IIC3675: MDPs y Monte Carlo

Introducción

En esta tarea resolverás MDPs utilizando programación dinámica y Monte Carlo. Para ello, te recomendamos estudiar la materia del curso y los capítulos 3, 4, y 5 del libro: "*Reinforcement Learning: An introduction.*"

El código base incluye dos grupos de dominios. Los dominios a resolver usando programación dinámica se encuentran en la carpeta `MDPs`. Los dominios para ser resueltos utilizando Monte Carlo están en la carpeta `MonteCarlo`.

Dominios programación dinámica

La carpeta `MDPs` contiene un `Main.py` y la carpeta `Problems`. La carpeta `Problems` tiene tres MDPs ya implementados: `GridProblem`, `CookieProblem` y `GamblerProblem`. Estos dominios heredan de `AbstractProblem`. Por lo mismos, todos ellos implementan los siguientes métodos:

- `states`: Es un property que retorna una lista con todos los estados del MDP.
- `get_initial_state()`: Retorna el estado inicial.
- `get_available_actions(state)`: Retorna una lista con las acciones válidas en el estado `state`.
- `is_terminal(state)`: Retorna verdadero si y solo si `state` es un estado terminal.
- `get_transitions(state, action)`: Retorna una lista con todas las transiciones posibles al ejecutar `action` en `state`. Cada transición es una tupla con tres elementos. El primero es la probabilidad de ocurrir. El segundo es el siguiente estado. El tercero es la recompensa obtenida.
- `show(state)`: Muestra en consola una versión legible de `state` (usado para debuggear).

Estos métodos te permitirán programar algoritmos genéricos de programación dinámica que funcionen en los tres problemas.

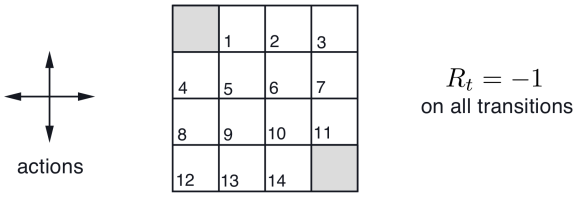
A modo de ejemplo, `Main.py` contiene el siguiente función que permite interactuar manualmente con cualquiera de estos tres MDPs:

```python
def play(problem):
    state = problem.get_initial_state()
    done = False
    total_reward = 0.0
    while not done:
        problem.show(state)
        actions = problem.get_available_actions(state)
        action = get_action_from_user(actions)
        transitions = problem.get_transitions(state, action)
        s_next, reward = sample_transition(transitions)
        done = problem.is_terminal(s_next)
        state = s_next
        total_reward += reward
    print("Done.")
    print(f"Total reward: {total_reward}")
```

Este código comienza desde el estado inicial. Luego entra en un loop hasta que se llegue a un estado terminal. En el loop, primero se muestra el estado actual y se le pide al usuario que seleccione una acción. El estado actual y la acción se utilizan para obtener todas las transiciones posibles al ejecutar la acción en el estado actual. Luego una de esas transiciones es muestreada (según su probabilidad de ocurrir) y así se define el siguiente estado y recompensa recibida. Este procedimiento se repite hasta que el siguiente estado sea un estado terminal.

GridProblem

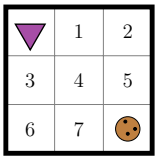
Este dominio es una réplica del *Example 4.1* del libro de Sutton & Barto.



Cada casilla en la grilla tiene un número asociado. En cada momento, el agente se encuentra en una casilla y puede moverse a la izquierda, derecha, arriba o abajo. El estado del MDP es igual al número de la casilla en que se encuentra el agente. Toda transición genera una recompensa de -1. El episodio termina cuando el agente llega a una de las casillas grises. Es posible cambiar las dimensiones de la grilla cambiando el parámetro `grid_size` entregado al constructor del `GridProblem`.

CookieProblem

Este es el dominio que hemos estado utilizando a la largo del curso:



El agente recibe recompensa por comerse la galleta. Luego de comer la galleta una nueva galleta aparece de forma aleatoria. El estado consiste en la posición del agente y la posición de la galleta (ambos codificados con un número que representa a la casilla que los contiene). La recompensa es +1 cuando el agente come la galleta y 0 en otro caso. Al igual que en el `GridProblem`, es posible cambiar el tamaño de la grilla utilizando el parámetro `grid_size` dado al constructor del `CookieProblem`.

GamblerProblem

Este dominio es una réplica del *Example 4.3* del libro de Sutton & Barto. Se trata de un juego de apuestas que consiste en tirar una moneda cargada. El jugador gana cuando la moneda sala cara. El estado es un número entre 0 y 100 que representa cuánto dinero tiene el jugador. En cada turno, el jugador decide cuánto apostar. Si al lanzar la moneda sale cara, entonces el jugador gana el valor apostado. Si sale sello, pierde el valor apostado. El juego continúa hasta que el jugador pierde todo su dinero o llega a 100. La recompensa es siempre 0 salvo cuando se llega al estado 100. Si se llega al estado 100, entonces la recompensa es +1.

Por defecto, la probabilidad de que el lanzamiento sea cara es de 0.4, aunque este valor se puede cambiar en el constructor del `GamblerProblem`. Notar que en este problema el número de acciones válidas dependen del estado. Por ejemplo, si el estado es 50 hay 50 acciones válidas (que son apostar 1, apostar 2, apostar 3, etc). Pero en el estado 25 solo hay 25 acciones válidas.

Dominios Monte Carlo

La carpeta `MonteCarlo` contiene un `Main.py` y la carpeta `Environments`. La carpeta `Environments` tiene dos ambientes implementados: `BlackjackEnv` y `CliffEnv`. Estos ambientes (que heredan de `AbstractEnv`) implementan los siguientes métodos:

- `action_space`: Es un property que retorna una lista con todas los acciones disponibles para el agente.
- `reset()`: Reinicia el ambiente y retorna el estado inicial del problema.

- `step(action)`: Ejecuta la acción `action` en el ambiente y, como resultado, retorna una tupla con 3 elementos: (1) el estado resultante luego de aplicar la acción, (2) la recompensa obtenida producto de ejecutar la acción y (3) un booleano que es verdadero si y solo si se llegó a un estado terminal.
- `show()`: Muestra en consola una versión legible del estado actual.

Estos métodos te permitirán programar agentes genéricos de aprendizaje reforzado que puedan resolver cualquier ambiente que herede de `AbstractEnv`. A modo de ejemplo, el `Main.py` contiene el siguiente código que permite interactuar con un ambiente que herede de `AbstractEnv`:

```
def play(env):
    actions = env.action_space
    state = env.reset()
    total_reward = 0.0
    done = False
    while not done:
        env.show()
        action = get_action_from_user(actions)
        state, reward, done = env.step(action)
        total_reward += reward
    env.show()
    print("Done.")
    print(f"Total reward: {total_reward}")
```

Este código parte obteniendo la lista de acciones disponibles para el agente. Luego reinicia el ambiente y entra en un loop que se repita hasta llegar a un estado terminal. En el loop primero se muestra el estado actual, se elige una acción y se ejecuta. Ejecutar la acción retorna el siguiente estado, recompensa inmediata y el flag `done`. Mientras `done` sea `False` el juego continúa. Cuando `done` es `True` significa que se llegó a un estado terminal y el episodio termina.

BlackjackEnv

Este ambiente implementa el juego Blackjack con un mazo infinito.[1]

Blackjack es un juego de cartas inglesas donde el objetivo es sumar 21 puntos (sin pasarse). Las cartas *jack*, *queen*, *king* valen 10 puntos cada una. El *ace* puede valer 1 o 11 puntos (según lo que más le convenga al jugador). El resto de las cartas valen lo que indica su número.

Al inicio del juego, el jugador saca cartas hasta sumar, al menos, 12 puntos (técnicamente saca dos cartas, pero cuando la suma de puntos es menor a 12 siempre conviene robar otra carta). Por otro lado, el *dealer* roba dos cartas: una puede ser vista por el jugador y la otra no.

En cada turno, el jugador elige entre robar una carta más o quedarse con los puntos que tiene. Si roba una carta y su puntaje sigue siendo menor o igual a 21, vuelve a decidir entre seguir robando o quedarse con los puntos que tiene. Si al robar una carta el puntaje del jugador es mayor a 21 entonces pierde automáticamente.

Cuando el jugador decide dejar de sacar cartas pasa a ser el turno del dealer. El dealer muestra la segunda carta que había sacado. Luego está forzado a seguir la siguiente estrategia (independiente de los puntos que tenga el jugador): Si tiene menos de 17 puntos, saca otra carta; en otro caso, deja de sacar cartas. El jugador gana automáticamente si el delear supera los 21 puntos. Si los puntajes del jugador y del dealer son menores o iguales a 21 entonces gana quien tenga el puntaje más alto.

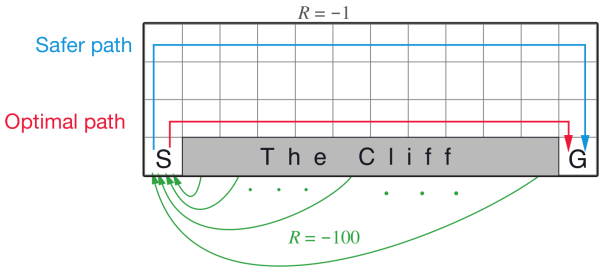
Las recompensas son cero hasta que termina el juego. Al terminar el juego la recompensa es +1 si gana el jugador, -1 si gana el dealer y 0 si es empate.

El estado consiste en una tupla con tres valores: los puntos actuales del jugador, un *bool* que indica si el jugador tiene un *ace* que podría transformar en 1 (de ser necesario) y el valor de la carta que el dealer está mostrando (si es un *ace* se muestra un 1).

[1] Para más detalles, ver *Example 5.1* del libro de Sutton & Barto.

CliffEnv

Este ambiente implementa el dominio *Cliff Walking* descrito en el *Example 6.6* del libro de Sutton & Barto.



Este problema consiste en una grilla donde el agente se puede mover en las 4 direcciones cardinales. El agente parte en el estado `S` y el episodio termina al llegar a `G`. La recompensa en cada paso es -1. Si el agente cae al *cliff*, entonces recibe recompensa -100 y vuelve al estado `S`.

Si luego de 100 mil pasos el agente aún no llega a `G`, el episodio termina. Esta condición fue agregada para forzar a que todo episodio eventualmente termine.

El largo de la grilla puede ser cambiado en el constructor del ambiente.

Instrucciones

Esta tarea puede ser resuelta en pareja o de manera individual.

Debes subir un informe junto con tu código. El código debe incluir un README con instrucciones claras que permitan replicar los experimentos que realizaste. El informe debe incluir gráficos con los experimentos realizados y responder las siguientes preguntas (justifica tus respuestas):

a) **[1 punto]** Demuestra paso a paso que:

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a).$$

b) **[1 punto]** Demuestra paso a paso que:

$$q_\pi(s,a) = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(r, s'|s, a)(r + \gamma v_\pi(s')).$$

c) **[1 punto]** Considera el siguiente MDP determinístico:

- $\mathcal{S} = \{s_0, s_l, s_r\}$, $\mathcal{A} = \{a, a_l, a_r\}$, $\mathcal{A}(s_0) = \{a_l, a_r\}$, $\mathcal{A}(s_l) = \mathcal{A}(s_r) = \{a\}$, $\mathcal{R} = \{0, 1, 2\}$.
- $p(s_l, 1|s_0, a_l) = 1$, $p(s_r, 0|s_0, a_r) = 1$, $p(s_0, 0|s_l, a) = 1$, $p(s_0, 2|s_r, a) = 1$.

En este MDP existen dos políticas deterministicas: $\pi_l(s_0) = a_l$ y $\pi_r(s_0) = a_r$ ¿Cuál de ellas es óptima si se utiliza $\gamma = 0$, $\gamma = 0.9$ y $\gamma = 0.5$? ¿Por qué?

d) **[3 puntos]** Programa el algoritmo de *iterative policy evaluation* (Sección 4.1 del libro de Sutton & Barto). Fija el error máximo (θ) a 0.0000000001. Luego evalúa el desempeño de una política uniformemente aleatoria en los siguientes problemas:

- `GridProblem` donde el tamaño de la grilla es 3, 4, 5, 6, 7, 8, 9 y 10. Utiliza $\gamma = 1.0$.
- `CookieProblem` donde el tamaño de la grilla es 3, 4, 5, 6, 7, 8, 9 y 10. Utiliza $\gamma = 0.99$.

- `GamblerProblem` donde la probabilidad de que sea cara es 0.25, 0.4 y 0.55. Utiliza $\gamma = 1.0$.

En todos estos casos reporta el valor del estado inicial (con 3 decimales) y cuánto tiempo tomó a *iterative policy evaluation* resolver cada problema.

e) **[1 punto]** Respecto a los resultados en la parte (d), existen problemas que toman más tiempo que otros. ¿A qué se debe esto? ¿Cuál es el problema que toma más tiempo en ser resuelto y por qué?

f) **[1 punto]** Respecto a los resultados en la parte (d), si bajamos el valor del factor de descuento, ¿uno esperaría que *iterative policy evaluation* demore más o menos tiempo? ¿por qué?

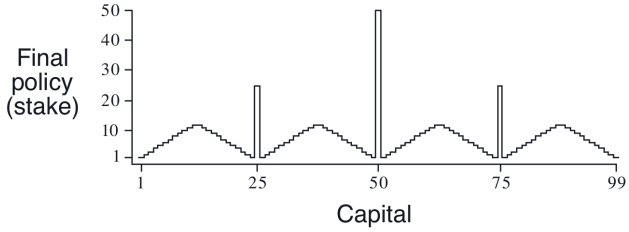
g) **[3 puntos]** Evalúa la política greedy que resulta de los valores calculados en cada uno de los experimentos de la parte (d). Reporta los valores de esta política en el estado inicial (con tres decimales). Además, en cada caso, indica si la política greedy (que fue evaluada) era óptima o no. Justifica tu respuesta.

h) **[3 puntos]** Implementa el algoritmo *value iteration* (Sección 4.4 del libro de Sutton & Barto). Fija el error máximo (θ) en 0.0000000001. Luego utiliza este algoritmo para encontrar los valores óptimos para los siguientes problemas:

- `GridProblem` donde el tamaño de la grilla es 3, 4, 5, 6, 7, 8, 9 y 10. Utiliza $\gamma = 1.0$.
- `CookieProblem` donde el tamaño de la grilla es 3, 4, 5, 6, 7, 8, 9 y 10. Utiliza $\gamma = 0.99$.
- `GamblerProblem` donde la probabilidad de que sea cara es 0.25, 0.4 y 0.55. Utiliza $\gamma = 1.0$.

En todos estos casos reporta el valor óptimo del estado inicial (con 3 decimales) y cuánto tiempo tomó a *value iteration* resolver cada problema.

i) **[3 puntos]** Reporta el set de todas las políticas óptimas para el `GamblerProblem` usando $p_h = 0.25$, $p_h = 0.4$ y $p_h = 0.55$. Para extraer todas las políticas óptimas, considera que existen errores de aproximación en los valores v_* calculados por value iteration. Por lo que debes aproximar los valores de v_* al quinto decimal. Así, si $q(s, a_1) > q(s, a_2)$ cuando ambos valores están redondeados al quinto decimal significa que a_1 efectivamente es mejor que a_2 en el estado s. Para mostrar todas las políticas óptimas usa un gráfico similar a este pero, en vez de utilizar barras, usa puntos indicando los valores de apuestas óptimas:



j) **[6 puntos]** Implementa una variante del algoritmo *On-policy first-visit MC control (for ϵ-soft policies)* – Sección 5.4 del libro de Sutton & Barto. Tu variante debe estimar los Q-values usando *every visit* en vez de *first visit*. Además, en vez de guardar los retornos en listas, debes estimar los Q-values usando una actualización incremental similar a la utilizada en Bandits (Sección 2.4).[2] Luego utiliza tu algoritmo para resolver los siguientes problemas:

- `Blackjack`: Corre Monte Carlo por 10 millones de episodios, 5 veces, usando $\epsilon = 0.01$ y $\gamma = 1.0$. Cada 0.5 millones de episodios, testea la política greedy aprendida hasta el momento. Para testear la política, corre 100 mil simulaciones de la política actual (con $\epsilon = 0$) y reporta el retorno medio obtenido. También reporta el rendimiento en el primer episodio. Luego pon en un gráfico los resultados obtenidos por cada una de las 5 corridas del algoritmo. Es decir, tu gráfico debería tener el rendimiento de la política actual luego de 1 episodio, 0.5 millones de episodios, 1 millón de episodios, y así hasta llegar a los 10 millones de episodios.

[2]Notar que no utilizar updates incrementales hará que el algoritmo sea demasiado lento.

- `Cliff`: Corre Monte Carlo por 200 mil episodios en el ambiente cliff cuando el largo del dominio es 6. Repite el experimento 5 veces usando $\epsilon = 0.1$ y $\gamma = 1.0$. Cada mil episodios de entrenamiento, reporta el rendimiento de la política actual. Para ello, corre una vez la política greedy (con $\epsilon = 0$) y reporta el retorno obtenido desde el estado inicial. Al igual que en el caso del Blackjack, pon los resultados de las 5 corridas de Monte Carlo en un gráfico (que también incluya el rendimiento de la política en el primer episodio).

k) **[2 puntos]** Según Monte Carlo, ¿cuándo uno debería dejar de robar cartas al jugar Blackjack con un mazo infinito? ¿se llega siempre a la misma conclusión en las 5 corridas de Monte Carlo? ¿Por qué?

l) **[2 puntos]** Según Monte Carlo, ¿cuál es el mejor curso de acción en *Cliff*? ¿es este resultado consistente en las 5 corridas de Monte Carlo? Sea sí o no la respuesta anterior ¿por qué ocurre lo que ocurre?

m) **[1 punto]** Considerando los resultados de Monte Carlo tanto en blackjack como en cliff, ¿es Monte Carlo un algoritmo estable?

n) **[1 punto]** ¿Qué ocurre si corremos Monte Carlo en el dominio *Cliff* cuando el largo de la grilla es 12? ¿puede Monte Carlo resolver este problema? ¿Por qué sí o por qué no?