Práctica 1

Laura Rodríguez Navas
rodrigueznavas@posgrado.uimp.es

3 de abril de 2021

Ejercicio 1

1. Descargar el código fuente para esta práctica, *softpractica1.zip*, de la página web de la asignatura.
2. Descomprimir el fichero anterior.
3. Abrir un terminal o consola de comandos y entrar dentro de la carpeta *softpractica1*.
4. Para empezar vamos a ejecutar GridWorld en el modo de control manual, usando el comando *python gridworld.py -m -n 0*, que utiliza las teclas de flecha.
5. El objetivo es lograr llegar lo antes posible a la celda etiquetada con un 1, evitando caer en la celda con un -1.

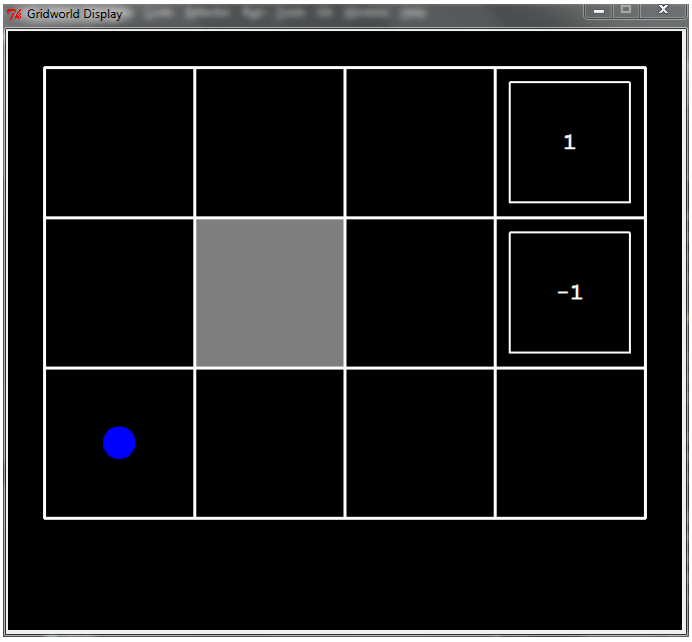

Figura 1: Interfaz del dominio GridWorld en el modo de control manual.

Preguntas

1. ¿Cuántas celdas/estados aparecen en el tablero? ¿Cuántas acciones puede ejecutar el agente? Si quisieras resolver el juego mediante aprendizaje por refuerzo, ¿cómo lo harías?

   En el tablero aparecen 11 celdas/estado y el agente puede ejecutar 4 acciones: arriba (*north*), abajo (*south*), izquierda (*west*) y derecha (*east*).

   Para resolver el juego usaremos el algoritmo Q-learning con el objetivo de que el agente llegue lo antes posible a la celda etiquetada con un 1 (estado *done*), evitando caer en la celda con un -1 (estado *exit*). La clave del algoritmo Q-learning será la construcción de su tabla Q, que es una matriz donde tendremos las recompensas que obtendrá el agente para cada acción y en cada estado, es decir, los valores de $Q(s, a)$. El algoritmo hará que el agente vaya tomando decisiones y a cada decisión actualizará uno de los valores de $Q(s, a)$ de la tabla Q. A la hora de tomar las decisiones el algoritmo usa la estrategia ϵ-greedy. Esta estrategia consiste en que todas las acciones sean tomadas buscando el valor máximo de $Q(s, a)$ pero existiendo una probabilidad pequeña, ϵ, de tomar una decisión aleatoria para que el agente explore todo el espacio de soluciones. Para la actualización de la tabla Q e ir completándola, cada decisión tomada por el agente se evaluará por la siguiente expresión:

   $$\hat{Q}(s_t, a_t) \leftarrow (1 - \alpha)\, \hat{Q}(s_t, a_t) + \alpha_t \left[r_{t+1} + \gamma\, max_a\, \hat{Q}(s_{t+1}, a)\right] \tag{1}$$

   En esta expresión $\hat{Q}(s_t, a_t)$ es el valor de la tabla Q del estado s_t y la acción a_t. $\hat{Q}(s_t, a_t)$ será el nuevo valor de la tabla Q para dicho estado y acción. El valor γ es el *learning rate*, que puede tomar valores entre 0 y 1. Finalmente r_{t+1}, determina la recompensa inmediata asociada a la acción tomada y es el *discount factor* que también puede tomar valores entre 0 y 1.

2. Abrir el fichero *qlearningAgents.py* y buscar la clase *QLearningAgent*. Describir los métodos que aparecen en ella.

   Los métodos que aparecen en la clase *QLearningAgent* son:

   - **__init__**: Inicializa la *Q-Table* a partir del fichero *qtable.txt*, es decir, la *Q-Table* se inicializa a cero.
   - **readQtable**: Lee la *Q-Table* del fichero *qtable.txt*.
   - **writeQtable**: Escribe la *Q-Table* en el fichero *qtable.txt*.
   - **__del__**: Llama al método *writeQtable* que escribe el resultado final de la *Q-Table* en el fichero *qtable.txt*.
   - **computePosition**: Calcula la fila de la *Q-Table* para un estado dado.
   - **getQValue**: Devuelve el valor $Q(s, a)$ para un estado y una acción dados. De lo contrario, devuelve 0.0, si nunca hemos visto el estado o el valor del nodo Q.
   - **computeValueFromQValues**: Devuelve el valor máximo de $Q(s, a)$ para un estado dado. Este valor se encuentra por encima de las acciones válidas. Si no hay acciones válidas, como en el caso del estado *exit*, devuelve 0.0.
   - **computeActionFromQValues**: Calcula la mejor acción a realizar para un estado dado. Si no hay acciones válidas, como en el caso del estado *exit*, devuelve *None*.

- **getAction**: Calcula la acción a realizar para un estado dado. En caso contrario, con probabilidad *self.epsilon*, elige una acción aleatoria y la mejor acción política. Si no hay acciones válidas, como en el caso del estado *exit*, elige *None* como acción.
- **update**: Actualiza la *Q-Table*. El método para un acción dada, observa una recompensa, introduce un estado nuevo (que depende del estado anterior y de la acción dada), y actualiza el valor $Q(s, a)$.

  Si el nuevo estado introducido es el estado *exit*, se sigue la regla:

  $$Q(state, action) < -(1 - self.alpha) * Q(state, action) + self.alpha * (reward + 0)$$

  De lo contrario, si el nuevo estado introducido no es el estado *exit*, se sigue la regla:

  $$Q(state, action) < -(1 - self.alpha) * Q(state, action) + self.alpha * (reward + self.discount * max_{a'} Q(nextState, a'))$$

- **getPolicy**: Devuelve la mejor acción de la *Q-Table* para un estado dado.
- **getValue**: Devuelve el valor $Q(s, a)$ más alto para un estado dado.

3. Ejecuta ahora el agente anterior con: *python gridworld.py -a q -k 100 -n 0.*

   A diferencia de la primera ejecución, en esta ejecución le indicamos el tipo de agente, que en este caso es q, y el número de movimientos/episodios MDP a realizar, que en este caso son 100.

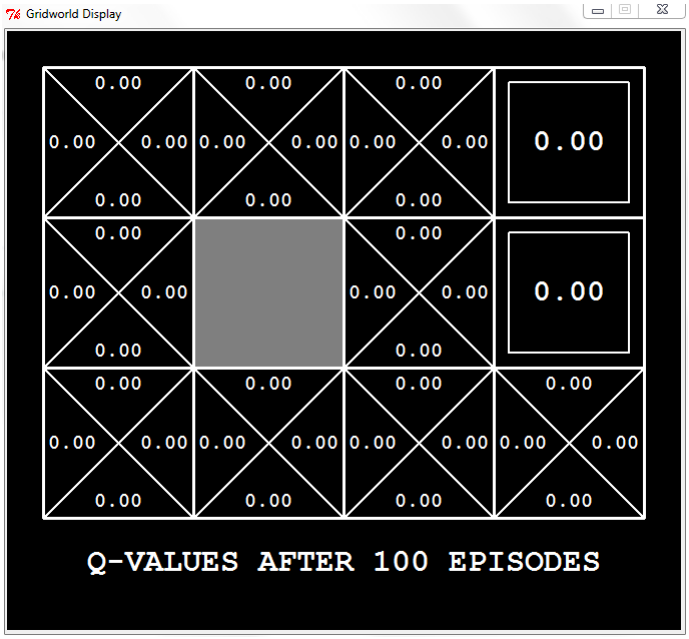

Figura 2: Interfaz del dominio GridWorld usando *python gridworld.py -a q -k 100 -n 0.*

4. ¿Qué información se muestra en el laberinto? ¿Qué aparece por terminal cuando se realizan los movimientos en el laberinto?

   Si observamos la Figura 2, vemos que la información que se muestra en el laberinto son los valores de $Q(s, a)$ después de realizarse 100 movimientos/episodios. Como la función de actualización (*update*) de los valores de $Q(s, a)$ no se ha implementado aún, observamos que todos los valores de $Q(s, a)$ después de realizarse 100 movimientos son 0.

En la terminal cuando se realizan los movimientos/episodios, vemos que aparecen los siguientes valores por cada episodio hasta el estado TERMINAL_STATE (*exit*):

- La posición (x, y) donde empieza el estado. Por ejemplo: (2, 1).
- La acción tomada. Por ejemplo: derecha (*east*).
- La posición (x, y) donde acaba el estado. Por ejemplo: (3, 1).
- La recompensa obtenida. Por ejemplo: 0.0, en este caso no ha habido recompensa.

Cuando se ha alcanzado en cada episodio el estado TERMINAL_STATE, vemos que aparece por la terminal el valor máximo de $Q(s, a)$ de cada episodio. Por ejemplo: EPISODE 93 COMPLETE: RETURN WAS 0.0182480036314. Si hemos alcanzado el objetivo (la celda etiquetada con un 1) durante un episodio, tendremos una recompensa igual a 1 y el valor máximo de $Q(s, a)$ será positivo. Por el contrario, el valor máximo de $Q(s, a)$ será negativo y tendremos una recompensa igual a -1, al no alcanzar el objetivo y caer en la celda etiquetada con un -1. Así pues, consideramos que hemos ganado en un episodio si el valor máximo de $Q(s, a)$ del episodio es positivo, y consideramos que hemos perdido si el valor máximo de $Q(s, a)$ del episodio es negativo. En el caso del ejemplo, vemos que en el episodio 93 el valor máximo de $Q(s, a)$ es positivo, ganamos.

Finalmente, aparece en la terminal un último valor. Este valor es el promedio de todos los valores máximos de $Q(s, a)$ de todos los movimientos/episodios de una ejecución. Por ejemplo: AVERAGE RETURNS FROM START STATE: -0.079260284124. Con el planteamiento del párrafo anterior, si el promedio es negativo consideramos que hemos perdido la partida, y por el contrario si este valor es positivo consideramos que hemos ganado la partida. En el caso del ejemplo, vemos que el promedio es negativo, perdemos.

5. ¿Qué clase de movimiento realiza el agente anterior?

   El agente desde una posición, siempre que decida moverse en una dirección por el laberinto, lo hace en esa dirección con probabilidad igual a 1, es decir, se mueve una posición.

   Este tipo de movimiento se define en un proceso de decisión de Markov (MDP, siglas en inglés) determinista, donde durante la ejecución de una acción desde un estado siempre se produce la misma transición de estado y la misma recompensa.

6. ¿Se pueden sacar varias políticas óptimas? Describe todas las políticas óptimas para este problema.

   En este caso, las políticas óptimas se inducen por la estimación de los valores máximos de $Q(s, a)$ y cambian después de cada movimiento/episodio. Los valores máximos de $Q(s, a)$ se almacenan en la tabla Q.

   Así pues, se pueden sacar varias políticas óptimas si observamos la tabla Q, que almacena todas la políticas óptimas para este problema. Concretamente observando la Tabla 1, o que es lo mismo, el fichero *qtable.txt* que contiene la tabla Q que almacena todas la políticas óptimas para este problema; vemos que tenemos 34 políticas óptimas donde parece que hemos alcanzado el objetivo (valores máximos de $Q(s, a) > 0$).

0.590489969903	0.590489976465	0.531440972641	0.531440969095	0.0
0.590489859224	0.656099982789	0.590489980107	0.5314409716	0.0
0.728999990732	0.59048968547	0.656099948059	0.590489940173	0.0
0.0	0.590488144602	0.590489453559	0.656099880246	0.0
0.656099976912	0.59048996484	0.531440967592	0.590489963482	0.0
0.0	0.0	0.0	0.0	0.0
0.809999995782	0.0	0.656099462996	0.728999845279	0.0
0.0	0.0	0.0	0.0	-1.0
0.65609997485	0.728999985783	0.590489975178	0.656099945627	0.0
0.728999987496	0.809999996062	0.728999945859	0.656099934053	0.0
0.809999962699	0.899999998166	0.728999985397	0.728999978711	0.0
0.0	0.0	0.0	0.0	0.999999999942

Tabla 1: Contenido del fichero *qtable.txt* cuando *epsilon* es igual a 1.

7. Escribir el método *update* de la clase *QLearningAgent* utilizando las funciones de actualización del algoritmo *Q-Learning*. Para ello, inserta el código necesario allí donde aparezca la etiqueta INSERTA TU CÓDIGO AQUÍ siguiendo las instrucciones que se proporcionan, con el fin de conseguir el comportamiento deseado.

   El código que se ha insertado en el método *update* de la clase *QLearningAgent*, siguiendo las instrucciones que se proporcionan, es:

```
position = self.computePosition(state)
action_column = self.actions[action]

if nextState != 'TERMINAL_STATE':
  # Q(state,action) <- (1-self.alpha) * Q(state,action) + self.alpha * (reward
    + self.discount * max a' Q(nextState, a'))
  sample = (1 - self.alpha) * self.getQValue(state, action) + self.alpha * (
     reward + self.discount * self.computeValueFromQValues(nextState))
  self.q_table[position][action_column] = sample

elif nextState == 'TERMINAL_STATE':
  # Q(state,action) <- (1-self.alpha) * Q(state,action) + self.alpha * (reward
    + 0)
  sample = (1 - self.alpha) * self.getQValue(state, action) + self.alpha * (
    reward + 0)
  self.q_table[position][action_column] = sample
```

   En la Figura 3 podemos observar que con el código insertado anteriormente, los valores de $Q(s, a)$ se han actualizado después de volver a ejecutar *python gridworld.py -a q -k 100 -n 0*.

8. Establece en el constructor de la clase *QLearningAgent* el valor de la variable *epsilon* a 0,05. Ejecuta nuevamente con: *python gridworld.py -a q -k 100 -n 0*. ¿Qué sucede?

   Cuando *epsilon* es igual a 0,05:

   - AVERAGE RETURNS FROM START STATE: 0.530784257974.

   Cuando *epsilon* es igual a 1:

   - AVERAGE RETURNS FROM START STATE: -0.079260284124.

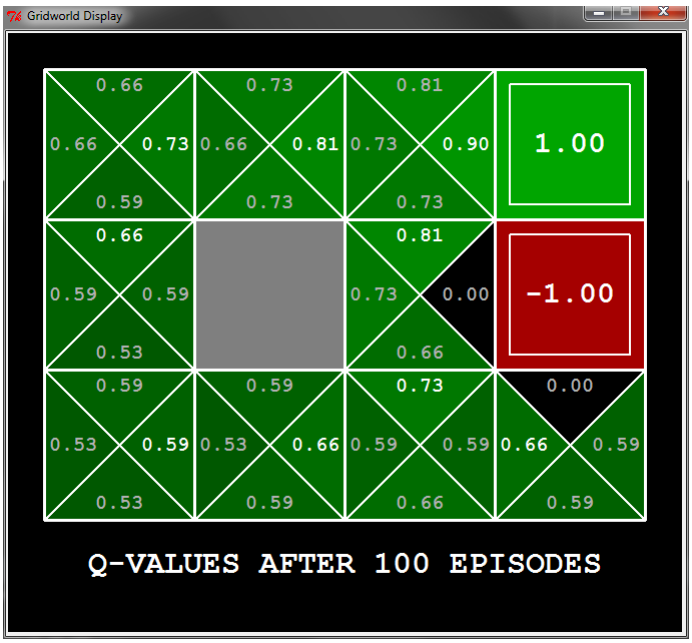

Figura 3: Interfaz del dominio GridWorld cuando *epsilon* es igual a 1.

Cuando *epsilon* es igual a 0,05, el promedio de todos los valores máximos de $Q(s, a)$ es positivo, por lo tanto, ganamos la partida después de todos los movimientos/episodios. Al contrario, cuando *epsilon* es igual a 1, el promedio de todos los valores máximos de $Q(s, a)$ es negativo, por lo tanto, perdemos la partida después de todos los movimientos/episodios. Eso se debe a la estrategia ϵ-greedy comentada anteriormente (ver Pregunta 1), donde si el valor de ϵ es más pequeño, la probabilidad de que el agente tome decisiones aleatorias será menor. Si la aleatoriedad es menor, menor será la inestabilidad del algoritmo Q-learning.

9. Después de la ejecución anterior, abrir el fichero *qtable.txt*. ¿Qué contiene?

   El fichero *qtable.txt* contiene la tabla Q que almacena los valores máximos de $Q(s, a)$, es decir, las políticas óptimas para este problema.

0.59049	0.119569809907	0.0	0.0	0.0
0.0	0.0	0.0	0.398575844342	0.0
0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.6561	0.153677329102	0.0	0.4428675	0.0
0.0	0.0	0.0	0.0	0.0
0.568245849609	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	-0.75
0.32805	0.729	0.295245	0.327896389246	0.0
0.0	0.81	0.364499433769	0.0	0.0
0.0	0.9	0.148078125	0.0	0.0
0.0	0.0	0.0	0.0	1.0

Tabla 2: Contenido del fichero *qtable.txt* cuando *epsilon* es igual a 0,05.

Figura 4: Interfaz del dominio GridWorld cuando *epsilon* es igual a 0,05.

Ejercicio 2

En el ejercicio anterior, siempre que el agente decidía moverse hacia una dirección se movía en esa dirección con probabilidad 1. Es decir, se trataba de un MDP determinista. Ahora vamos a crear un MDP estocástico.

Preguntas

1. Ejecuta y juega un par de partidas con el agente manual: *python gridworld.py -m -n 0.3*. ¿Qué sucede? ¿Crees que el agente *QLearningAgent* será capaz de aprender en este nuevo escenario? .

   Si ejecutamos *python gridworld.py -m -n 0.3*, con el parámetro -n que añade ruido en la ejecución, sucede que a veces la dirección indicada no resulta la dirección deseada.

   Creo que el agente *QLearningAgent* será capaz de aprender en este nuevo escenario, aunque al haber añadido ruido se podría añadir el problema de la sobreestimación, que es la propiedad más dañina resultante del ruido. Sobretodo porqué en entornos ruidosos el algoritmo Q-learning a veces puede sobrestimar los valores de la acción, retrasando el aprendizaje.

2. Reiniciar los valores de la tabla Q del fichero *qtable.txt*.

3. Ejecutar el agente *QLearningAgent*: *python gridworld.py -a q -k 100 -n 0.3*.

   Con esta nueva ejecución obtenemos:

   - La Figura 5.
   - La Tabla 3, que contiene la tabla Q del fichero *qtable.txt*.

0.553445100458	0.105618413429	0.199229717446	0.24706559221	0.0
0.340125559311	0.20729569976	0.142353922	0.0	0.0
0.492279805329	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.598455517406	0.349085290292	0.373117962712	0.352953780041	0.0
0.0	0.0	0.0	0.0	0.0
0.311843501505	0.0	0.0	0.0794004649324	0.0
0.0	0.0	0.0	0.0	-0.999755859375
0.536520688909	0.699846713972	0.284330620577	0.0	0.0
0.222647958984	0.788829333025	0.0	0.505855779933	0.0
0.730856716955	0.894327291772	0.486130902902	0.520064147566	0.0
0.0	0.0	0.0	0.0	1.0

Tabla 3: Contenido del fichero *qtable.txt* cuando *epsilon* es igual a 0,05 y existe ruido.

4. Tras unas cuantos episodios, ¿se genera la política óptima? Y si se genera, ¿se tarda más o menos que en el caso determinista?

   Tras unos cuantos episodios se genera la política óptima y cuando se genera tarda más que en el caso determinista. Si en el caso determinista, la mejor política óptima se encontraba en el movimiento/episodio 8, en el caso estocástico se encuentra en el episodio 22. Se tarda bastante más, seguramente por el problema de sobreestimación comentado anteriormente en la Pegunta 1.

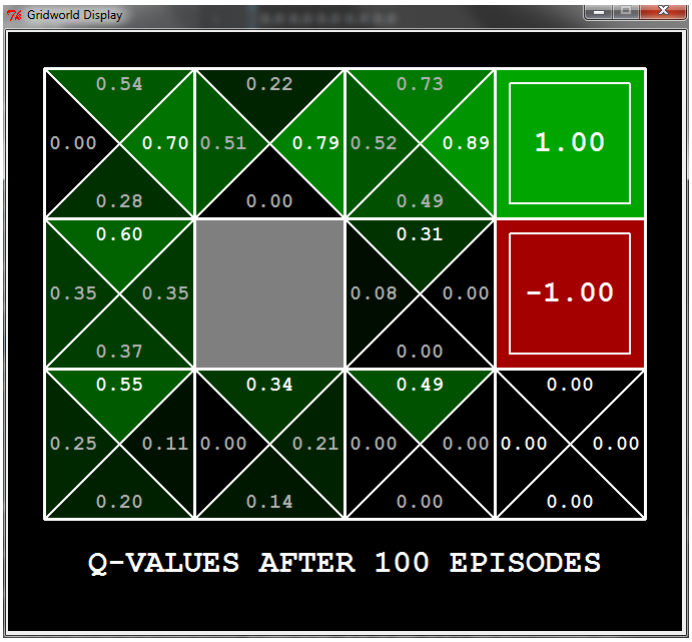

Figura 5: Interfaz del dominio GridWorld cuando *epsilon* es igual a 0,05 y existe ruido.

Todo el contenido de esta práctica se puede encontrar en el repositorio personal de GitHub: `https://github.com/lrodrin/masterAI/tree/master/A21/softpractica1`.