `DoublePosition simulateMovement(DoublePosition startingPos, Movement move)` con el mismo comportamiento. Podéis usar cualquiera de los dos, según el que os parezca más intuitivo o cómodo. En `GreedyAgentNaive` se emplea el primero, y en `GreedyAgentRd` el segundo.

- `isInGoal(DoublePosition dp)`.- Nos dice si la posición indicada se encuentra ya en la portería, y en situación por tanto de finalizar el juego con éxito.
- `isInHole(DoublePosition dp).- Le pasamos la posición del agente (dp), tal cual la gestiona el juego, y nos dice si está o no en un hoyo.`
- `getAgentCenterOf(DoublePosition dp)`.- Nos daría, si es que lo queremos usar para algo, la posición exacta donde se encuentra el centro de la pelota.
- `setHoleCost(int c)`.- Aunque en las ejecuciones en modo 'practica' no se va a cambiar el valor de la penalización, en principio, consideramos que puede serútil jugar con el valor de este coste para depurar vuestro código y comprobar el comportamiento de los agentes. Mediante este método podéis cambiar la penalización por pasar por un hoyo, sin necesitad de cambiar el código del atributo correspondiente (`holeCost`) en la clase `Game`.

4. Trabajo a realizar por el alumno

Lo primero que tendréis que hacer es formular la tarea a resolver por el agente como un problema de búsqueda (definición de estado, estado inicial, acciones disponibles – función sucesor, test objetivo, coste de un camino, etc.).

Una vez hecho ésto tendréis que resolver, con al menos la técnica de búsqueda básica que tiene en cuenta el coste (**búsqueda de coste uniforme**), un ejemplo "a mano". Este ejemplo será el que resulta al ejecutar `java Main -type 5 -agent search.desarrollo.XXXX` (ver figura 10).

Este pequeño ejercicio lo tendréis que incluir en la memoria, pero lo interesante es que lo hagáis antes de empezar a programar, para entender bien el problema. Para el orden de generación de nodos, debéis seguir la convención indicada en el siguiente apartado (4.1).

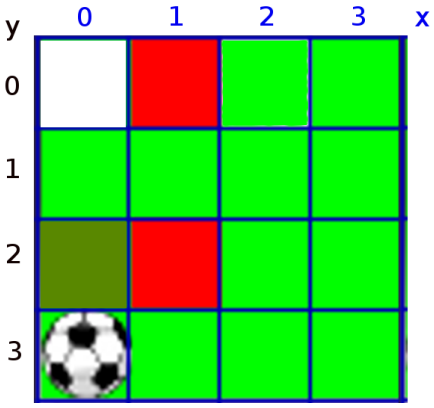


Figura 10: Ejemplo para hacer el árbol de búsqueda a mano

4.1. Agentes a implementar

En la parte obligatoria de la práctica tendréis que implementar cinco agentes de búsqueda **en grafos**:

- Un agente ⚽ basado en **búsqueda en anchura**.
- Un agente ⚽ basado en **búsqueda por profundidad**.
- Un agente ⚽ basado en **búsqueda de coste uniforme**.
- Un agente ⚽ basado en **búsqueda primero mejor** (búsqueda informada).
- Un agente ⚽ basado en el **algoritmo** A^* (búsqueda informada).

Sin embargo, en realidad estos cinco agentes pueden ser implementados como simples variaciones de un agente de búsqueda genérico que responde al siguiente esquema muy simplificado e incompleto:

```
// Inicializacion

nodo = CrearNodo(estado=EstadoInicial, f=-1, ....)
// notese que el valor f() para el nodo raiz no influye

Abiertos = {nodo}    // lista ordenada por valor f(). De menor a mayor.
Cerrados = {}

// Bucle principal - busqueda en grafos

Mientras Abiertos!=vacio y no-solucion hacer

    nodoActual = sacarPrimero(Abiertos)

    Si esFinal(nodoActual) Entonces devolver la solucion y finalizar

    Si NO existe nodo en Cerrados t.q. (nodo.estado == nodoActual.estado) Entonces
        hijos = generarSucesores(nodoActual.estado) //estados sucesores
```

```
        Para i=1 hasta |hijos| hacer
            valor = calcularFSegunEstrategia(nodoActual, hijos[i], ...)
            nodo = CrearNodo(estado=hijo[i], f=valor, ...)
            InsertarSegunF(nodo,Abiertos)
        FinPara

        Cerrados = Union(Cerrados, nodoActual)

    FinSi

FinMientras
```

Para instanciar el procedimiento anterior a las distintas estrategias, basta con implementar la función calcularFSegunEstrategia(), por ejemplo, sabemos que en el algoritmo A-estrella ese valor corresponde a la suma del coste real de alcanzar el estado actual (hijos[i]) y el valor de la heurística asociada a dicho estado: $f(n) = g(n) + h(n)$.

Bien, tenéis que implementar una clase abstracta MySeachAgent.java que extiende a la clase SearchAgent proporcionada[4].

En esta clase que implementa el método genérico, es **muy importante seguir las siguientes convenciones**:

- Los sucesores se generarán de menor a mayor índice dentro del vector `possibleAngles`
- En el orden mantenido en la lista de Abiertos (o frontera), a igual valor de la función de evaluación tendrán preferencia (aparecen antes y, por tanto, se extraerán antes) los nodos de mayor antigüedad.

La clase abstracta MySearchAgentcontendrá el método calcularFSegunEstrategia(.....) que deberá ser sobreescrito por todos los cinco agentes de búsqueda solicitados. De hecho, el código de estos cinco agentes debería contener poco más que dicho método. Los agentes deben ser nombrados:

1. `BusquedaAnchura.java`
2. `BusquedaCosteUniforme.java`
3. `BusquedaProfundidad.java`
4. `BusquedaPrimeroMejor.java`
5. `BusquedaAEstrella.java`

Estos cinco ficheros junto con vuestra clase genérica MySearchAgent.java y cualquier otra clase que tengáis que implementar, deberá estar contenida en un subdirectorio de search con nombre desarrollo, de forma que no debéis tocar ningún otro código. Por ejemplo para llamar al agente de búsqueda en anchura la opción será:

-agent search.desarrollo.BusquedaAnchura

¡Ojo!: para superar la práctica todos los agentes presentados **deben funcionar correctamente**, y encargarse de devolver al juego la información requerida (estadísticas sobre nodos y el índice -1, en caso de no encontrar un camino solución).

[4]Recordad que si un agente hereda de la clase `SearchAgent`, debe permitir como parámetro en su constructor un número n, que será el número de ángulos empleados en los movimientos.

4.2. Experimentación

4.2.1. Parte obligatoria

Una vez implementados vuestros agentes, y funcionando correctamente, se os pide que hagáis una serie de experimentaciones para valorar su comportamiento. Para ello tendréis que realizar las siguientes tareas:

1. Realizar una comparativa entre los distintos tipos de búsqueda.
2. Comparativa en los agentes sobre el uso de distintos números de ángulos, al menos con `numAngles=4` y otro valor de `numAngles`> 4. Tendréis que analizar el comportamiento de los agentes con diferentes tipos de movilidad.
3. Fijado el número de ángulos (a cada uno de sus valores escogidos), tendréis que realizar un estudio del comportamiento de los algoritmos básicos (anchura, coste-uniforme, profundidad, primero-mejor y A^*) entre ellos. Esto incluye estudiar si los resultados en tiempo, nodos generados/explorados/expandidos, calidad de la solución encontrada, etc. se corresponden con el análisis de los algoritmos vistos en clase. Vuestros resultados y razonamientos en este aportado serán muy importantes en la valoración de la práctica.

   Importante: Recordad que con el parámetro `-seed` podéis usar diferentes semillas y vuestros estudios se pueden beneficiar de realizar varias ejecuciones diferentes.

4.2.2. Parte opcional

Opcionalmente, se propone extender vuestra experimentación con la implementación de un sexto agente basado en búsqueda con profundidad iterativa (`BusquedaProfIterativa`) e incluirlo en el estudio y comparación.

5. Presentación de la práctica

Todos los alumnos tendrán que presentar un único archivo comprimido **¡en formato zip!**[5] que contendrá:

- La memoria de prácticas **¡en formato pdf!**.
- El código fuente de `MySearchAgent.java` y de los 5 (ó 6) agentes implementados, que estarán listos para ubicarse en la carpeta `search`, y que se ejecutarán sin errores con el resto del código proporcionado. Opcionalmente, si habéis empleado clases auxiliares, su código fuente también se entregará y también se ubicaría en la carpeta `search.desarrollo`.
- Un archivo de texto `readme.txt` donde describáis brevemente el contenido del .zip y alguna particularidad que consideréis necesaria. Se trata de algo muy esquemático, los detalles vendrán en la memoria.

[5] Este archivo comprimido se enviará a través de la plataforma moodle, dentro de la tarea correspondiente a la práctica uno.

Con respecto a la MEMORIA, **es importante que sigáis estas pautas**:

- Tendréis que formular el problema correctamente, como se indica al principio de la sección 4, y hacer el ejemplo a mano propuesto.
- En cuanto a la implementación de cada agente o algoritmo, tendréis que indicar cómo se ha abordado el problema y cuáles son las soluciones aportadas. Estas soluciones se deben describir con suficiente nivel de detalle, incluyendo pseudocódigo cuando sea necesario. Por otra parte, cualquier argumento sobre el buen/mal funcionamiento de los agentes desarrollados, así como una comparación entre ellos debería ser incluido en la memoria, nos referimos a estadísticas tomadas sobre una serie de ejecuciones que indiquen el comportamiento *promedio* de los agentes en función del tipo de búsqueda empleado.
- Puesto que los escenarios (campos de fútbol, hoyos y obstáculos) son aleatorios, tendríais que realizar una experimentación de varias ejecuciones para poder sacar conclusiones claras, así como calcular los promedios de los valores importantes a tener en cuenta.
- En cuanto al formato, la memoria debería cumplir los estándares esperados para un documento científico. Si queréis profundizar en ésto os recomendamos consultar el documento sobre rúbricas de la escuela, que hemos dejado en el espacio *Moodle* de la asignatura.

Tened en cuenta que la **calidad de la memoria** de la práctica (en contenido y forma) será muy importante a la hora de evaluar la nota de prácticas, como está indicado en la guia-e de la asignatura y como se os explicó en la presentación de la asignatura.

Las prácticas deben realizarse en grupos de dos, si bien pueden presentarse también de forma individual (lo que implicará un sobreesfuerzo). Los profesores van a citar a los alumnos para que defiendan mediante entrevista personal el trabajo entregado. En dicha entrevista cada alumno mostrará en el laboratorio a los profesores el funcionamiento de la práctica en el ordenador y se contestará a las preguntas (individuales) que se estimen oportunas a fin de conocer cómo se ha llevado a cabo la realización de la práctica.

Además, **se fijará una fecha límite para enviar la práctica** (lo mismo sucederá con cada una de las siguientes). Para esta práctica la fecha de entrega es: **viernes, 7 de noviembre de 2014** a las 23:55. Todas las entregas se harán via Moodle: http://campusvirtual.uclm.es [6]

[6] En caso de que alguien envíe la práctica fuera de plazo la calificación obtenida será multiplicada por un **factor de penalización de 0.8**