




21 DE JULIO DE 2020

ROBOTUA

ENTORNO DE APLICACIÓN DE ALGORITMOS DE APRENDIZAJE POR
REFUERZO.

VÍCTOR NAVRRO MARTÍNEZ-REINA

UNIVERSIDAD DE ALICANTE, ESCUELA POLITECNICA SUPERIOR, DPTO. DE LENGUAJES Y SISTEMAS
INFORMATICOS. GRUPO DE RECONOCIMIENTO DE FORMAS E INTELIGENCIA ARTIFICIAR.



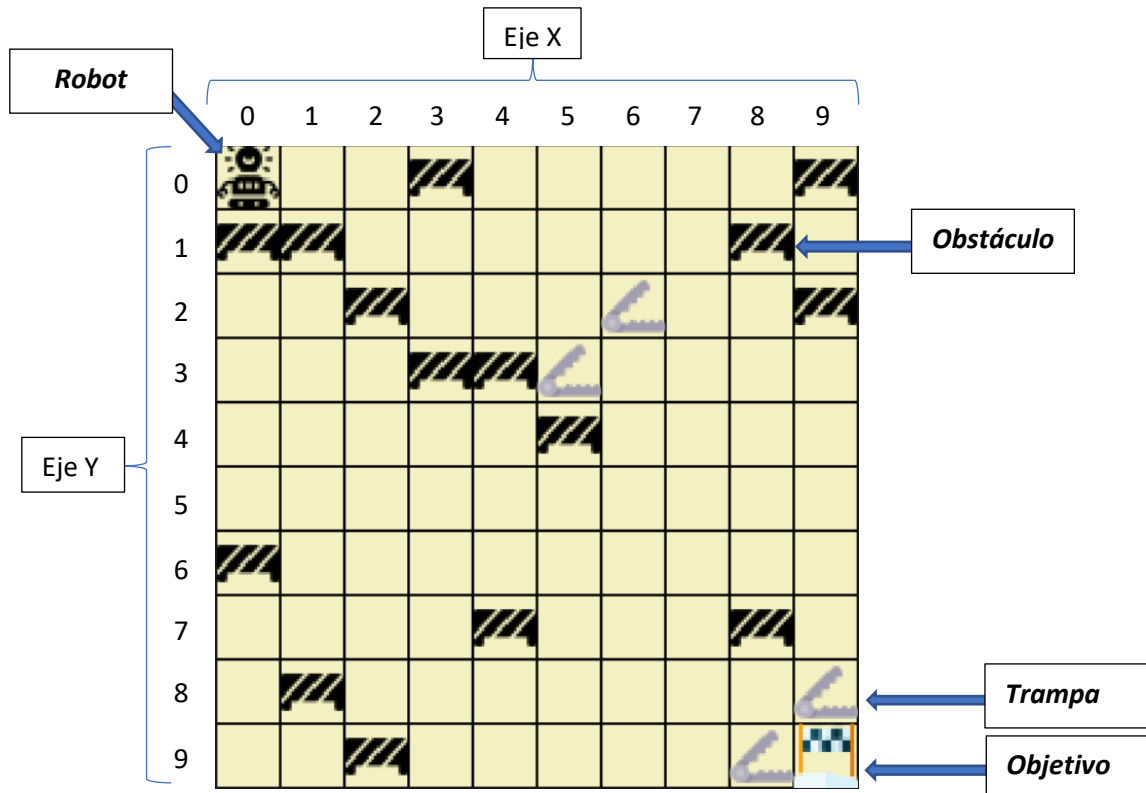
INDICE

| | | |
|------|--|----|
| 1. | Definición general. | 2 |
| 2. | Definición del sistema | 3 |
| 2.1. | Constructor | 3 |
| 2.2. | Step | 3 |
| 2.3. | Reset | 4 |
| 2.4. | Render..... | 4 |
| 2.5. | Action_space.sample | 4 |
| 2.6. | Contadores..... | 4 |
| 3. | Configuración del entorno..... | 5 |
| 3.1. | Descripción de parámetros de configuración. | 5 |
| 3.2. | Ejecución por defecto. | 6 |
| 3.3. | Ficheros JSON | 7 |
| 3.4. | Ejemplos de ejecución con configuración de ficheros JSON | 7 |
| 4. | Ejemplo de utilización del entorno | 10 |
| 5. | Instalación del entorno..... | 12 |
| 6. | Bibliografía..... | 13 |

1. Definición general.

RobotUA es un entorno para la aplicación de algoritmos de aprendizaje por refuerzo. Este entorno ha sido elaborado en Python utilizando la librería Gym de OpenAI.

El objetivo de este entorno es hacer llegar un robot a un punto objetivo en un circuito que contiene obstáculos y trampas. El robot no podrá salirse del circuito ni atravesar los obstáculos. Además, si pisa una trampa se bloqueará el siguiente movimiento. El robot únicamente podrá realizar los siguientes movimientos: izquierda, arriba, derecha y abajo.



- **Robot:** El robot empieza en la posición (0,0).
- **Obstáculo:** Los obstáculos pueden estar repartidos por todo el circuito y puede haber un numero variable de ellos, dependiendo de la configuración. El robot no puede moverse encima de ellos. El obstáculo señalado en el circuito está en la posición (8,1).
- **Trampa:** Las trampas pueden estar repartidas por todo el circuito y puede haber un numero variable de ellas, dependiendo de la configuración. Cuando el robot pisa una trampa el siguiente movimiento queda bloqueado. La trampa señalada en el circuito está en la posición (9,8).
- **Objetivo:** El robot debe llegar al objetivo para resolver el circuito. Este objetivo puede cambiar de posición, dependiendo de la configuración. El objetivo señalado en el circuito se encuentra en la posición (9,9)

2. Definición del sistema

Al utilizar la librería Gym, nuestro sistema tiene una serie de métodos de los que hará uso el código cliente para construir el circuito, realizar movimientos del robot, representar la imagen del circuito y restablecerlo al estado inicial.

El sistema está compuesto por dos clases fundamentales: **RobotUAE** y **Circuito2D**. Ambas clases se encuentran dentro de la carpeta del proyecto en: `gym_robotUA/envs/robotUA_env.py` y `gym_robotUA/envs/circuito_2d.py`, respectivamente.

La clase RobotUAE (extiende de `gym.Env`) representa el entorno en general. Es la clase que se encarga de construir el circuito bajo la configuración deseada (predeterminada o mediante un fichero JSON). También, se encarga de decidir cuándo se podrá mover el robot, además de contener y devolver información útil para el código cliente.

La clase Circuito2D es la representante del circuito con todos sus componentes. Utiliza la librería Pygame para representar el circuito y poder visualizarlo en caso de que sea necesario. También, cuando se desea realizar un movimiento se encarga de comprobar que se realice dentro de la matriz y no haya obstáculos.

A continuación, se explicarán los métodos de la clase más relevante (**RobotUAE**):

2.1. Constructor

```
def __init__(self, config=""):
```

El **constructor** puede recibir un parámetro. El parámetro `config` se espera que contenga una cadena con la dirección absoluta dentro de la carpeta del proyecto que contenga un archivo JSON.

- Si el parámetro `config` está vacío el circuito se construirá con unos valores predeterminados.
- Si el parámetro `config` no está vacío se tomarán los parámetros del archivo JSON para construir el circuito.

2.2. Step

```
def step(self, action):
```

La función **step** es la encargada de realizar el movimiento del robot siempre y cuando las condiciones lo permitan. Devuelve 4 variables:

- Estado: representa la posición lógica dentro de la matriz del robot.
- Reward: es la recompensa asociada al realizar el movimiento (puede ser 1 si ha llegado al objetivo o -1 si no ha llegado, el movimiento está bloqueado o se ha alcanzado el número máximo de movimientos).
- Done: representa si la resolución del circuito a finalizado (True o False).
- Info: un diccionario que contiene información adicional (en este entorno no es necesario devolver ninguna información adicional).

El parámetro `action` hace referencia al movimiento que se desea realizar. Se espera que el movimiento sea un número entero:

- 0: movimiento izquierda.
- 1: movimiento arriba.

- 2: movimiento derecha.
- 3: movimiento abajo.

Las condiciones para poder realizar el movimiento son las siguientes:

- Que el número de movimientos hasta el momento sea menor que el número máximo de movimientos.
- Que el movimiento no esté bloqueado a causa de haber pisado una trampa en el movimiento anterior.

Si se cumplen las condiciones anteriores, el movimiento se tomará como realizado y el contador de movimientos aumentará (aunque haya sido un movimiento hacia el exterior de la matriz o hacia un obstáculo).

Una vez realizado el movimiento, se comprueba:

- Si se ha pisado una trampa (si es así se bloquea el siguiente movimiento) .
- Si ha llegado al objetivo.

2.3. Reset

```
def reset(self):
```

La función **reset** se utiliza para restablecer el entorno al estado inicial. El robot vuelve a su posición (0,0). Devuelve 1 variable:

- Estado: representa la posición lógica dentro de la matriz del robot.

2.4. Render

```
def render(self):
```

La función **render** se utiliza para visualizar el estado del circuito (siempre y cuando se desee visualizar). No devuelve nada.

2.5. Action_space.sample

Esta función nos provee una acción dado un espacio de acciones. El espacio de acciones está definido como un espacio donde se pueden realizar 4 acciones (espacio discreto). Esta función devuelve un numero aleatorio entre 0 y 3 ambos incluidos, que son los que espera recibir la función **step**.

2.6. Contadores

A continuación, se listan los diferentes contadores de la clase:

- episodios_completados: Número de veces que se ha resuelto el circuito.
- num_mov: Representa el número de movimientos que se han realizado desde el inicio del episodio.
- num_mov_totales: Es el número total de movimientos realizados en todos los episodios.
- trampas_pisadas: Cantidad de trampas que se ha pisado en un episodio.
- trampas_totales: Cantidad de trampas que se ha pisado en todos los episodios.

3. Configuración del entorno.

Como se ha nombrado antes, el entorno puede tener una configuración predeterminada o ajustarse a unos parámetros pasados en un fichero JSON.

3.1. Descripción de parámetros de configuración.

A continuación, se listan los diferentes parámetros y para que se usa cada uno:

- **MAX_MOVIMIENTOS:** Es un número entero que representa el máximo de movimientos que se pueden realizar para la resolución del circuito.
- **visualizar:** Es un booleano que se utiliza para activar la visualización del circuito y los movimientos del robot. Si esta opción esta desactivada el programa tendrá unos tiempos de ejecución mucho menores.
- **tam_matrix:** Es una tupla de valores enteros (x,y) que representan el tamaño de la matriz en su eje x e y.
- **delay:** Es un número entero que representa la cantidad de milisegundos entre cada movimiento del robot (este parámetro solo tiene sentido con la opción de visualización activada).

- **tam_casilla:** Es un número entero que representa el tamaño (en pixeles) de las celdas de la matriz (este parámetro solo tiene sentido con la opción de visualización activada).
- **objetivos:** Es un array con una o dos tuplas (x,y) con las coordenadas de los objetivos (donde deberá llegar el robot para completar el circuito).

Si tiene dos tuplas, la primera de ellas es el objetivo final, y la segunda es el punto intermedio por el que deberá pasar el robot antes de ir al objetivo final.

- **trampas:** Es un array de tuplas (x,y) con las coordenadas de las trampas.
- **obstaculos:** Es un array de tuplas (x,y) con las coordenadas de los obstáculos.
- **incluir_delay:** Es un número entero. Tras la resolución del circuito en este número de veces el delay entre movimientos se verá modificado (este parámetro solo tiene sentido con la opción de visualización activada).
- **delay_incluido:** Es un número entero. Cuando se resuelva el circuito tantas veces como marque la variable **incluir_delay**, se modificará el delay entre movimientos al valor que contenga esta variable (este parámetro solo tiene sentido con la opción de visualización activada).

3.2. Ejecución por defecto.

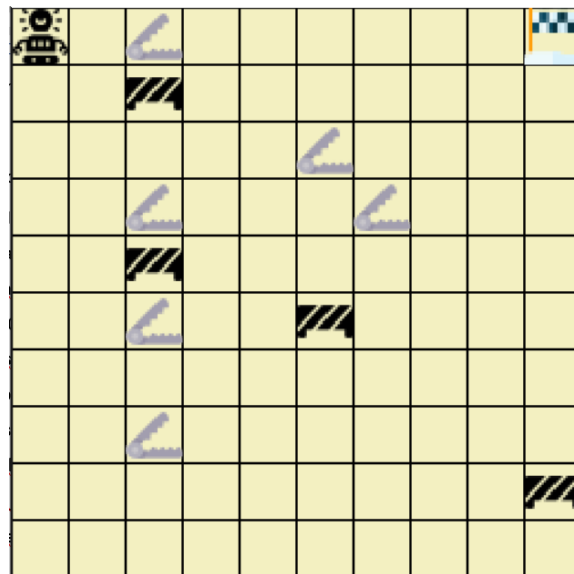
Para invocar el entorno con su configuración por defecto lo haríamos de la siguiente forma:

```
env = gym.make('robotUA-v0')
```

Los valores de los parámetros son los siguientes:

- **MAX_MOVIMIENTOS:** 1000
- **visualizar:** True
- **tam_matrix:** (10,10)
- **delay:** 50
- **tam_casilla:** 50
- **objetivos:** [(9, 0)]
- **trampas:** ([2, 0], (2, 3), (5, 2), (2, 5), (2, 7), (6, 3)]
- **obstaculos:** [(5, 5), (2, 1), (2, 4), (9, 8)]
- **incluir_delay:** 7
- **delay_incluido:** 150

Este sería el resultado:



3.3. Ficheros JSON

Los ficheros de configuración en formato JSON nos ayudan a mantener una persistencia de las diferentes configuraciones para la invocación del entorno.

Estos ficheros deben estar situados en la carpeta **config** situada en la raíz del proyecto. También, es posible distribuirlos en subcarpetas dentro de **config**. Esto sería un ejemplo de invocación del entorno configurado mediante un JSON:

```
env = gym.make('robotUA-v0', config="sinSubobjetivo/15x15.json")
```

Los ficheros tienen parámetros obligatorios y otros opcionales (estos cogerán un valor por defecto establecido en el código) que pueden no estar en el fichero.

Parámetros obligatorios:

- MAX_MOVIMIENTOS.
- visualizar.
- tam_matrix.
- objetivos.

Parámetros opcionales y valores predeterminados (en caso de no encontrarse en el JSON):

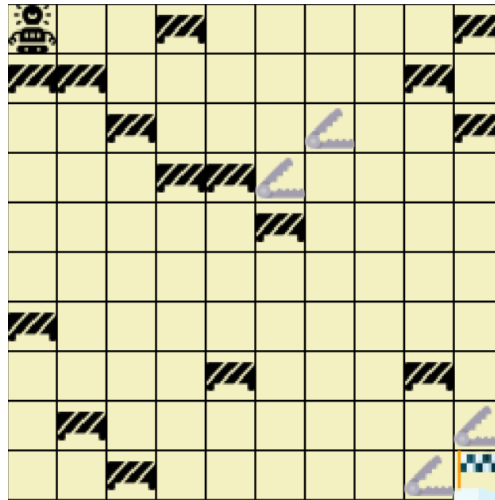
- delay, valor predeterminado: 0.
- tam_casilla, valor predeterminado: 35.
- trampas, valor predeterminado: array vacío.
- obstaculos, valor predeterminado: array vacío.
- incluir_delay, valor predeterminado: 7.
- delay_incluido, valor predeterminado: 100.

3.4. Ejemplos de ejecución con configuración de ficheros JSON

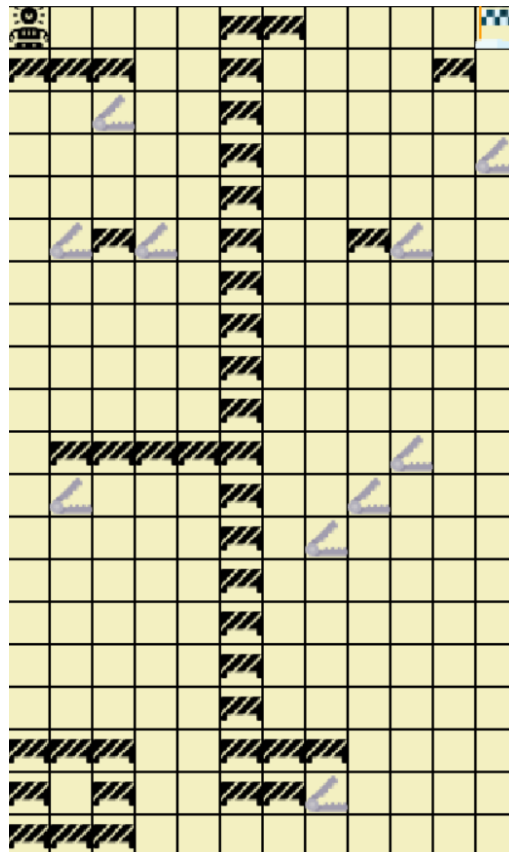
A continuación, mostraremos algunos circuitos de ejemplo aportados junto con el proyecto:



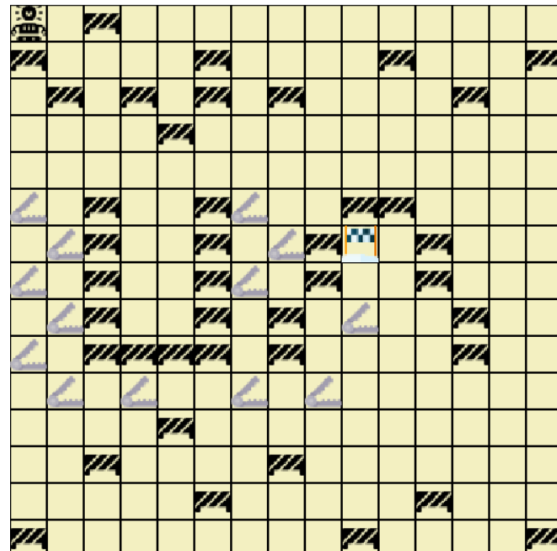
Fichero 1: config/sinSubobjetivo/5x5.json



Fichero 2: config/sinSubobjetivo/10x10.json



Fichero 3: config/sinSubobjetivo/12x20.json



Fichero 4: config/sinSubObjetivo/15x15.json

4. Ejemplo de utilización del entorno

El fichero **prueba.py** es un ejemplo de utilización de este entorno. Este fichero toma una configuración predeterminada, trata de resolver el circuito 10 veces realizando movimientos aleatorios y muestra movimiento a movimiento como avanza el robot sobre el circuito. En ningún caso es un algoritmo de aprendizaje, es solo un ejemplo de uso del entorno.

```
1  import gym
2  import gym_robotUA
3
4  env = gym.make('robotUA-v0')
5  done = False
6  observation = env.reset()
7  intento = 0
8
9  while intento < 10:
10     print("-----Intento", intento, "-----")
11     while not done:
12         #env.action_space.sample devuelve un numero entero entre 0 y 3, ambos incluidos.
13         #0: movimiento izquierda
14         #1: movimiento arriba
15         #2: movimiento derecha
16         #3: movimiento abajo
17         action = env.action_space.sample()
18         observation, reward, done, _ = env.step(action) #(_ es un valor que no lo usaremos)
19         env.render()
20         if done:
21             print("Movimiento realizados:\t", env.num_mov)
22             print("Trampas pisadas:\t", env.trampas_pisadas)
23             print("")
24             intento += 1
25             env.reset()
26             env.render()
27
28     done = False
29
30 print("-----")
31 print("Numero de veces completado:\t", env.episodios_completados)
32 print("Movimientos totales realizados:\t", env.num_mov_totales)
33 print("Trampas pisadas totales:\t", env.trampas_totales)
34 print("")
35 env.close()
```

Fichero prueba.py

Lo primero es importar tanto la librería gym y el entorno gym_robotUA.

Lo siguiente, es crear el entorno con sus valores por defecto (están explicados en apartados anteriores).

Después, mientras el número de veces que se ha intentado resolver el circuito sea inferior a 10 ,trataremos de resolverlo. Para ello, obtenemos una acción aleatoria dentro del espacio de acciones posibles (un numero entre 0 y 3, ambos incluidos). La función **step** es la encargada de realizar el movimiento del robot (siempre que cumpla las condiciones), el movimiento a realizar es el parámetro que se la pasa. Esta función devuelve 4 parámetros: **observation** (contendrá la posición del robot después de realizar el movimiento), **reward** (recompensa asociada al movimiento realizado), **done** (si se ha completado o no el circuito) y la cuarta variable que no usaremos en este entorno.

A continuación, la función **render** es la encargada de actualizar la imagen después de haber realizado el movimiento.

Después de todo esto, si el circuito ha sido resuelto (recompensa = 1) o se ha llegado al número máximo de movimientos (recompensa = -1) **done** tendrá un valor True, se incrementará el contador del bucle exterior, y se reiniciará el entorno a sus valores iniciales. A continuación, se llama de nuevo a la función **render** para que se visualice el entorno reiniciado.

Los movimientos realizados en las primeras 7 veces se realizarán más rápido que las 3 últimas. Esto es así porque se entiende que en las primeras iteraciones el algoritmo vaya aprendiendo y tenga un número de movimientos más elevado que en las iteraciones finales.

Finalmente, se imprime cierta información relevante a la resolución del circuito y se cierra el programa.

Este sería un ejemplo de salida:

```

-----Intento 0 -----
Movimiento realizados: 387
Trampas pisadas: 16

-----Intento 1 -----
Movimiento realizados: 1000
Trampas pisadas: 98

-----Intento 2 -----
Movimiento realizados: 568
Trampas pisadas: 19

-----Intento 3 -----
Movimiento realizados: 1000
Trampas pisadas: 77

-----Intento 4 -----
Movimiento realizados: 456
Trampas pisadas: 30

-----Intento 5 -----
Movimiento realizados: 302
Trampas pisadas: 21

-----Intento 6 -----
Movimiento realizados: 238
Trampas pisadas: 19

-----Intento 7 -----
Movimiento realizados: 542
Trampas pisadas: 26

-----Intento 8 -----
Movimiento realizados: 389
Trampas pisadas: 39

-----Intento 9 -----
Movimiento realizados: 470
Trampas pisadas: 29

-----
Numero de veces completado: 10
Movimientos totales realizados: 5352
Trampas pisadas totales: 374

```

Estas iteraciones se han resuelto con un delay de 50 milisegundos entre movimientos.

Las ultimas iteraciones se han resuelto con un delay de 150 milisegundos entre movimientos.

5. Instalación del entorno

Para poder ejecutar el entorno necesitaremos tener **una versión de Python igual o superior a la 3.5**.

Una vez la tengamos, necesitamos descargar la librería gym. Desde el entorno de Python adecuado ejecutaremos el siguiente comando: ***pip install gym***.

A continuación, iremos a la carpeta del proyecto con el comando ***cd <ruta del proyecto>***, y ejecutaremos el siguiente comando: ***pip install -e***. Esto creará los archivos necesarios para poder ejecutar nuestro entorno gym personalizado.

Para comprobar que todo funciona correctamente podemos ejecutar el comando ***python prueba.py***, esto ejecutará el código de ejemplo aportado con el proyecto.

6. Bibliografía.

Gym, OpenAI:

<https://gym.openai.com>

Pygame:

<https://www.pygame.org/news>

Python:

<https://www.python.org/downloads/>

Making a custom environment in gym:

<https://medium.com/@apoddar573/making-your-own-custom-environment-in-gym-c3b65ff8cdaa#:~:text=To%20create%20a%20different%20version,version%20number%20we%20are%20at.>

