Tabla de contenido

Pacman busca recompensa.
¿Debería comer o debería correr?
En caso de duda, Q-learning.

Nota: si estáis empleando una versión python superior a la 3.8 el cgi puede daros problemas. Sustituir el cgi.escape por html.escape o incluir cgi.escape = html.escape para no tener que cambiar cada instancia.

Introducción

En este proyecto, implementarás iteración de valor y Q-learning. Primero probarás a tus agentes en Gridworld, luego aplicarás lo programado a un Pacman.

Como en proyectos anteriores, este proyecto incluye un autograder para que califiques tus soluciones en tu máquina. Este permite calificar todas las preguntas con el comando:

```
python autograder.py
```

Se puede ejecutar para una pregunta en particular, como q2, mediante:

```
python autograder.py -q q2
```

Se puede ejecutar para un test en particular mediante comandos de la forma:

```
python autograder.py -t test_cases / q2 / 1-bridge-grid
```

El código de este proyecto contiene los siguientes archivos, disponibles en el eGela.

Archivos que editarás:	
valueIterationAgents.py	Un agente de iteración de valor para resolver MDP conocidos.
qlearningAgents.py	Agentes de Q-learning para Gridworld, robot “andador” y Pacman.
analysis.py	Un archivo para poner tus respuestas a las preguntas planteadas en el proyecto.
Archivos que debes leer pero NO editar:	
mdp.py	Define métodos en MDP generales.
learningAgents.py	Define las clases base ValueEstimationAgentyQLearningAgent, que extenderán tus agentes.
util.py	Utilidades, incluidas util.Counter, que es particularmente útil para Q-learners.
gridworld.py	La implementación de Gridworld.
featureExtractors.py	Clases para extraer características en pares (estado, acción). Se utiliza para el agente Q-learning aproximado (en qlearningAgents.py).
Archivos que puedes ignorar:	
environment.py	Clase abstracta para entornos generales de aprendizaje por refuerzo. Usado por gridworld.py.
graphicsGridworldDisplay.py	Pantalla gráfica Gridworld.
graphicsUtils.py	Utilidades gráficas.
textGridworldDisplay.py	Complemento para la interfaz de texto Gridworld.
crawler.py	El código del robot “andante” y su prueba. Se puede ejecutar

	pero no lo editarás.
graphicsCrawlerDisplay.py	GUI para el robot de orugas.
autograder.py	Autocalificador de proyectos
testParser.py	Analiza archivos de prueba y solución de autograder
testClasses.py	Clases generales de prueba de calificación automática
test_cases/	Directorio que contiene los casos de prueba para cada pregunta
reinforcementTestClasses.py	Clases de prueba de calificación automática específicas de Project 3

Archivos para editar y que serán evaluados: Tendrás que rellenar partes de valueIterationAgents.py, qlearningAgents.py y analysis.py durante el laboratorio. Por favor, no cambies los otros archivos.

Evaluación: tu código se calificará automáticamente para verificar su corrección técnica. Por favor, no cambies los nombres de las funciones o clases establecidas en el código, o causarás estragos en el autograder.

MDP

Para comenzar, ejecuta Gridworld en modo de control manual, que usa las teclas de flecha:

```
python gridworld.py -m
```

Verás el diseño de dos salidas de la clase. El punto azul es el agente. Ten en cuenta que cuando presiona hacia arriba, el agente solo se mueve hacia el norte el 80% del tiempo. ¡Así es la vida de un agente de Gridworld!. Esto sucede porque estamos en un MDP, donde cada acción tiene una probabilidad de llevarte a un estado u otro.

Puedes controlar muchos aspectos de la simulación. Una lista completa de opciones está disponible ejecutando:

```
python gridworld.py -h
```

El agente predeterminado se mueve aleatoriamente

```
python gridworld.py -g MazeGrid
```

Deberías ver al agente aleatorio rebotar alrededor de la cuadrícula hasta que encuentre una salida.

Nota: El MDP de Gridworld es tal que primero debe comenzar por un estado pre-terminal (las casillas dobles que se muestran en la GUI) y luego realizar la acción especial de 'salida' antes de que el episodio realmente termine (en el verdadero estado terminal llamado `TERMINAL_STATE`, que es no se muestra en la GUI). Si ejecutas un episodio manualmente, tu reward o premio total puede ser menor de lo esperado, debido a la tasa de descuento gamma (`-d` para cambiarla; 0.9 por defecto).

Mira la salida de la consola que acompaña a la salida gráfica (o utiliza la `-t` para verlo en modo texto). Se te informará sobre cada transición que experimenta el agente (para desactivar esto, usa `-q`).

Como en Pacman, las posiciones están representadas por `(x,y)` coordenadas cartesianas y las matrices están indexadas por `[x][y]`, `'north'` siendo la dirección de actualización de `y`, etc. De forma predeterminada, la mayoría de las transiciones recibirán una recompensa de cero, aunque puedes cambiar esto con la opción de live reward o premio de vida (`-r`).

Pregunta 1 (4 puntos): Iteración de valores

Recuerda la ecuación de actualización del estado de iteración de valor:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Escribe un agente de iteración de valor en `ValueIterationAgent`, en `valueIterationAgents.py`. Tu agente de iteración de valor es un planificador off-line, no un agente de reinforcement learning que aprende por episodios. Esto quiere decir que solo es el número de iteraciones de iteración de valor que debe ejecutar (opción `-i`) en su fase de planificación inicial lo que puede variar las cosas. `ValueIterationAgent` toma un MDP en la constructora y ejecuta el algoritmo de *iteración de valores* para el número especificado de iteraciones para calcular el Diccionario o Counter (dependiendo de como lo hayas definido) almacenado en self.values.

- `computeActionFromValues(state)` calcula la mejor acción de acuerdo con la función de valor dada por `self.values`. Es decir, de acuerdo al estado siguiente al que nos lleva cada acción y que mayor valor tiene. Este método llamará al siguiente.

- `computeQValueFromValues(state, action)` devuelve el valor Q del par (estado, acción) dado por la función de valor dada por `self.values`. **Nota:** Recuerda que para calcular el valor de un estado calcularemos los q_values (estado,acción), es decir los valores de las acciones posibles para quedarnos con el mayor (max de entre los q_values).

Todas estas cantidades se muestran en la GUI: los valores son los números que aparecen en cuadrados, los valores Q son números en cuartos cuadrados y las políticas son flechas que salen de cada cuadrado.

Esto puede ser importante cuando hagas las actualizaciones: Calcula todas las actualizaciones en la misma vuelta, no por cada acción. Esto significa que cuando el valor de un estado se actualiza en la iteración k con base a los valores de sus estados sucesores, los valores de estado sucesor usados en el cálculo de actualización de valor deben ser los de la iteración k-1 (incluso si algunos de los estados sucesores se podían haber actualizado en la iteración k). La diferencia se analiza en Sutton & Barto en el sexto párrafo del capítulo 4.1.

Sugerencia: opcionalmente, puedes usar la `util.Counter`clase en `util.py`, que es un diccionario con un valor predeterminado de cero. Sin embargo, ten cuidado con `argMax`: ¡el argmax real que deseas puede ser una clave que no está en el contador!, y al calcular el máximo, cuidado con la diferencia entre la condición >= y >, puede generar una diferencia en el autograder.

Nota: asegurate de considerar el caso cuando un estado no tenga acciones disponibles en un MDP (piense en lo que esto significa para las recompensas futuras).

Para probar tu implementación, ejecuta el autograder:

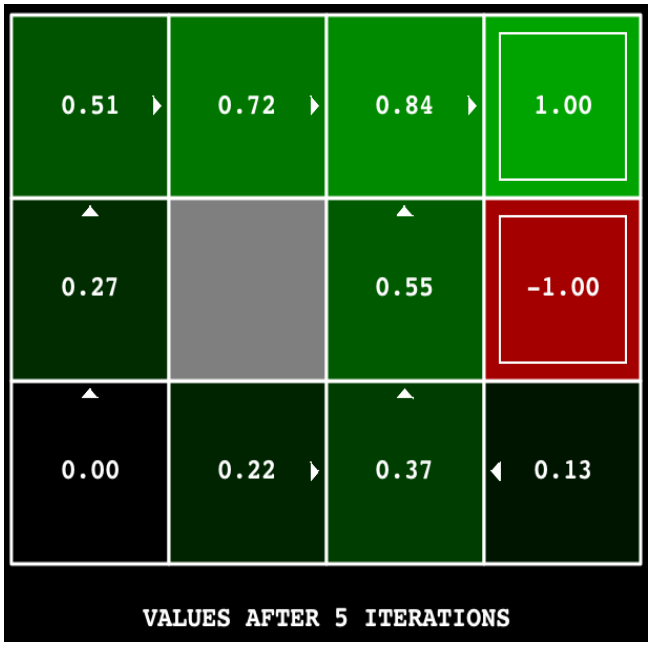
```
python autograder.py -q q1
```

El siguiente comando carga tu `ValueIterationAgent`, calculará una política y la ejecutará 10 veces. Presiona una tecla para recorrer los valores, los valores Q y la simulación. Debería encontrar que el valor del estado de inicio (`V(start)`que puede leer en la GUI) y la recompensa promedio resultante empírica (impresa después de las 10 rondas de finalización de la ejecución) están bastante cerca.

```
python gridworld.py -a value -i 100 -k 10
```

Sugerencia: en el gridworld predeterminado, la iteración del valor en ejecución para 5 iteraciones debería darte este resultado:

```
python gridworld.py -a value -i 5
```



Calificación: tu agente de iteración de valor se calificará en una nueva cuadrícula. El autograder comprobará tus valores, valores Q y políticas después de un número fijo de iteraciones y en la convergencia (por ejemplo, después de 100 iteraciones).

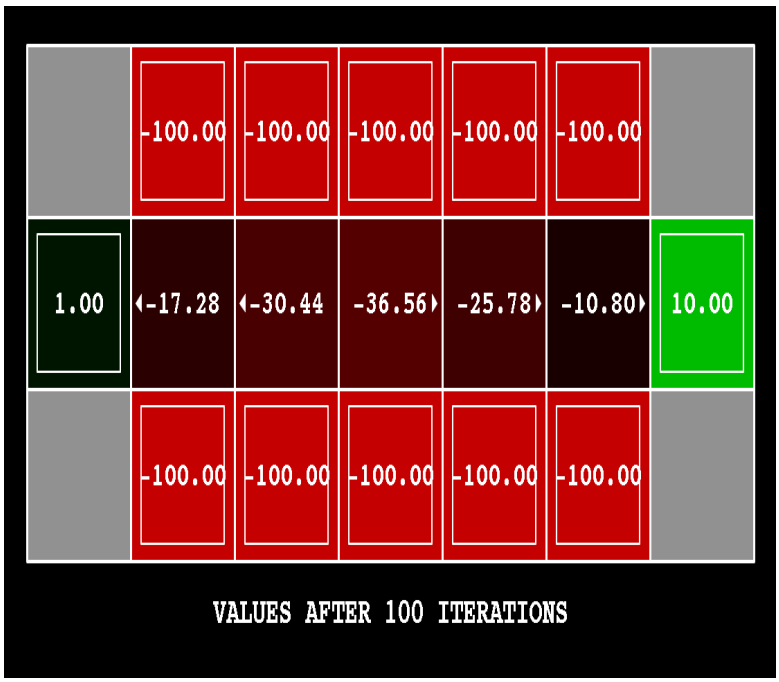
Pregunta 2 (1 punto): Análisis de cruce de puentes

`BridgeGrid` es GridWorld (un mapa de rejilla o cuadrícula) con un estado terminal de recompensa baja y un estado terminal de recompensa alta separados por un "puente" estrecho, donde a cada lado hay un abismo de recompensa negativa alta. El agente comienza cerca del estado de recompensa baja. Con un factor de descuento *gamma* de 0.9 y el factor *ruido*[1] predeterminado de 0.2, la política óptima no cruza el puente. Cambia solo UNO de los factores *gamma* o *ruido* para que la política óptima haga que el agente intente cruzar el puente. Pon tu respuesta en `question2()`de `analysis.py` (leer la nota al pie, el ruido se refiere a la probabilidad con la que un agente termina en un estado sucesor no esperado cuando realiza una acción). El valor predeterminado corresponde a:

```
python gridworld.py -a value

 -i 100 -g BridgeGrid --discount 0.9  --noise 0.2
```

1 self.noise = noise, es un factor que permite similar la realidad, porque la realidad no es perfecta y siempre existe una probabilidad de no poder moverte en la dirección que deseas. Así pues el factor de ruido es la probabilidad de moverte en una dirección que no prevista. Por ejemplo, en el ejemplo del coche, podría suceder que al querer realizar la acción de acelerar, no podamos porque hay una señal que nos lo impide, o no nos funciona bien el acelerador, etc. Este factor, introduce o simula ese tipo de indefinición en el sistema.



Calificación: El autograder verificará que solo hayas cambiado uno de los parámetros dados, y que con este cambio, un agente de iteración de valor correcto cruza el puente. Para verificar tu respuesta, ejecuta el autograder:

```
python autograder.py -q q2
```

Pregunta 3 (4 puntos): Q-Learning

Ten en cuenta que su agente de iteración de valor no aprende de la experiencia. Más bien, reflexiona sobre su modelo MDP (la función de transiciones y función de premios) para llegar a una política óptima antes de interactuar con un entorno real. Cuando interactúa con el entorno, simplemente sigue la política calculada previamente (por ejemplo, se convierte en un agente reflejo). Esta distinción puede ser sutil en un entorno simulado como Gridword, pero es muy importante en el mundo real, donde el MDP real no está disponible, es decir la función de transiciones y función de premios se desconoce.

Ahora escribirás un agente de Q-learning, que aprende por prueba y error de las interacciones con el entorno a través de su `update(state, action, nextState, reward)`método. Se especifica un código auxiliar de un Q-Learner `QLearningAgent`en `qlearningAgents.py`, y puedes seleccionarlo con la opción `'-a q'`. Para esta pregunta, debes implementar los métodos `update`, `computeValueFromQValues`, `getQValue`, y `computeActionFromQValues`.

Nota1: Inicialización de los q_values o valores q. Si en la constructora declaráis el atributo self.q_values como Counter() os inicializará todos los valores a 0 dado que un Counter() por defecto se inicializa a 0.

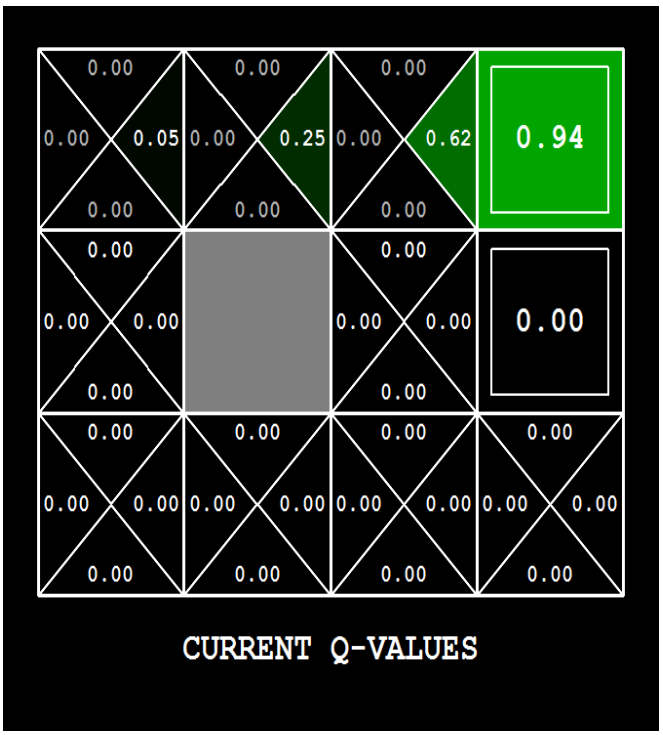
Nota2: Para `computeActionFromQValues`, debes romper los empates al azar para un mejor comportamiento. La función `random.choice()` te ayudará. En un estado en particular, las acciones que tu agente no ha visto antes tienen un valor Q, específicamente un valor Q de cero, y si todas las acciones que tu agente ha visto antes tienen un valor Q negativo, un valor no visto la acción puede ser óptima.

Importante: Asegurate de que en tus funciones `computeValueFromQValues` y `computeActionFromQValues`, solo accedan a los valores Q llamando `getQValue`. Esta abstracción será útil para la última pregunta cuando el `getQValue` deba emplear los rasgos (features) de pares de acción de estado en lugar de pares de acción de estado directamente.

Con la actualización de Q-learning implementada, puede ver a su Q-learner aprender bajo control manual, usando el teclado:

```
python gridworld.py -aq -k 5 -m
```

Recuerda que `-k`controlará la cantidad de episodios de los que tu agente puede aprender. Observa cómo el agente “aprende” en el estado en el que se encuentra, no al que se mueve, y "deja el aprendizaje a su paso". Sugerencia: para ayudar con la depuración, puedes desactivar el ruido usando el parámetro `--noise 0.0` (aunque esto obviamente hace que Q-learning sea menos interesante). Si diriges Pacman manualmente hacia el norte y luego hacia el este a lo largo de la ruta óptima durante cuatro episodios, deberías ver los siguientes valores Q:



Calificación: Ejecutaremos tu agente Q-learning y verificaremos que aprenda los mismos valores Q y política que nuestra implementación de referencia cuando cada uno se presenta con el mismo conjunto de ejemplos. Para calificar tu implementación, ejecute el autograder:

```
python autograder.py -q q6
```

Pregunta 4 (2 puntos): Epsilon Greedy

Completa tu agente de Q-learning implementando la selección de acciones *épsilon greedy* en `getAction`, lo que significa que elige acciones aleatorias una fracción épsilon del tiempo y, de lo contrario, sigue sus mejores valores Q actuales. Ten en cuenta que elegir una acción aleatoria puede resultar en la elección de la mejor acción, es decir, no debe elegir una acción aleatoria subóptima, sino *cualquier* acción legal aleatoria.

Puede elegir un elemento de una lista uniformemente al azar llamando a la función `random.choice`. Puedes simular el epsilón empleando una variable binaria con probabilidad `p` de éxito usando `util.flipCoin(p)`, que devuelve `True` con probabilidad `p` y `False`con probabilidad `1-p`.

Después de implementar el método `getAction`, observa el siguiente comportamiento del agente en gridworld (con epsilon = 0.3).

```
python gridworld.py -aq -k 100
```

Tus valores Q finales deben parecerse a los de su agente de iteración de valor, especialmente a lo largo de rutas muy transitadas. Sin embargo, sus rendimientos promedio serán más bajos de lo que predicen los valores Q debido a las acciones aleatorias y la fase de aprendizaje inicial.

También puedes observar las siguientes simulaciones para diferentes valores de épsilon. ¿Ese comportamiento del agente coincide con lo que esperas?

```
python gridworld.py -aq -k 100 - ruido 0.0 -e 0.1

python gridworld.py -aq -k 100 - ruido 0.0 -e 0.9
```

Para probar tu implementación, ejecuta el autograder:

```
python autograder.py -q q4
```

Sin código adicional, ahora deberías poder ejecutar un robot rastreador Q-learning:

```
python crawler.py
```

Si esto no funciona, probablemente hayas escrito un código demasiado específico para el problema `GridWorld` y deberías hacerlo más general para todos los MDP.

Esto invocará a un robot "andador" simulado usando su Q-learner. Prueba los diversos parámetros de aprendizaje para ver cómo afectan las políticas y acciones del agente. Tenga en cuenta que el retraso del paso es un parámetro de la simulación, mientras que la tasa de aprendizaje y épsilon son parámetros de tu algoritmo de aprendizaje, y el factor de descuento es una propiedad del entorno.

Pregunta 5 (1 punto): Q-Learning y Pacman

¡Es hora de jugar, Pacman! Pacman jugará juegos en dos fases. En la primera fase, entrenamiento, Pacman comenzará a aprender sobre los valores de las posiciones y acciones. Debido a que se necesita mucho tiempo para aprender valores Q precisos incluso para cuadrículas pequeñas, los juegos de entrenamiento de Pacman se ejecutan en modo silencioso de forma predeterminada, sin pantalla GUI (o consola). Una vez que se complete el entrenamiento de Pacman, pasará al modo de prueba. Al probar, Pacman `self.epsilon`y `self.alpha` se establecerá en 0.0, deteniendo efectivamente el Q-learning y deshabilitando la exploración, para permitir que Pacman explote su política aprendida. Los juegos de prueba se muestran en la GUI de forma predeterminada. Sin ningún cambio de código, deberías poder ejecutar Q-learning Pacman para cuadrículas muy pequeñas de la siguiente manera:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Ten en cuenta que `PacmanQAgent` ya está definido en términos de `QlearningAgent` lo que ya has escrito. `PacmanQAgent` solo es diferente porque tiene parámetros de aprendizaje predeterminados que son más efectivos para el problema de Pacman (`epsilon=0.05, alpha=0.2, gamma=0.8`). Recibirás crédito completo por esta pregunta si el comando anterior funciona sin excepciones y tu agente gana al menos el 80% del tiempo. El autograder ejecutará 100 juegos de prueba después de los 2000 juegos de entrenamiento.

Sugerencia: si tu `QlearningAgent` ha funcionado para `gridworld.py` y, `crawler.py` pero no parece estar aprendiendo una buena política para Pacman `smallGrid`, puede ser porque tus métodos `getAction` y/o `computeActionFromQValues` no consideran en algunos casos las acciones “invisibles” (no experimentadas hasta el momento). En particular, debido a que las acciones invisibles tienen por definición un valor Q de cero, si todas las acciones que se han visto tienen valores Q negativos, una acción invisible puede ser óptima. ¡Ten cuidado con la función argmax de util.Counter!

Nota: para calificar su respuesta, ejecute:

```
python autograder.py -q q5
```

Nota: Si deseas experimentar con los parámetros de aprendizaje, puedes usar la opción `-a`, por ejemplo `-a epsilon=0.1, alpha=0.3, gamma=0.7`. A continuación, estos valores serán accesibles como `self.epsilon, self.gamma`y `self.alpha` dentro del agente.

Nota: Si bien se jugará un total de 2010 juegos, los primeros 2000 juegos no se mostrarán debido a la opción `-x 2000`, que designa los primeros 2000 juegos para entrenamiento por episodios (sin salida). Por lo tanto, solo verás a Pacman jugar los últimos 10 de estos juegos. El número de juegos de entrenamiento también se pasa a tu agente como opción `numTraining`.

Nota: Si deseas ver 10 juegos de entrenamiento para ver qué está sucediendo, usa el comando:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining = 10
```

Durante el entrenamiento, verás resultados cada 100 juegos con estadísticas sobre cómo le está yendo a Pacman. Epsilon es positivo durante el entrenamiento, por lo que Pacman jugará mal incluso después de haber aprendido una buena política: esto se debe a que ocasionalmente hace un movimiento exploratorio aleatorio hacia un fantasma. Como punto de referencia, deberían pasar entre 1,000 y 1400 juegos

antes de que las recompensas de Pacman por un segmento de 100 episodios se vuelvan positivas, lo que refleja que comenzó a ganar más que a perder. Al final del entrenamiento, debería permanecer positivo y ser bastante alto (entre 100 y 350).

Asegúraté de comprender lo que está sucediendo: el estado de MDP es la configuración exacta del grip (mapa) que enfrenta Pacman, con las transiciones complejas que describen completamente el cambio a ese estado. Las configuraciones de juego intermedias en las que Pacman se ha movido pero los fantasmas no han respondido no son estados de MDP, sino que están incluidas en las transiciones.

Una vez que Pacman haya terminado de entrenar, debería ganar de manera muy fiable en los juegos de prueba (al menos el 90% del tiempo), ya que ahora está explotando su política aprendida.

Sin embargo, encontrarás que entrenar al mismo agente en el aparentemente simple `mediumGrid` no funciona bien. En nuestra implementación, las recompensas de entrenamiento promedio de Pacman siguen siendo negativas durante el entrenamiento. En el momento de la prueba, juega mal, probablemente perdiendo todos sus juegos de prueba. La fase de entrenamiento también llevará mucho tiempo, por su ineficacia.

Pacman no gana en diseños más grandes porque cada configuración del mapa (grid) es un estado con valores Q que no se relaciona (no generaliza) con estados similares. Es decir, la representación de sus estados no tiene forma de generalizar de forma que represente que toparse con un fantasma es malo en cualquier posiciones. Obviamente, este enfoque no escalará.

Pregunta 6 (3 puntos): Q-Learning aproximado

Implementa un agente Q-learning aproximado que aprenda los pesos de las características de los estados, donde muchos estados pueden compartir las mismas características. Escribe su implementación en `ApproximateQAgent` clase en `qlearningAgents.py`, que es una subclase de `PacmanQAgent`.

Nota: Q-learning aproximado asume la existencia de una función de rasgos f(s, a) sobre pares de acción y estado, lo que produce un vector $f_1(s, a) .. f_i(s, a) .. f_n(s, a)$ de valores de las función de rasgos. Se proveen funciones de rasgos en `featureExtractors.py`. Los vectores de rasgos son `util.Counter` (como un diccionario) objetos que contienen pares de características y valores distintos de cero; todas las características omitidas tienen valor cero.

La función Q aproximada toma la siguiente forma

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

donde cada peso w_i está asociado con una característica particular $f_i(s, a)$. En tu código, debes implementar el vector de peso diccionario mapea funciones de rasgos (que los extractores de características devolverán) y valores de peso. Actualizarás tus vectores de peso de manera similar a como actualizaste los valores Q:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Ten en cuenta que el término *difference* es el mismo que en Q-learning normal, y r es la recompensa experimentada.

De forma predeterminada, `ApproximateQAgent` utiliza `IdentityExtractor`, que asigna una única función a cada par `(state,action)`. Con este extractor de funciones, tu agente Q-learning aproximado debería funcionar de manera idéntica `PacmanQAgent`. Puedes probar esto con el siguiente comando:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Importante: `ApproximateQAgent` es una subclase de `QLearningAgent`y, por lo tanto, comparte varios métodos como `getAction`. Asegúrate de que tus métodos en `QlearningAgent` invoquen `getQValue en` lugar de acceder a los valores Q directamente, de modo que se actualice `getQValue` en tu agente aproximado, y los nuevos valores q aproximados se utilicen para calcular las acciones.

Una vez que estés seguro de que tu aproximador funciona correctamente con los rasgos que se te proporcionan, puedes ejecutar tu agente Q-learning aproximado con el extractor de características personalizadas, que puede aprender a ganar con facilidad:

```
python pacman.py -p ApproximateQAgent -a extractor = SimpleExtractor -x 50 -n 60 -l
mediumGrid
```

Incluso los diseños mucho más grandes no deberían ser un problema para tu `ApproximateQAgent`. (Advertencia : esto puede tardar unos minutos en entrenarse)

```
python pacman.py -p ApproximateQAgent -a extractor = SimpleExtractor -x 50 -n 60 -l mediumClassic
```

Si no tienes errores, tu agente de Q-learning aproximado debería ganar casi siempre con estas funciones simples, incluso con solo 50 juegos de entrenamiento.

Calificación: El autograder ejecutará tu agente Q-learning aproximado y verificará que aprenda los mismos valores Q y pesos de rasgos de la función feature extractor básica aportada en la implementación de referencia cuando se le presenta el mismo conjunto de ejemplos. Para calificar tu implementación, ejecuta el autograder:

```
python autograder.py -q q6
```

¡Felicidades! ¡Tienes un agente Pacman que está aprendiendo!
