Práctica 2

Laura Rodríguez Navas
rodrigueznavas@posgrado.uimp.es

12 de mayo de 2021

1. Introducción

Esta práctica se apoya en una plataforma que recrea el videojuego clásico Pac-Man. Utiliza una versión muy simplificada del juego para poder desarrollar un sistema de control automático y explorar algunas capacidades del Aprendizaje por Refuerzo aprendidas durante la asignatura. Especialmente, en esta práctica se aplica el algoritmo *Q-learning* para construir un agente que funcione de forma automática en los tres mapas disponibles: *lab1.lay, lab2.lay y lab3.lay*. El objetivo de este agente será maximizar la puntuación obtenida en una partida.

Asimismo, en este documento se describen las tareas realizadas, que se dividen en distintas fases (definición de los estados, función de refuerzo, construcción del agente y evaluación), y que se detallan a continuación.

En la sección 2 se justifica el conjunto de atributos elegido y su rango para la definición de los estados. Una vez se ha elegido el conjunto de atributos que se van a utilizar para representar cada estado, en la sección 3 se detalla el diseño de la función de refuerzo que vamos a emplear y que permitirá al agente lograr su objetivo de maximizar la puntuación obtenida en una partida. Después de esto, en la sección 4 se explica cómo se ha procedido a la construcción del agente con el fin de funcionar bien en todos los laberintos. Cuando se obtiene el agente, en la sección 5 se presentan los resultados de las puntuaciones que ha obtenido en los mapas proporcionados y se añaden comentarios sobre su comportamiento. Finalmente, en la parte final del documento se añaden las conclusiones (sección 6) y un apéndice que contiene métodos auxiliares del código desarrollado en las secciones 3 y 4.

2. Definición de los estados

En esta sección, a fin de definir los estados, hay que seleccionar unos valores adecuados para los parámetros: *self.nRowsQTable, self.alpha, self.gamma y self.epsilon*. Primero intentaremos descubrir el valor del parámetro *self.nRowsQTable*, o que es lo mismo, intentaremos descubrir cuántas filas debe tener la *QTable*. Este valor dependerá de las características seleccionadas para representar los estados, ya que según estas características tendremos un número diferente de filas en nuestra *QTable*. Nos fijamos en la información que nos proporciona la función *printInfo*, concretamente en la información de los mapas *Walls* y *Food*. Según esta información y las acciones que puede ejecutar el agente, las características que se han elegido son:

- nearest_ghost_north, no_wall
- nearest_ghost_south, no_wall
- nearest_ghost_north, no_food
- nearest_ghost_south, no_food

- nearest_ghost_east, no_wall
- nearest_ghost_west, no_wall
- nearest_ghost_north, wall
- nearest_ghost_south, wall
- nearest_ghost_east, wall
- nearest_ghost_west, wall
- nearest_ghost_east, no_food
- nearest_ghost_west, no_food
- nearest_ghost_north, food
- nearest_ghost_south, food
- nearest_ghost_east, food
- nearest_ghost_west, food

Vemos que se han elegido 16 características, así que el valor del parámetro *self.nRowsQTable* será igual a 16. Modificaremos su valor en la función *registerInitialState* de la clase *RLAgent(BustersAgent)* del archivo *bustersAgents.py*, como se puede observar a continuación:

```
self.nRowsQTable = 16
```

A la hora de dar valor al resto de parámetros (*self.alpha, self.gamma y self.epsilon*), entrenaremos a un agente *Q-Learning* completamente aleatorio, con una tasa de aprendizaje determinada durante 50 episodios y observaremos si encuentra la política óptima. Para ello, haremos uso de la práctica 1 de la asignatura, pero esta vez solo usaremos 50 episodios para no tardar demasiado en encontrar la política óptima. Empezamos cambiando el valor predeterminado de aprendizaje (*epsilon*) y de descuento (*gamma*) a 0.05 y 0.8 respectivamente, copiándolos de la función *__init__* de la clase *PacmanQAgent(QLearningAgent)* del fichero *qlearningAgents.py*, porqué me pareció una buena elección. Modificados los valores, entrenamos el agente con el siguiente comando:

```
python gridworld.py -a q -k 50 -n 0.2
```

Vemos el resultado con una tasa de ruido igual a 0.2:

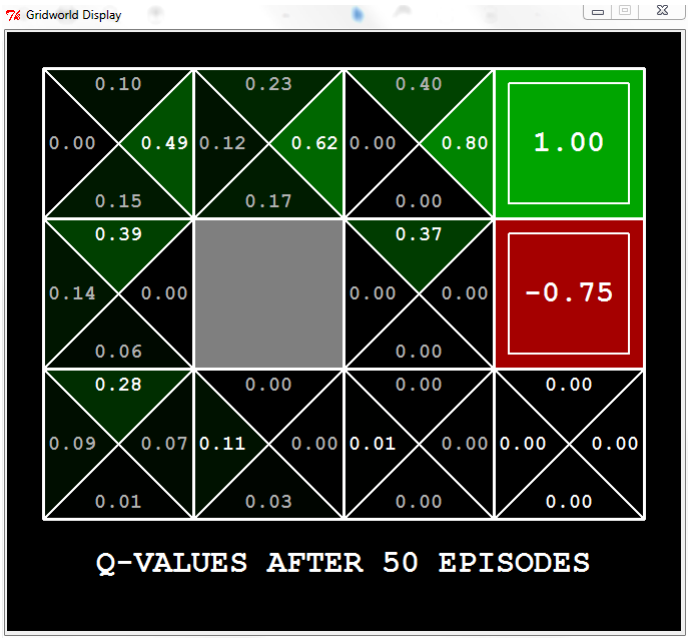

Figura 1: Interfaz del dominio GridWorld cuando *epsilon* es igual a 0.05.

Volvemos a ejecutar el mismo comando, pero con otro valor de aprendizaje. La Figura 2 muestra el resultado de la ejecución con una tasa de aprendizaje igual a 1.

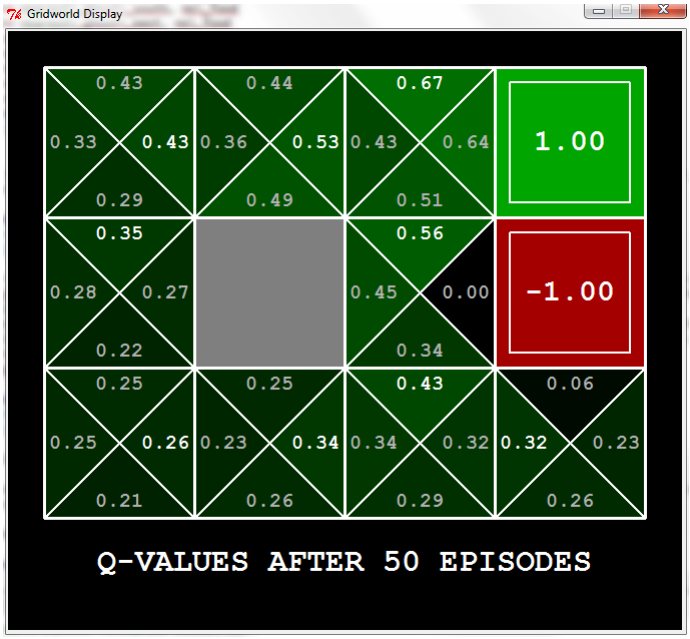

Figura 2: Interfaz del dominio GridWorld cuando *epsilon* es igual a 1.

Igual que descubrimos en la práctica 1, vemos que con un valor de aprendizaje menor (Figura 1), el agente no explora tanto y elige una ruta óptima mejor. En consecuencia modificaremos los valores de *self.alpha, self.gamma y self.epsilon* en el archivo *bustersAgents.py* como se puede observar a continuación:

```
self.alpha = float(0.2)
self.gamma = float(0.8)
self.epsilon = float(0.05)
```

Con este procedimiento nos aseguramos de elegir unos valores adecuados para Pac-Man.

3. Función de refuerzo

```
def getReward(self, state, nextState):
  """
  Return a reward value based on the information of state and nextState
  """
  reward = 0

  if nextState.isWin():
  return 1000

  next_state_ghost_index_distances = self.getLivingGhostIndexDistances(nextState)
  actual_state_ghost_index_distances = self.getLivingGhostIndexDistances(nextState)
  min_distance_ghost_index_next_State = self.getMinIndex(
    next_state_ghost_index_distances)[0]
  min_distance_ghost_index_actual_State = self.getMinIndex(
    actual_state_ghost_index_distances)[0]

  min_ghost_distance_next_state = nextState.data.ghostDistances[
    min_distance_ghost_index_next_State]
```

```
    min_ghost_distances_actual_state = state.data.ghostDistances[
      min_distance_ghost_index_actual_State]
    number_ghost_actual_state = len(self.getAliveGhostDistances(state.data.
      ghostDistances))
    number_ghost_next_state = len(self.getAliveGhostDistances(nextState.data.
      ghostDistances))
    actual_state_has_walls = self.directionIsBlocked(state, state.getGhostPositions()[
      min_distance_ghost_index_next_State])
    next_state_has_walls = self.directionIsBlocked(nextState, nextState.
      getGhostPositions()[min_distance_ghost_index_next_State])

    if number_ghost_next_state < number_ghost_actual_state:
    reward += 100

    if min_ghost_distance_next_state < min_ghost_distances_actual_state and not
      actual_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
    reward += 3

    elif min_ghost_distance_next_state > min_ghost_distances_actual_state and
      actual_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
    reward += 1

    elif min_ghost_distance_next_state < min_ghost_distances_actual_state and
      actual_state_has_walls \
    and number_ghost_next_state == number_ghost_actual_state:
    reward += -1

    elif (min_ghost_distance_next_state > min_ghost_distances_actual_state and not
      actual_state_has_walls) or \
    (min_ghost_distance_next_state == min_ghost_distances_actual_state and
      number_ghost_next_state == number_ghost_actual_state):
    reward += -min_ghost_distance_next_state

    # If next_state_has_walls
    if not actual_state_has_walls and next_state_has_walls and number_ghost_next_state
       == number_ghost_actual_state:
    reward -= 4
    elif actual_state_has_walls and not next_state_has_walls and
      number_ghost_next_state == number_ghost_actual_state:
    reward += 1

    return reward
```

4. Código desarrollado

```
def update(self, state, action, nextState, reward):
  """
  The parent class calls this to observe a
  state = action => nextState and reward transition.
  You should do your Q-Value update here
  """
  reward = reward + self.getReward(state, nextState)  # Calculate reward
```

```
  print "Started in state:"
  self.printInfo(state)
  print "Took action: ", action
  print "Ended in state:"
  self.printInfo(nextState)
  print "Got reward: ", reward
  print "------------------------------------"

  position = self.computePosition(state)
  action_column = self.actions[action] - 1
  sample = reward + self.gamma * self.getValue(nextState)

  if nextState.isWin():   # If a terminal state is reached
    self.writeQtable()

  else:
    self.q_table[position][action_column] = (1 - self.alpha) * self.q_table[position
    ][action_column] + self.alpha * sample
```

Este proyecto se divide en varias tareas pequeñas que deben completarse en consecuencia. La primera tarea gira en torno al diseño de Value-Iteration Agent. Este agente puede planificar su acción dado el conocimiento del entorno antes incluso de interactuar con él. Para ello, debemos calcular los valores Q de las acciones que seguirá el agente y elegir la mejor acción que devolverá el valor Q máximo. Después de diseñar el agente de iteración de valor, necesitamos cambiar el valor de descuento, ruido y vida. recompensa para lograr la mejor política óptima.

La siguiente tarea es escribir código para Q learning agent. Este agente, a diferencia del agente de iteración de valor, necesita interactuar activamente con el entorno para que pueda aprender a través de las experiencias. Necesitamos aproximar la función Q de los pares estado-acción a partir de las muestras de Q (s, a) que observamos durante la interacción con el medio ambiente. Después de una aproximación exitosa, el agente debería poder tomar la mejor acción para lograr el objetivo. Q-learning Pacman debería poder ganar al menos el 80 % del tiempo.

Sin embargo, a medida que intentamos utilizar este agente en diseños más grandes, por ejemplo, el mundo Grid medio El entrenamiento de PacMan de alguna manera ya no funcionará bien. Para resolver este problema, debemos optimizar nuestro agente para implementar también un agente Q-learning aproximado que aprenda los pesos de características de los estados, donde muchos estados pueden compartir las mismas características.

5. Resultados

6. Conclusiones

Todo el contenido de esta práctica se puede encontrar en el repositorio personal de GitHub: `https://github.com/lrodrin/masterAI/tree/master/A21/softpractica2`.

7. Apéndice

7.1. Métodos de la función *get_reward*

7.2. Métodos de la función *update*