UNIVERSIDAD DE GRANADA

/ UGR / decsai

Departamento de Ciencias de la Computación e Inteligencia Artificial

Técnicas de los Sistemas Inteligentes

Curso 2018-19

Práctica 1

Tutorial 2: API/wrapper para el juego Boulder Dash.

1. La API GVG-AI

GVG-AI (www.gvgai.net) es una API común que permite crear agentes inteligentes para una gran cantidad de juegos diferentes. De todos los juegos que trae, en la Práctica solo se va a usar el número 11. Debido a que la API es la misma para todos los juegos, es muy abstracta y relativamente difícil de usar. Por esa razón, se ha creado otra API que funciona como "wrapper" de GVG-AI y facilita la tarea de construir un agente para este juego solamente. Esta API está formada por varias clases y enumerados, todos dentro del paquete practica_busqueda que se proporciona en el material de la práctica.

La clase más importante es **BaseAgent**, de la que debe heredar la clase que se use para implementar el agente. Esta clase nos permite obtener información de forma sencilla sobre el juego. Implementa métodos que, pasándoles como parámetro el estado del juego, devuelven los enemigos, paredes, casillas vacías, piedras, la salida, el jugador y el número de gemas recogidas o las que faltan por recoger para poder abandonar el nivel.

Para ver ejemplos de uso de estos métodos (y otros) se puede usar el agente **InformationAgent,** que los utiliza para obtener información sobre el entorno y la imprime por pantalla. También hay otro agente de ejemplo, **DemoAgent**, que usa estos métodos para escoger en cada turno una acción a realizar. Los métodos del tipo get***List devuelven una lista con los elementos correspondientes, sin ningún orden en concreto. Si se quieren obtener los elementos ordenados según la casilla en la que se encuentren (por ejemplo obtener los enemigos que hay en una casilla determinada) se puede usar el método getObservationGrid, que devuelve todos los elementos del mapa en un momento dado en forma de matriz de forma que podemos ver aquellos elementos que se encuentran en una casilla (x,y) determinada.
Estos elementos del mapa (enemigos, paredes, rocas, gemas, casillas vacías, etc.) se denominan "observaciones" y cada una se representa como una instancia diferente de la clase **Observation.** Cada observación tiene un tipo que la diferencia del resto (por ejemplo gema o roca). Todos los tipos se definen en el enumerado **ObservationType**. Aparte, cada observación tiene una posición (x, y) que define la casilla que ocupa en el mapa (el eje Y aumenta hacia abajo). Aparte, la clase Observation tiene métodos para ver si dos

observaciones "chocan" (ocupan la misma casilla) y calcular la distancia euclídea o Manhattan entre las casillas que ocupan. Se pueden consultar ejemplos de estos métodos en la clase InformationAgent.
El jugador es una observación especial por lo que se representa mediante una instancia de la clase **PlayerObservation**, que hereda de Observation. Aparte de la información que guarda la clase Observation, esta clase contiene la orientación del jugador (Norte, Sur, Este u Oeste), definida en el enumerado **Orientation.** También añade el método hasDied para comprobar si el jugador ha muerto (por un enemigo o roca) en un estado del juego. En la clase DemoAgent se puede ver un ejemplo de uso.

2. Tutorial Creación Agente

Ahora explicaremos los pasos iniciales que hay que seguir para crear un agente para este juego:

1. En el paquete practica_busqueda, creamos una clase (da igual el nombre) que herede de BaseAgent. En esta clase implementaremos nuestro agente.

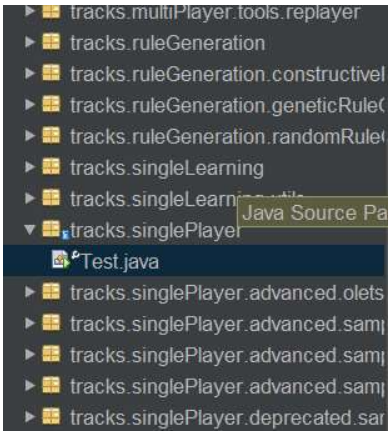
```
package practica_busqueda;

public class TestAgent extends BaseAgent{

}
```

2. A esta clase le añadimos dos métodos: un **constructor**, que tiene que llamar al de la clase padre, y un método **act**. Ambos métodos reciben dos objetos como parámetros: una instancia de la clase **StateObservation** y otro de la clase **ElapsedCpuTimer.** La clase StateObservation encapsula el estado del juego en un momento dado y es necesario pasársela a muchos métodos de la API como parámetro (por ejemplo a los métodos de la clase BaseAgent para obtener información del juego). La clase ElapsedCpuTimer es un temporizador que nos sirve para saber el tiempo que ha pasado desde el comienzo del turno y si nos hemos excedido del tiempo máximo para actuar (más abajo aparece un ejemplo de uso).
   El método act se llama al comienzo de cada turno del agente y es responsable de devolver la acción a realizar en ese turno. Las acciones son del tipo ontology.Types.ACTIONS y pueden ser: ACTION_NIL (no hacer nada), ACTION_UP, ACTION_LEFT, ACTION_DOWN, ACTION_RIGHT (orientarse o moverse en esa dirección) o ACTION_USE (excavar la casilla de delante, si es posible). En el ejemplo, nuestro agente no hace nada.

```java
package practica_busqueda;

import core.game.StateObservation;
import ontology.Types;
import tools.ElapsedCpuTimer;

public class TestAgent extends BaseAgent{

    public TestAgent(StateObservation so, ElapsedCpuTimer elapsedTimer){
        super(so, elapsedTimer);
    }

    @Override
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){

        return Types.ACTIONS.ACTION_NIL;
    }
}
```

Para poder ejecutar nuestro agente debemos irnos a la clase Test, del paquete tracks.singlePlayer.

Una vez en esa clase, debemos cambiar el String **miControlador** a la ruta



(incluyendo el paquete) de nuestra clase (practica_busqueda.*nombre_clase*). En nuestro ejemplo, la clase se llama TestAgent, por lo que quedaría "practica_busqueda.TestAgent".

```
// Available tracks:
String sampleRandomController = "tracks.singlePlayer.simple.sampleRandom.Agent";
String doNothingController = "tracks.singlePlayer.simple.doNothing.Agent";
String sampleOneStepController = "tracks.singlePlayer.simple.sampleonesteplookahead.Agent";
String sampleFlatMCTSController = "tracks.singlePlayer.simple.greedyTreeSearch.Agent";

String sampleMCTSController = "tracks.singlePlayer.advanced.sampleMCTS.Agent";
String sampleRSController = "tracks.singlePlayer.advanced.sampleRS.Agent";
String sampleRHEAController = "tracks.singlePlayer.advanced.sampleRHEA.Agent";
String sampleOLETSController = "tracks.singlePlayer.advanced.olets.Agent";

// Mi controlador
String miControlador = "practica_busqueda.TestAgent";
```

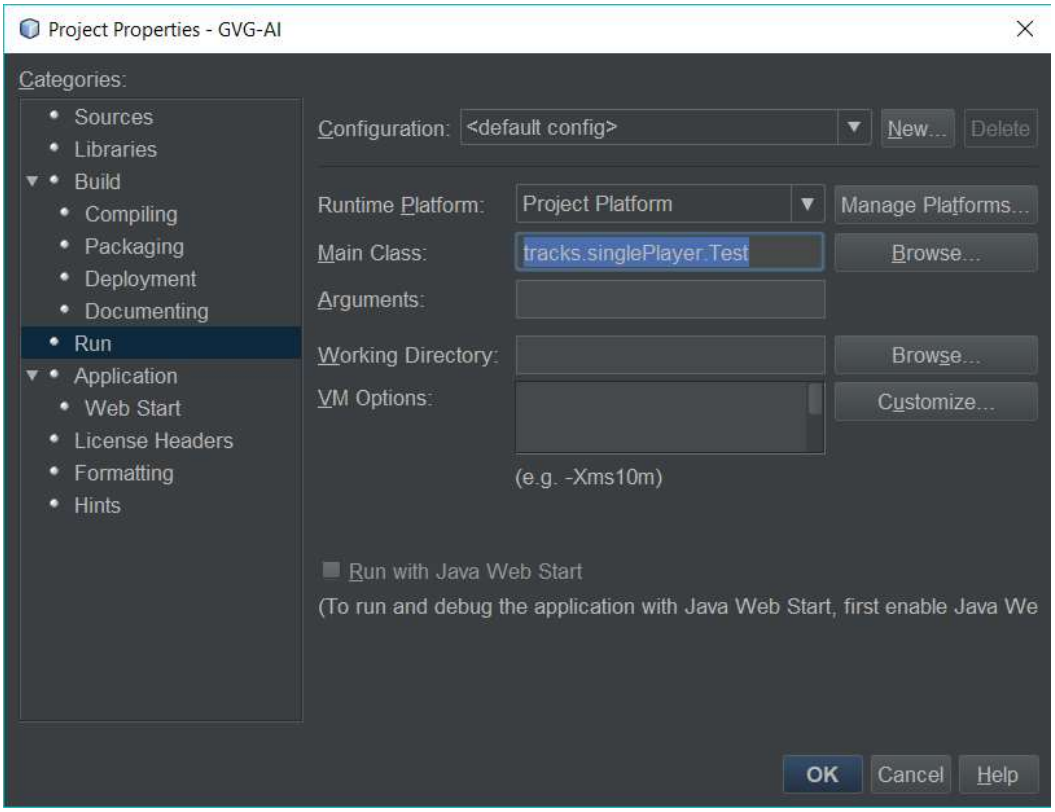
Aparte, hay varios controladores de prueba que vienen con la API y que no solo valen para este juego, por lo que no usan la API simplificada del paquete practica_busqueda. Si se quieren probar, solo hace falta cambiar la variable miControlador por uno de ellos (por ejemplo, sampleMCTSController, que implementa una búsqueda en árboles de MonteCarlo).

```
// 2. This plays a game in a level by the controller.
if (!juego_yo)
    ArcadeMachine.runOneGame(game, level1, visuals, miControlador, recordActionsFile, seed, 0);
```

La variable **juego_yo** controla quién es el jugador: si vale true, jugamos nosotros usando el teclado, si vale false, el jugador es controlado por nuestro agent. Por tanto, para probar nuestro agente, la ponemos a **false**.

```
boolean juego_yo = false; // Decide si juega el agente o tú
```

3. Por último, para empezar el juego solo hace falta ejecutar el proyecto (la clase tracks.singlePlayer.Test debe ser la clase principal).



Al ejecutarse el proyecto, deberíamos tener un agente que no hace nada (ya que su método act devuelve ACTION_NIL). Se puede probar a cambiar la acción o a elegir una al azar también para probar el agente.

4. También puede ser interesante controlar en el método act el tiempo que nos queda para devolver una acción. Para hacer esto usamos el temporizador ElapsedCpuTimer que se pasa como parámetro a act (y al constructor).
   El método **elapsedMillis()** nos devuelve los milisegundos que han transcurrido desde el comienzo del turno. El método **remainingTimeMillis()** funciona igual pero nos devuelve los milisegundos que nos quedan, en vez de los que han transcurrido.

```
@Override
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){

    System.out.println("Al comienzo del turno: " + elapsedTimer.elapsedMillis());

    try{
        Thread.sleep(10);
    }
    catch (InterruptedException e){}

    System.out.println("Tras espera: " + elapsedTimer.elapsedMillis() + "\n\n");

    return Types.ACTIONS.ACTION_NIL;
}
```

```
Executing tracks.singlePlayer.Te
 * WARNING: Time limitations bas
Al comienzo del turno: 1
Tras espera: 10


Al comienzo del turno: 0
Tras espera: 10


Al comienzo del turno: 0
Tras espera: 9


Al comienzo del turno: 0
Tras espera: 10
```

Otro método es **exceededMaxTime(),** que nos devuelve si nos hemos pasado del tiempo límite para devolver una acción.

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer){

    System.out.println("Al comienzo del turno: " + elapsedTimer.exceededMaxTime());

    try{
        Thread.sleep(100);
    }
    catch (InterruptedException e){}

    System.out.println("Tras espera: " + elapsedTimer.exceededMaxTime() + "\n\n");

    return Types.ACTIONS.ACTION_NIL;
}
```

```
run:
Executing tracks.singlePlayer.Test.java ...
 * WARNING: Time limitations based on WALL TIME on Windows *
Al comienzo del turno: false
Tras espera: true


Too long: 0(exceeding 60ms): controller disqualified.
Result (1->win; 0->lose): Player0:-100, Player0-Score:-1000.0, timesteps:0
```