



APRENDIZAJE POR REFUERZO I

Desafío práctico

16Co2024

Índice

Consigna	2
Descripción del problema.....	2
Generación del entorno.....	2
Personalización del entorno.....	7
Q-learning	10
Conclusiones	18
Anexos.....	19
Referencias.....	19

Consigna

El desafío consiste en desarrollar un script en Python que permita encontrar, mediante un algoritmo de aprendizaje por refuerzo, una solución óptima a un problema simple a elección aplicando cualquiera de las técnicas de Aprendizaje por Refuerzo del cursado, es decir, Monte Carlo ES, Monte Carlo IS (ordinario o ponderado), SARSA, Q-Learning o Deep Q-Network.

Puede utilizar bibliotecas como Gymnasium, MiniHack, MuJoCo (u otras del interés del estudiante) para los environments, o bien crear un environment propio.

Descripción del problema

Para el presente desafío se utiliza como base el entorno "**Taxi-v3**" provisto por la biblioteca **Gymnasium**.

Consiste en un entorno en el que un taxi autónomo debe aprender a:

- Recoger a un pasajero en una ubicación específica.
- Llevarlo a su destino.
- Realizar esto moviéndose en una cuadrícula con restricciones.

Generación del entorno

El entorno en su versión por defecto provista por **Gymnasium** tiene las siguientes características:

- El taxi se mueve en una cuadrícula de **5x5 celdas**, con paredes internas que impiden ciertos movimientos.
- Hay **cuatro** ubicaciones fijas que son los posibles **puntos de origen y destino** del pasajero (R = Red, G = Green, Y = Yellow, B = Blue)
- Las **acciones** posibles son:
 - ✓ 0: Move south (down) - Movimiento en dirección sur (hacia abajo)

- ✓ 1: Move north (up) - Movimiento en dirección norte (hacia arriba)
 - ✓ 2: Move east (right) - Movimiento en dirección este (hacia la derecha)
 - ✓ 3: Move west (left) - Movimiento en dirección oeste (hacia la izquierda)
 - ✓ 4: Pickup passenger (Subir a un pasajero al taxi)
 - ✓ 5: Drop off passenger (Dejar al pasajero en su destino)
- Cada **estado** se representa como un número entero entre 0 y 499 (hay **500 posibles estados discretos** ($5 \times 5 \times 5 \times 4$)).
 - ✓ Posición fila del taxi (entre 0 y 4)
 - ✓ Posición columna del taxi (entre 0 y 4)
 - ✓ Ubicación del pasajero (posiciones R, G, Y, B o “en el taxi”)
 - ✓ Destino (posiciones R, G, Y o B)
 - Las **recompensas** ante cada movimiento son:
 - ✓ +20 por dejar al pasajero en el destino correcto.
 - ✓ -1 por cada acción que no sea el Drop off exitoso (es decir, penaliza la demora en llegar a destino).
 - ✓ -10 por intentos inválidos de Pick up o Drop off (penalización mayor por levantar o dejar al pasajero en donde no corresponde).
 - El episodio **termina** cuando:
 - ✓ Termination = 1 (Drop off exitoso - Se deja al pasajero en el destino correcto)
 - ✓ Truncation (time_limit wrapper) = 1 (Máx. steps: 200) - El episodio se trunca por defecto luego de los 200 pasos si no llega a encontrar la solución.
 - El **estado inicial** es una observación aleatoria formada por:
 - ✓ Una posición del taxi (fila y columna).
 - ✓ Una ubicación del pasajero (R, G, Y o B).
 - ✓ Un destino (R, G, Y o B), diferente de la ubicación inicial donde el pasajero toma el taxi. Esto resulta entonces en $(25 \times 4 \times 3) = 300$ estados iniciales posibles.

Para probar inicialmente el funcionamiento del entorno se define una función donde el agente realiza movimientos aleatorios sin aprendizaje alguno.

```
# Crear el entorno Taxi
# Devolver cada frame como un array de imagen

env = gym.make("Taxi-v3", render_mode="rgb_array", max_episode_steps=20)
```

```
# Simular un episodio con acciones aleatorias

def execute_random_episode(env):

    # Reiniciar el entorno a su estado inicial
    # Fijar la semilla aleatoria para que la simulación sea reproducible
    state, info = env.reset(seed=42)

    # Almacenar frames para la animación del resultado
    frames = [env.render()] # Estado inicial
    rewards = ['-']
    actions = ['-']

    terminated = False
    truncated = False
    step_count = 0

    while not (terminated or truncated):

        # Acción aleatoria => POLÍTICA RANDOM A SER REEMPLAZADA LUEGO POR EL
AGENTE
        action = env.action_space.sample()

        state, reward, terminated, truncated, info = env.step(action)

        rewards.append(reward)
        actions.append(action)
        frames.append(env.render())
        step_count += 1

    env.close()

    return frames, rewards, actions
```

```
frames, rewards, actions = execute_random_episode(env)
```

Y para poder visualizar los resultados gráficamente y en forma de video, se crea la función **generateVideo** para animar las imágenes resultantes de cada estado observado.

```
def generateVideo(size, iframe, frames, rewards, actions, interval_seg):
    """
    Función para mostrar los movimientos del agente en video
    """
    # Crear figura
    fig, ax = plt.subplots(figsize=size)
    plt.axis('off')
    im = ax.imshow(iframe)
    title = ax.set_title(f"Step 0 | Action: - | Reward: 0")
    # Función de actualización
    def update(i):
        im.set_array(frames[i])
        title.set_text(f"Step {i+1} | Action: {desc_action[actions[i]]} | Reward: {rewards[i]}")
        return [im, title]
    # Animación
    ani = animation.FuncAnimation(fig, update, frames=len(frames), interval=interval_seg*1000, blit=True)
    plt.close()
    video_html = ani.to_html5_video()
    # Quitar bucle y añadir estilos
    video_html = video_html.replace('loop', '')
    video_html = video_html.replace(
        '<video ',
        '''<video style="
            display: block;
            margin: 20px auto;
            border: 3px solid #444;
            border-radius: 12px;
            box-shadow: 4px 4px 12px rgba(0,0,0,0.3);
            width: 500px;
            max-width: 100%;
            ''')
    )
    return video_html
```

```
resultVideo = generateVideo((5,5), frames[0], frames[1:], rewards[1:],  
actions[1:], 1)  
  
# Mostrar el HTML resultante  
HTML(resultVideo)
```

Ejemplos de los algunos fotogramas generados en el video:



Personalización del entorno

Para generar un entorno alternativo al estándar ofrecido por defecto en la biblioteca, se genera una clase “**CustomTaxiEnv**” que permita extender el entorno original y personalizarlo.




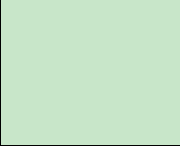
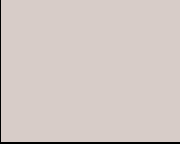
Con la nueva clase custom se puede instanciar un entorno donde se puede modificar por parámetro:

1. El tamaño de la grilla.
2. La cantidad y posición de los destinos posibles.
3. La cantidad y posición de obstáculos.
4. El porcentaje de aleatoriedad si se desea que los movimientos sean estocásticos.
5. La posibilidad de que el pasajero cambie de destino durante el viaje.
6. Log con el detalle de los pasos.

Los métodos considerados son:

- **encode** - Codifica el estado en un número entero como hace el entorno original en función de las posiciones del taxi, del pasajero y de su destino a alcanzar.
- **decode** – Decodifica el entero que representa un estado en las variables originales.
- **reset** - Reinicia el entorno a un estado inicial aleatorio.
- **_get_obs** - Obtiene la observación del estado actual codificado.
- **_apply_stochastic_movement** - En función del porcentaje indicado por parámetro imprime cierta imprevisibilidad a las decisiones del chofer del taxi. El porcentaje indica la probabilidad de tomar la decisión correcta. En cualquier otro caso desviará aleatoriamente a izquierda o derecha con igual probabilidad (similar al efecto de lluvia en algunas versiones del entorno original).
- **step** - Ejecuta la acción y se aplica la recompensa o penalización correspondiente. Esta función tiene en cuenta la posición de los obstáculos y los límites de la grilla (ambos datos ingresados por parámetro) para determinar los movimientos válidos. Incluye también el comportamiento del pasajero indeciso (similar a algunas versiones del entorno original) en donde el pasajero puede cambiar de destino una vez iniciado el viaje.

- **render** - Genera una imagen del estado actual para visualizar el entorno gráficamente.

	Taxi
	Pasajero esperando para tomar el taxi
	Celda celeste - Destino en el que hay que dejar al pasajero.
	Celdas verdes - Otros destinos posibles que el pasajero podría seleccionar.
	Celdas marrones - Representan paredes u obstáculos para el movimiento del taxi.

Ejemplo de instanciación y visualización del entorno personalizado:

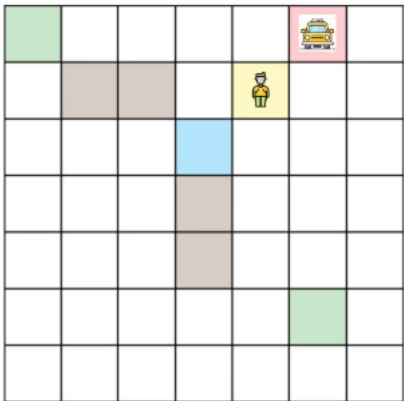
```
env = CustomTaxiEnv(
    grid_size=(7, 7),
    destination_locs=[(0, 0), (2, 3), (1, 4), (5, 5)],
    max_steps=20,
    walls={(1, 1), (1, 2), (3, 3), (4, 3)},
    stochastic=0.8,
    fickle_passenger=True,
    verbose=True
)

frames, rewards, actions = execute_random_episode(env)

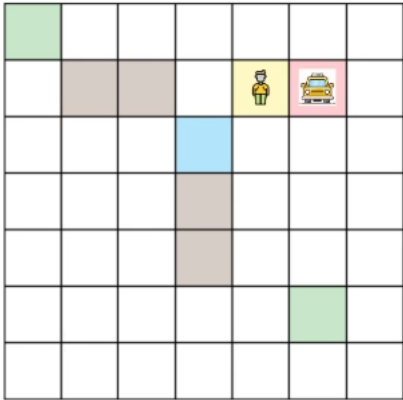
resultVideo = generateVideo((5,5), frames[0], frames[1:], rewards[1:],
actions[1:], 1)

# Mostrar el HTML resultante
HTML(resultVideo)
```

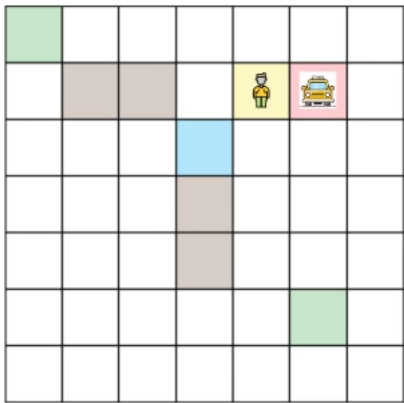
Step 1 | Action: east | Reward: -1



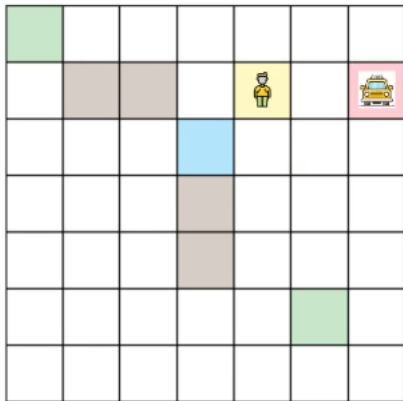
Step 6 | Action: drop | Reward: -10



Step 7 | Action: pick | Reward: -10



Step 20 | Action: north | Reward: -1



Q-learning

Se selecciona **Q-learning** para el aprendizaje del agente por su simplicidad, eficiencia computacional y la suficiencia del algoritmo para entornos con espacios de estado relativamente pequeños y discretos, como en este caso.

```
def q_learning(env, episodes=1000, alpha=0.5, gamma=0.95, epsilon_start=1.0,
epsilon_min=0.05, epsilon_decay=0.995, max_steps=200):

    q_table = np.zeros((env.observation_space.n, env.action_space.n))

    epsilon = epsilon_start
    rewards_all_episodes = []

    for episode in range(episodes):
        state, _ = env.reset()
        done = False
        total_rewards = 0

        for step in range(max_steps):
            # Epsilon-greedy action selection
            if random.uniform(0, 1) < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state])

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            # Actualizar Q-table usando la fórmula de Q-learning
            old_value = q_table[state, action]
            next_max = np.max(q_table[next_state])
            new_value = (1 - alpha) * old_value + alpha * (reward + gamma *
next_max)
            q_table[state, action] = new_value

            state = next_state
            total_rewards += reward

            if done:
                break

        # Decaimiento de epsilon
        epsilon = max(epsilon_min, epsilon * epsilon_decay)
        rewards_all_episodes.append(total_rewards)

        # Mostrar progreso cada 100 episodios
        if (episode + 1) % 100 == 0:
            avg_reward = np.mean(rewards_all_episodes[-100:])
            print(f"Episode {episode+1}/{episodes} - Avg Reward (last 100):
{avg_reward:.2f} - Epsilon: {epsilon:.3f}")

    return q_table, rewards_all_episodes
```

Como valores por defecto de los parámetros se considera:

- **alpha** (Tasa de aprendizaje): 0.5 - Controla qué tan rápido se actualiza el valor de la tabla Q. Es decir, cuánto se "aprende" de cada nuevo dato que recibe el agente. Un valor alto de alpha significa que el agente dará más peso a la nueva información en lugar de la información almacenada en la tabla Q, mientras que un valor bajo significa que el agente dará más peso a lo aprendido previamente.
- **gamma** (Factor de descuento): 0.95 - Este parámetro decide cuánta importancia dar a las recompensas futuras en comparación con las inmediatas. Un valor de gamma cercano a 1.0 indica que el agente valorará mucho las recompensas futuras, mientras que un valor cercano a 0 hace que el agente se enfoque más en las recompensas inmediatas. En este caso, 0.95 sugiere que el agente valora un poco más las recompensas futuras, pero también se interesa por las recompensas inmediatas.
- **epsilon_start**: 1.0 - Es el valor inicial de *epsilon*, que determina el porcentaje de veces que el agente va a explorar el espacio de acciones (en lugar de explotar la mejor acción conocida). 1 significa que al principio el agente será completamente exploratorio, eligiendo acciones al azar. Con el tiempo, *epsilon* disminuirá, lo que hará que el agente se enfoque más en explotar lo aprendido.
- **epsilon_min**: 0.05 - Este es el valor mínimo que *epsilon* puede alcanzar. Después de cada episodio, *epsilon* se irá reduciendo, pero nunca por debajo de este valor. Esto asegura que el agente siga explorando algo, incluso cuando el entrenamiento ya esté más avanzado. El valor 0.05 indica que el agente explorará una pequeña fracción del tiempo, incluso al final del entrenamiento.
- **epsilon_decay**: 0.995 - Este parámetro controla la velocidad con la que *epsilon* disminuye. Un valor de 0.995 significa que *epsilon* se reducirá en un 0.5% en cada episodio. Con esta tasa, el agente comenzará a explotar más y a explorar menos conforme avance el entrenamiento.

Ejemplo del entorno custom instanciado y resultados del entrenamiento:

```
env = CustomTaxiEnv(
    grid_size=(7, 7),
    destination_locs=[(0, 0), (2, 3), (1, 4), (5, 5)],
    max_steps=20,
    walls={(1, 1), (1, 2), (3, 3), (4, 3), (5, 3), (5, 4)},
    stochastic=0,
    fickle_passenger=False,
    verbose=False
)

q_table, rewards_all_episodes = q_learning(env, episodes=15000)
```

```
# Últimos 5 estados de la Q-table
last_5_states = q_table[-5:]

# Convertir la Q-table de los últimos 5 estados en un DataFrame de pandas
df = pd.DataFrame(last_5_states, columns=desc_action)

display(df.style.hide(axis='index'))
```

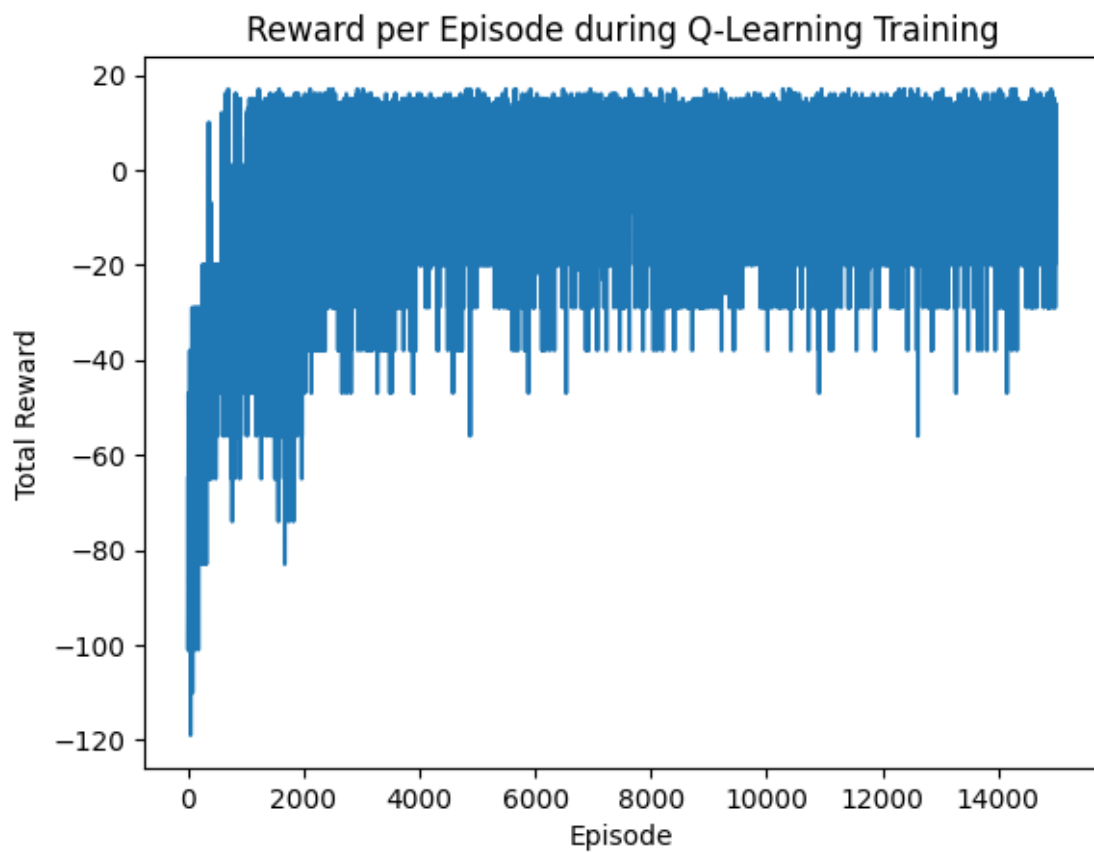
Los valores en cada celda de la tabla son los valores Q de cada acción en ese estado. Los valores Q representan la calidad de la acción en ese estado, es decir, cuán buena es esa acción según el aprendizaje del agente hasta ese momento.

El agente elige la acción con el valor Q más alto en un estado dado, ya que esto indica que es la acción que, en teoría, llevará a la mayor recompensa acumulada a largo plazo.

south	north	east	west	pick	drop
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
-5.112277	100.822196	-4.861572	-5.048319	-5.000000	-5.000000
-3.248168	206.027042	-3.248168	-3.024352	-5.000000	-5.000000
-2.378086	-2.326541	-2.378086	-2.470789	-5.000000	-5.000000
-0.987500	179.156279	-0.500000	-0.500000	-5.000000	-5.000000

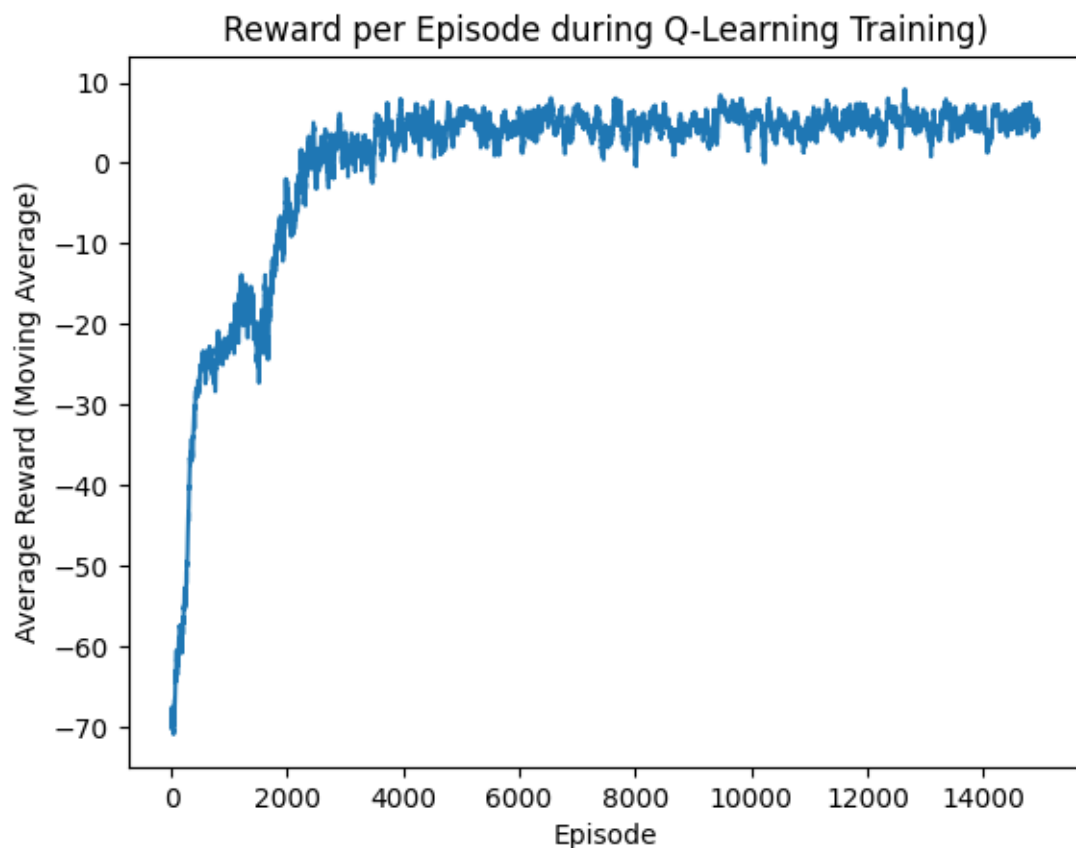
Se grafica la recompensa a lo largo de los episodios para verificar la convergencia de la curva durante el entrenamiento.

```
# Graficar
plt.plot(rewards_all_episodes)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Reward per Episode during Q-Learning Training')
plt.show()
```



Suavizando la curva para una mejor visualización, se observa claramente la convergencia en el aprendizaje.

```
def moving_average(data, window_size):  
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')  
  
window_size = 50  
  
smoothed_rewards = moving_average(rewards_all_episodes, window_size)  
  
plt.plot(smoothed_rewards)  
plt.xlabel('Episode')  
plt.ylabel('Average Reward (Moving Average)')  
plt.title(f'Reward per Episode during Q-Learning Training')  
plt.show()
```



Para poder evaluar y visualizar claramente el aprendizaje del agente, se crea la siguiente función:

```
def evaluate_agent_detailed(env, q_table, episodes=1, max_steps=200):
    all_frames = []
    all_actions = []
    all_rewards = []

    for ep in range(episodes):
        state, _ = env.reset()
        done = False
        frames = [env.render()] # Frame inicial
        actions = [0]
        rewards = [0]
        step_count = 0

        while not done and step_count < max_steps:
            action = np.argmax(q_table[state])
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            frames.append(env.render())
            actions.append(action)
            rewards.append(reward)

            state = next_state
            step_count += 1

        all_frames.append(frames)
        all_actions.append(actions)
        all_rewards.append(rewards)

    return all_frames, all_actions, all_rewards
```

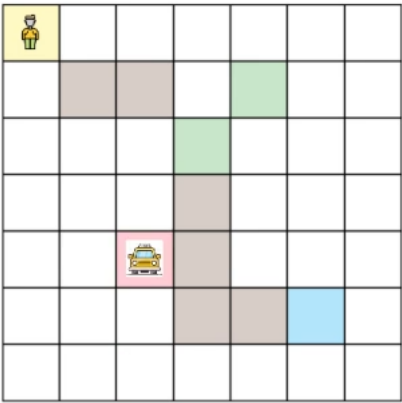
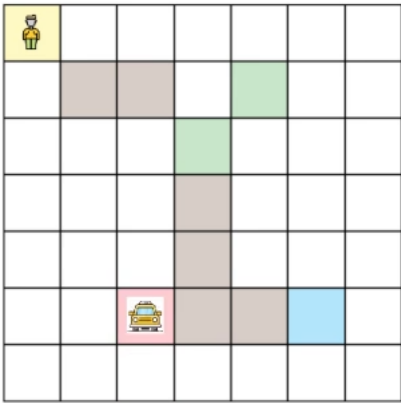
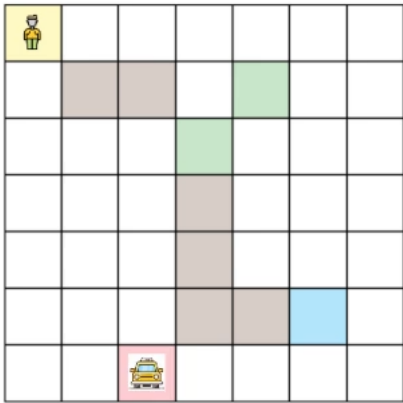
```
# Evaluar guardando frames, acciones y recompensas
all_frames, all_actions, all_rewards = evaluate_agent_detailed(env, q_table,
episodes=1)
```

```
resultVideo = generateVideo((5,5), all_frames[0][0], all_frames[0][1:],
all_rewards[0][1:], all_actions[0][1:], 1)

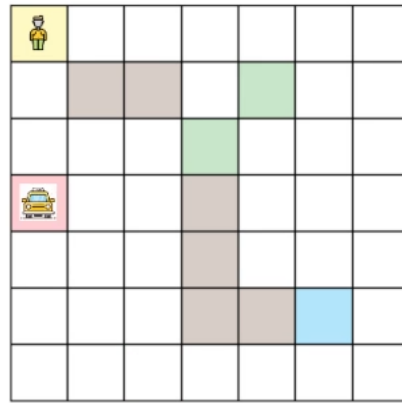
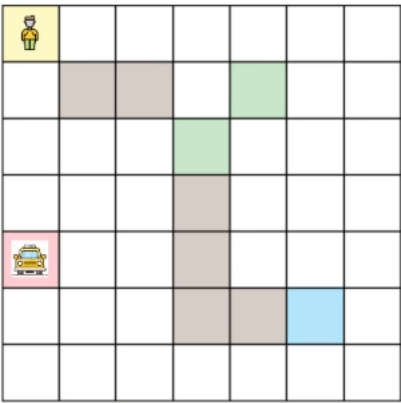
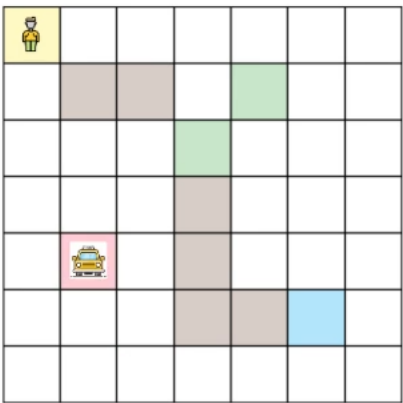
# Mostrar el HTML resultante
HTML(resultVideo)
```


Ejemplo de los fotogramas paso a paso con el agente realizando exitosamente la tarea para la cual fue entrenado:

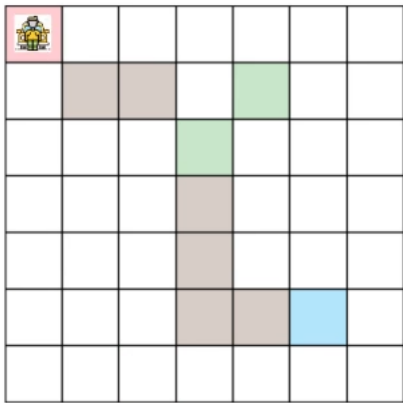
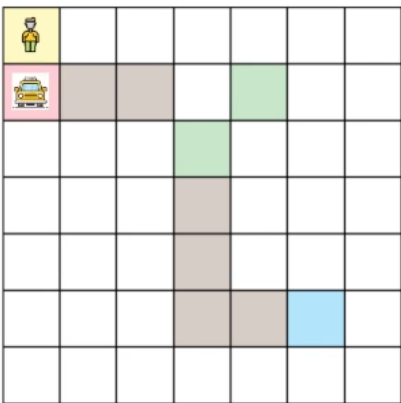
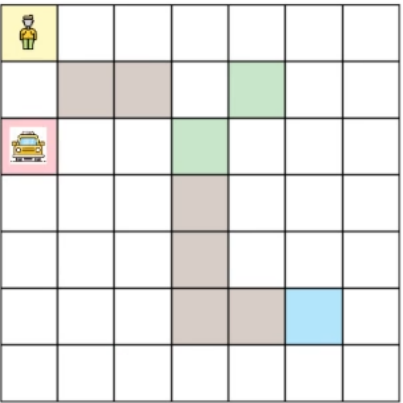
Step 1 | Action: west | Reward: -1 Step 2 | Action: north | Reward: -1 Step 3 | Action: north | Reward: -1



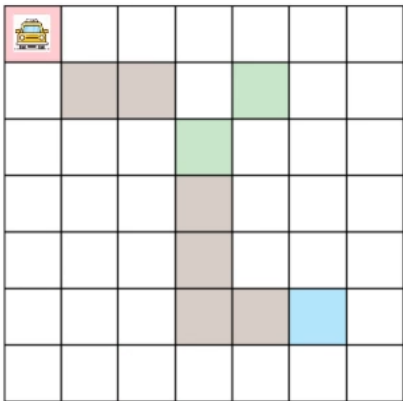
Step 4 | Action: west | Reward: -1 Step 5 | Action: west | Reward: -1 Step 6 | Action: north | Reward: -1



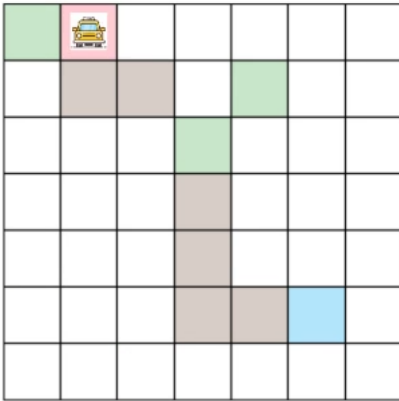
Step 7 | Action: north | Reward: -1 Step 8 | Action: north | Reward: -1 Step 9 | Action: north | Reward: -1



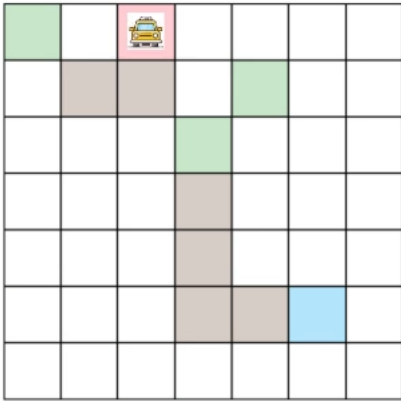
Step 10 | Action: pick | Reward: -1



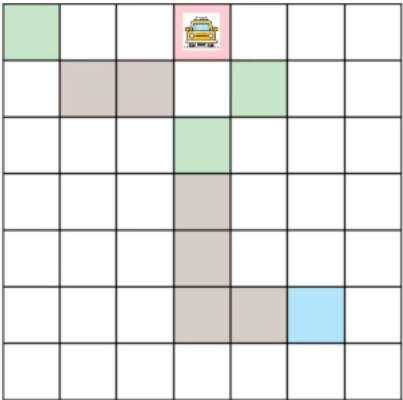
Step 11 | Action: east | Reward: -1



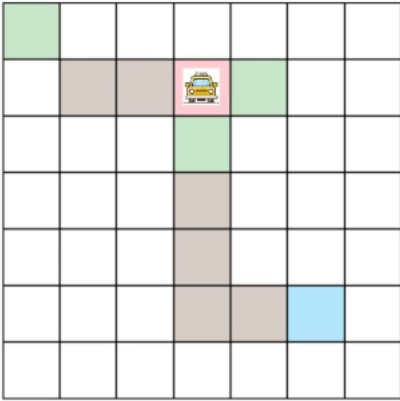
Step 12 | Action: east | Reward: -1



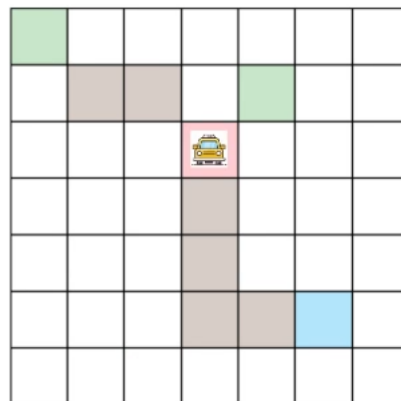
Step 13 | Action: east | Reward: -1



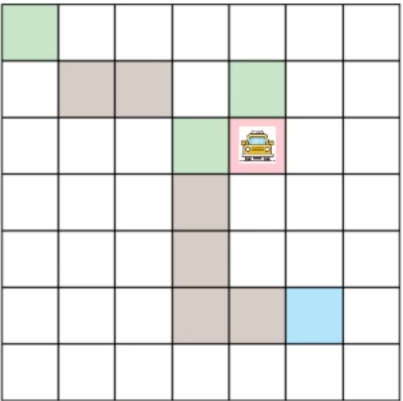
Step 14 | Action: south | Reward: -1



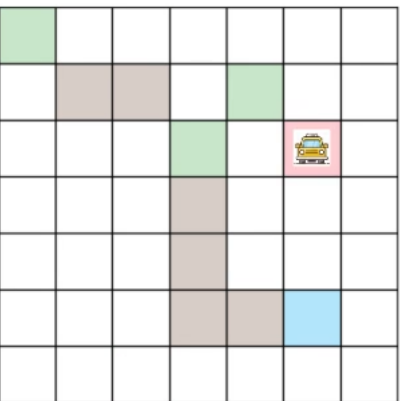
Step 15 | Action: south | Reward: -1



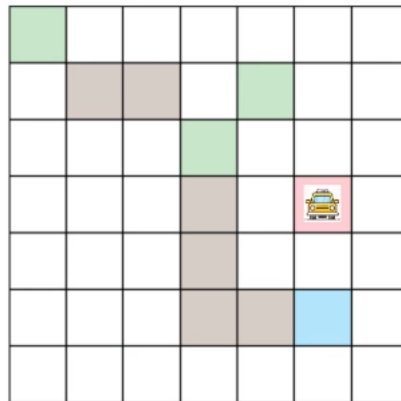
Step 16 | Action: east | Reward: -1



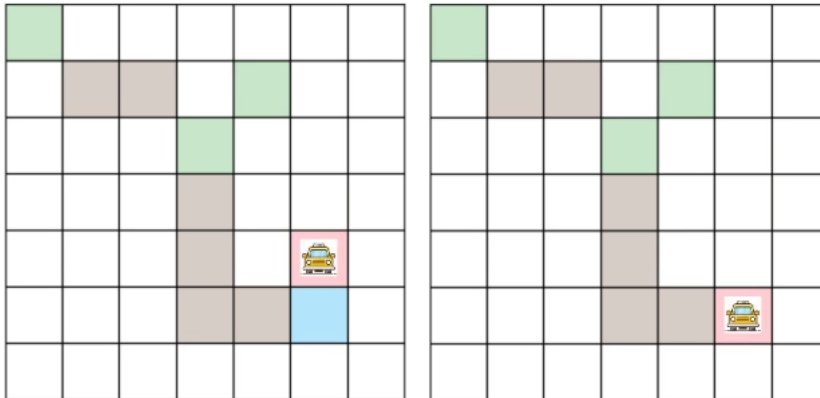
Step 17 | Action: east | Reward: -1



Step 18 | Action: south | Reward: -1



Step 19 | Action: south | Reward: -1 Step 20 | Action: south | Reward: -1 **Drop – Termination = 1**



Conclusiones

En este trabajo se desarrolló e implementó un entorno personalizado basado en el clásico entorno **Taxi** de **Gymnasium**, con el objetivo de explorar y evaluar el comportamiento de un agente de aprendizaje por refuerzo.

Se utilizó el algoritmo **Q-Learning**, técnica de control basada en valores que permite al agente aprender una política óptima a través de la exploración del entorno y la actualización de una tabla Q.

Durante el proceso de entrenamiento, se observó una clara **tendencia de convergencia en la curva de recompensas acumuladas**, lo que indica que el agente logró mejorar su desempeño progresivamente al interactuar con el entorno.

Adicionalmente, se verificó visualmente que el agente ha aprendido a cumplir exitosamente la tarea de recoger y dejar al pasajero en la ubicación correcta, siguiendo rutas eficientes y evitando penalizaciones innecesarias. Estos resultados evidencian que el enfoque aplicado fue efectivo y que el agente logró adquirir una **política de comportamiento adecuada** para resolver el problema propuesto.

En posibles extensiones futuras, sería interesante explorar algoritmos más avanzados, como Deep Q-Learning, o entornos con mayor complejidad, para evaluar la escalabilidad y robustez del enfoque implementado.

Anexos



❖ [Repositorio GitHub](#) con todo el contenido del trabajo.



✓ [Código fuente](#) en formato compatible para ser ejecutado en [Google Colab](#).



✓ [Videos ejemplo](#) de los resultados del agente realizando la tarea aprendida.

Referencias

- Material de clase "Aprendizaje por refuerzo I"
<https://campusposgrado.fi.uba.ar/course/view.php?id=412>
<https://github.com/aeear-uba/ar1/tree/ar1-2025-b3>
- Gymnasium Documentation
<https://gymnasium.farama.org/index.html>

