

AUTORES:

Carlos Ramos Mateos

Adrián Rubio Pintado

▼ GridWorld 2:

GridWorld es un mundo en forma de cuadrícula muy utilizado como entorno de pruebas para técnicas de Aprendizaje por Refuerzo. Dentro de esta cuadrícula hay varios tipos de celdas: iniciales, libres, obstáculos, terminales... ¡y ahora también agujeros de gusano! Los agentes tienen que llegar desde una celda inicial hasta otra terminal evitando los obtáculos y recorriendo una distancia mínima.

Paquetes necesarios para *GridWorld* 2:

```
1 import numpy as np
2 import math
```

Funciones auxiliares para visualizar información:

```
1 def printMap(world):
2     # Visualiza el mapa de GridWorld
3     m = "["
4     for i in range(world.size[0]):
5         for j in range(world.size[1]):
6             if world.map[(i, j)] == 0:
7                 m += " O "
8             elif world.map[(i, j)] == -1:
9                 m += " X "
10            elif world.map[(i, j)] == 1:
11                m += " F "
12            elif world.map[(i, j)] == 2:
13                m += " T "
14            if i == world.size[0] - 1:
15                m += "]\n"
16            else:
17                m += "\n"
18        print(m)
19
20 def printPolicy(world, policy):
21     # Visualiza la política con flechas
22     p = "["
23     for i in range(world.size[0]):
24         for j in range(world.size[1]):
```

```

25     if policy[i][j] == 0:
26         p += " ^ "
27     elif policy[i][j] == 1:
28         p += " V "
29     elif policy[i][j] == 2:
30         p += " < "
31     else:
32         p += " > "
33     if i == world.size[0] - 1:
34         p += "]\n"
35     else:
36         p += "\n"
37 print(p)

```

▼ Clase *World*:

Esta clase almacena la información del mundo:

- *Map*: Matriz con la codificación del mundo con celdas libres (0), obstáculos (-1) y terminales (1)
- *Size*: Vector con el tamaño de la matriz de codificación del mundo (ancho, alto)

Para crear un mundo hay que aportar los siguientes datos:

- Tamaño del mapa (ancho, alto)
- Lista de celdas terminales
- Lista de celdas con obstáculos
- Agujero de gusano

Notas:

- Cuando el agente cae en un obstáculo se queda atrapado para siempre en él
- Cuando el agente entra por un extremo del agujero de gusano sale por el otro extremo

Por ejemplo:

```
w = World((10, 10), [(9, 9)], [(2, 4), (4, 2)], [(0, 2), (9, 7)])
```

Crea un mundo de 10 filas y 10 columnas con un estado terminal (9, 9), dos obstáculos en (2, 4) y (4, 2) y un teletransporte entre (0, 2) y (9, 7).

0	0	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	2	0	1	

```

1 class World:
2
3     def __init__(self, size, terminal, obstacle, hole):
4         # Crea un mundo
5         self.size = size
6         self.map = {}
7         for i in range(size[0]):
8             for j in range(size[1]):
9                 # Estados libres
10                self.map[(i, j)] = 0
11                # Estados terminales
12                for t in terminal:
13                    if i==t[0] and j==t[1]:
14                        self.map[(i, j)] = 1
15                # Estados con obstáculos
16                for o in obstacle:
17                    if i==o[0] and j==o[1]:
18                        self.map[(i, j)] = -1
19                for h in hole:

```

```

20     if i==h[0] and j==h[1]:
21         self.map[(i, j)] = 2

```

Prueba de la clase *World*:

```

1 if __name__ == "__main__":
2     w = World((10, 10), [(9, 9)], [(2, 4), (4, 2)], [(0, 2), (9, 7)])
3     printMap(w)

[ 0  0   T  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  X  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  X  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  T  0  F ]

```

▼ Clase *Agent*:

Esta clase controla el agente que aprende por refuerzo en *GridWorld*.

Para crear un agente se necesitan los siguientes datos:

- *World*: Mundo en el que se desenvuelve el agente.
- *Initial State*: Estado inicial del agente.

Para controlar el agente se usan los siguientes métodos:

- *nextState = move(state, action)*: Mueve el agente del estado *state* a un nuevo estado *nextState* aplicando una acción *action*.
- *reward = reward(nextState)*: Devuelve el refuerzo *reward* que recibe el agente al transicionar al estado *nextState*.
- *nextState, reward = checkAction(state, action)*: Comprueba a qué estado *nextState* y con qué refuerzo *reward* cambia el agente al aplicar la acción *action* en el estado *state*. Este método no cambia el estado interno del agente, por lo que puede usarse para hacer barridos del espacio de estados.
- *nextState, reward = executeAction(action)*: Ejecuta la acción *action* en el estado actual y devuelve el nuevo estado *nextState* y el refuerzo *reward*. Este método cambia el estado interno del agente, por lo que sólo debe usarse cuando se realice un recorrido por el mundo.

Nota: Podéis hacer cambios en el agente (distribución de refuerzos, comportamiento en obstáculos...) buscando mejorar el rendimiento de los algoritmos.

Cambiamos el valor de los refuerzos, añadiendo refuerzo al salirse del mapa, editando el código del agente para ello.

```

1 import sys
2
3 #FLAGS
4 REWARD_OBSTACLE = -100
5 REWARD_TERMINAL = sys.maxsize
6 REWARD_NEWCELL = -1
7 REWARD_OUTOFCOMP = -15

```

Observamos que con refuerzos negativos cada vez que avanza una celda obtenemos mejores resultados: convergen antes los algoritmos en menos épocas. Además hemos añadido una recompensa negativa si intenta salirse del mapa, que de nuevo, facilita la convergencia también.

El reward negativo para los obstáculos lo situamos a un valor tan alto, ya que dado que el agente se queda atrapado en ellos una vez ha caído en ellos, debemos penalizar mucho este comportamiento para que evite los obstáculos a toda costa.

El reward del estado terminal lo hemos puesto a un número muy grande, debido a que en función de la longitud necesaria para resolver laberintos grandes, este refuerzo positivo atraiga de una manera muy fuerte al agente.

```

1 from numpy.lib.type_check import real
2 class Agent:
3
4     def __init__(self, world, initialState):
5         # Crea un agente
6         self.world = world
7         self.state = np.array(initialState)
8
9     def setInitialState(self, initialState):
10        self.state = np.array(initialState)
11
12    def move(self, state, action):
13        # Gestiona las transiciones de estados
14        nextState = state + np.array(action)
15        if nextState[0] < 0:
16            nextState[0] = 0
17        elif nextState[0] >= self.world.size[0]:
18            nextState[0] = self.world.size[0] - 1
19        if nextState[1] < 0:
20            nextState[1] = 0
21        elif nextState[1] >= self.world.size[1]:
22            nextState[1] = self.world.size[1] - 1
23        if self.world.map[(nextState[0], nextState[1])] == 2:
24            aux = nextState
25            for i in range(self.world.size[0]):

```

```
26     for j in range(self.world.size[1]):  
27         if self.world.map[(i, j)] == 2 and (nextState[0] != i and nextState[1] != j):  
28             aux = np.array([i, j])  
29             nextState = aux  
30     return nextState  
31  
32 def reward(self, nextState):  
33     # Gestiona los refuerzos  
34     if self.world.map[(nextState[0], nextState[1])] == -1:  
35         # Refuerzo cuando el agente intenta moverse a un obstáculo  
36         reward = REWARD_OBSTACLE # ** Prueba varios valores **  
37     elif self.world.map[(nextState[0], nextState[1])] == 1:  
38         # Refuerzo cuando el agente se mueve a una celda terminal  
39         reward = REWARD_TERMINAL # ** Prueba varios valores **  
40     else:  
41         # Refuerzo cuando el agente se mueve a una celda libre  
42         reward = REWARD_NEWCELL # ** Prueba varios valores **  
43     return reward  
44  
45 def move_and_reward(self, state, action):  
46  
47  
48     #Estoy en un Obstaculo  
49     if self.world.map[(self.state[0], self.state[1])] == -1:  
50         nextState = self.state  
51         reward = REWARD_OBSTACLE  
52         return nextState,reward  
53  
54  
55     # Gestiona las transiciones de estados  
56     nextState = state + np.array(action)  
57     reward = np.nan  
58  
59     #Fuera del mapa  
60     if nextState[0] < 0:  
61         nextState[0] = 0  
62         reward = REWARD_OUTOFCMAP  
63     elif nextState[0] >= self.world.size[0]:  
64         nextState[0] = self.world.size[0] - 1  
65         reward = REWARD_OUTOFCMAP  
66  
67     if nextState[1] < 0:  
68         nextState[1] = 0  
69         reward = REWARD_OUTOFCMAP  
70     elif nextState[1] >= self.world.size[1]:  
71         nextState[1] = self.world.size[1] - 1  
72         reward = REWARD_OUTOFCMAP  
73  
74     #Agujero de gusano  
75     if self.world.map[(nextState[0], nextState[1])] == 2:  
76         aux = nextState  
77         for i in range(self.world.size[0]):  
78             for j in range(self.world.size[1]):  
79                 if self.world.map[(i, j)] == 2 and (nextState[0] != i and nextState[1] != j):  
80                     aux = np.array([i, j])
```

```

81         nextState = aux
82
83
84
85
86     if(np.isnan(reward) ):
87         reward = self.reward(nextState)
88
89     return nextState,reward
90
91 def checkAction(self, state, action):
92     # Planifica una acción
93     nextState = self.move(state, action)
94     if self.world.map[(state[0], state[1])] == -1:
95         nextState = state
96     reward = self.reward(nextState)
97     return nextState, reward
98
99 def executeAction(self, action):
100    # Planifica y ejecuta una acción
101    #nextState = self.move(self.state, action)
102    nextState,reward = self.move_and_reward(self.state, action)
103    self.state = nextState
104    #reward = self.reward(nextState)
105    return self.state, reward

```

Prueba de la clase Agent:

```

1 if __name__ == "__main__":
2     # Crear el mundo
3     w = World((10, 10), [(9, 9)], [(2, 4), (4, 2)], [(0, 2), (9, 7)])
4     printMap(w)
5     # Crear el agente
6     a = Agent(w, (0, 0))
7     # Mover el agente en la diagonal principal
8     for i in range(1, 5):
9         # Mostrar cada nuevo estado y su recompensa
10        print(a.executeAction((0, 1)))

```

```

[ 0  0  T  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  X  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  X  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  T  F ]

```

```

(array([0, 1]), 0)
(array([9, 7]), 0)
(array([9, 8]), 0)
(array([9, 9]), 9223372036854775807)

```

▼ Trabajo:

En este trabajo vais a implementar los dos algoritmos más comunes de aprendizaje por refuerzo basados en el valor: SARSA y QLearning. Además, vais a probar ambos algoritmos en una serie de escenarios para evaluar su funcionamiento y comparar sus resultados.

Mundos:

Para probar los algoritmos se ofrecen los siguientes mundos en varios tamaños:

- Mundo 1: Laberinto fácil que se puede recorrer en zigzag
- Mundo 2: Mundo con obstáculos aleatorios en el que el teletransporte acorta la distancia desde el inicio hasta el final
- Mundo 3: Mundo con obstáculos aleatorios en el que el teletransporte no reduce la distancia desde el inicio hasta el final
- Mundo 4: Laberinto difícil con caminos correctos y equivocados

Nota: Sentíos libres de utilizar todos o algunos de estos escenarios o directamente crear vuestros propios escenarios.

```
1 if __name__ == "__main__":
2
3     # Mundo 1 pequeño: Laberinto fácil
4     obstacles = []
5     for j in range(0, 4):
6         obstacles.append((j, 1))
7     for j in range(1, 5):
8         obstacles.append((j, 3))
9     w1p = World((5, 5), [(4, 4)], obstacles, [])
10    print("World 1: ")
11    printMap(w1p)
12
13    # Mundo 1 mediano: Laberinto fácil
14    obstacles = []
15    for i in [1, 5]:
16        for j in range(0, 8):
17            obstacles.append((j, i))
18    for i in [3, 7]:
19        for j in range(1, 9):
20            obstacles.append((j, i))
21    w1m = World((9, 9), [(8, 8)], obstacles, [])
22    print("World 1: ")
23    printMap(w1m)
24
25    # Mundo 1 grande: Laberinto fácil
26    obstacles = []
27    for i in [1, 5, 9, 13, 17]:
28        for j in range(0, 20):
29            obstacles.append((j, i))
```

```
30  for i in [3, 7, 11, 15, 19]:  
31      for j in range(1, 21):  
32          obstacles.append((j, i))  
33  w1g = World((21, 21), [(20, 20)], obstacles, [])  
34  print("World 1: ")  
35  printMap(w1g)  
36  
37  # Mundo 2 pequeño: Obstáculos aleatorios, teletransporte útil  
38  obstacles = []  
39  for i in range(3):  
40      obstacles.append((np.random.randint(1, 4), np.random.randint(1, 4)))  
41  w2p = World((5, 5), [(4, 4)], obstacles, [(2, 0), (4, 2)])  
42  print("World 2: ")  
43  printMap(w2p)  
44  
45  # Mundo 2 mediano: Obstáculos aleatorios, teletransporte útil  
46  obstacles = []  
47  for i in range(10):  
48      obstacles.append((np.random.randint(1, 9), np.random.randint(1, 9)))  
49  w2m = World((10, 10), [(9, 9)], obstacles, [(3, 1), (8, 6)])  
50  print("World 2: ")  
51  printMap(w2m)  
52  
53  # Mundo 2 grande: Obstáculos aleatorios, teletransporte útil  
54  obstacles = []  
55  for i in range(50):  
56      obstacles.append((np.random.randint(1, 19), np.random.randint(1, 19)))  
57  w2g = World((21, 21), [(20, 20)], obstacles, [(6, 2), (18, 14)])  
58  print("World 2: ")  
59  printMap(w2g)  
60  
61  # Mundo 3 pequeño: Obstáculos aleatorios, teletransporte inútil  
62  obstacles = []  
63  for i in range(3):  
64      obstacles.append((np.random.randint(1, 4), np.random.randint(1, 4)))  
65  w3p = World((5, 5), [(4, 4)], obstacles, [(4, 0), (0, 4)])  
66  print("World 3: ")  
67  printMap(w3p)  
68  
69  # Mundo 3 mediano: Obstáculos aleatorios, teletransporte inútil  
70  obstacles = []  
71  for i in range(10):  
72      obstacles.append((np.random.randint(1, 9), np.random.randint(1, 9)))  
73  w3m = World((10, 10), [(9, 9)], obstacles, [(8, 1), (1, 8)])  
74  print("World 3: ")  
75  printMap(w3m)  
76  
77  # Mundo 3 grande: Obstáculos aleatorios, teletransporte inútil  
78  obstacles = []  
79  for i in range(50):  
80      obstacles.append((np.random.randint(1, 19), np.random.randint(1, 19)))  
81  w3g = World((21, 21), [(20, 20)], obstacles, [(18, 2), (2, 18)])  
82  print("World 3: ")  
83  printMap(w3g)  
84
```

```

85 # Mundo 4: Laberinto difícil
86 obstacles = [(0,1),(0,3),(0,9),(0,15),(0,16),(0,17),(0,19),
87             (1,1),(1,3),(1,4),(1,5),(1,6),(1,7),(1,9),(1,10),(1,11),(1,12),(1,13),(1,
88             (2,1),(2,9),(2,13),(2,15),(2,16),(2,17),(2,19),
89             (3,1),(3,3),(3,5),(3,7),(3,9),(3,11),(3,16),(3,19),
90             (4,3),(4,5),(4,7),(4,8),(4,9),(4,10),(4,11),(4,12),(4,13),(4,14),(4,16),
91             (5,0),(5,1),(5,2),(5,3),(5,5),(5,9),(5,16),
92             (6,5),(6,6),(6,7),(6,9),(6,10),(6,11),(6,12),(6,13),(6,14),(6,16),(6,17)
93             (7,0),(7,1),(7,2),(7,3),(7,5),(7,7),(7,9),(7,19),
94             (8,3),(8,7),(8,8),(8,9),(8,12),(8,13),(8,14),(8,15),(8,16),(8,17),(8,18)
95             (9,1),(9,3),(9,5),(9,7),(9,11),(9,12),(9,19),(9,20),
96             (10,1),(10,3),(10,5),(10,6),(10,7),(10,9),(10,11),(10,14),(10,15),(10,16)
97             (11,1),(11,3),(11,5),(11,9),(11,11),(11,13),(11,14),(11,17),(11,18),(11,
98             (12,1),(12,5),(12,6),(12,8),(12,9),(12,11),(12,13),(12,19),
99             (13,1),(13,2),(13,3),(13,4),(13,5),(13,8),(13,15),(13,16),(13,17),(13,19)
100            (14,4),(14,7),(14,8),(14,10),(14,12),(14,13),(14,15),(14,19),
101            (15,0),(15,1),(15,2),(15,6),(15,7),(15,10),(15,13),(15,14),(15,15),(15,16)
102            (16,2),(16,3),(16,5),(16,6),(16,7),(16,8),(16,9),(16,10),(16,11),(16,15)
103            (17,0),(17,3),(17,5),(17,9),(17,13),(17,14),(17,15),(17,17),(17,19),
104            (18,0),(18,1),(18,5),(18,6),(18,7),(18,9),(18,10),(18,11),(18,15),(18,19)
105            (19,1),(19,2),(19,4),(19,5),(19,11),(19,13),(19,14),(19,15),(19,16),(19,
106            (20,7),(20,8),(20,9),(20,11),(20,19)]
107 print("World 4: ")
108 w4 = World((21, 21), [(20, 20)], obstacles, [])
109 printMap(w4)

```

World 3:

```
[ 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 X 0 0 0 0 0 T 0
 0 X 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 X 0 0 0 X 0
 0 0 0 X 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0
 0 X 0 0 0 0 0 0 0 0 0
 0 0 0 0 X 0 0 0 0 0 0
 0 T 0 0 X 0 X 0 X 0
 0 0 0 0 0 0 0 0 0 F ]
```

World 3:

```
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X 0 0 0
 0 0 X 0 0 X 0 0 X 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 T 0 0
 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 X 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 X X X 0 0 0 X 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 X 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 X 0 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 X X 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 X X X 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 X 0 X X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 T 0 0 X 0 0 X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
```

World 4:

▼ SARSA:

SARSA (State-Action-Reward-State-Action) es un método basado en el valor que permite resolver problemas de aprendizaje por refuerzo. Al igual que el resto de métodos basados en el valor, SARSA calcula de forma iterativa la función de valor $Q(S, A)$ y, a partir de ella, determina la política óptima π .

SARSA recibe su nombre de las cinco variables implicadas en su función de actualización: el estado actual (S_t), la acción actual (A_t), el refuerzo actual (R_t), el siguiente estado (S_{t+1}) y la siguiente acción (A_{t+1}). Esta función de actualización tiene la siguiente forma:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Nota: α es la longitud del episodio y γ el factor de descuento.

El algoritmo SARSA sigue el siguiente esquema:

1. Inicializar $Q(S, A)$ para todos los estados y acciones
 2. **Bucle** (repetir 3 – 9 hasta la convergencia):
 3. Inicializar S_t
 4. Elegir A_t en S_t siguiendo la política derivada de $Q(S, A)$
 5. **Bucle** (repetir 6 – 9 hasta que S_t sea terminal):
 6. Tomar la acción A_t en S_t y observar R_t y S_{t+1}
 7. Elegir A_{t+1} en S_{t+1} siguiendo la política derivada de $Q(S, A)$
 8. Actualizar el valor $Q(S_t, A_t)$ con la función de actualización

9. Tomar S_{t+1} y A_{t+1} como los nuevos S_t y A_t

El algoritmo SARSA utiliza un parámetro $\epsilon \in (0, 1)$ para buscar un equilibrio entre exploración y explotación. A la hora de elegir A_t en S_t , si un número aleatorio es menor que ϵ , el algoritmo tomará una acción aleatoria; mientras que si ese número aleatorio es mayor que ϵ , el algoritmo tomará la mejor acción conocida.

Ejercicio 1:

Implementad el algoritmo SARSA para el agente y entorno definidos anteriormente.

```

1 ACTIONS = [(0,1), (0,-1), (1,0), (-1,0)]
2 DICCIONARIO = {(0,1):'Derecha', (0,-1): 'Izquierda', (1,0) :'Abajo', (-1,0):'Arriba'}

1 import random
2
3 def choose_max_Q(S, Q):
4
5
6     actions_S = [ (A, Q[(S,A)]) for A in ACTIONS ]
7     max_value = max(actions_S, key = lambda x: x[1])[1]
8     possible_actions = [A[0] for A in actions_S if A[1] == max_value]
9     return random.choice(possible_actions)
10
11
12 def elige_accion(epsilon,Q, S):
13     rand_num = np.random.rand(1)[0]
14     if(rand_num <= epsilon):
15         return random.choice(ACTIONS)
16     else:
17         return choose_max_Q(S, Q)
18
19 def sarsa(agent,init_state = (0,0), alpha = 0.5,gamma = 0.9 , epsilon = 0.1 , max_iterations = 1000):
20
21     assert alpha >= 0 and alpha <= 1, 'Alpha debe de estar entre 0 y 1'
22     assert gamma >= 0 and gamma <= 1, 'Gamma debe de estar entre 0 y 1'
23
24     size = agent.world.size
25
26     #Incializando valores
27     Q = {}
28     for i in range(size[0]):
29         for j in range(size[1]):
30             Q.update( { ((i,j),a):(0 if agent.world.map[(i,j)] == 1 else -1) for a in ACTIONS})
31             #Q.update( { ((i,j),a):0. for a in ACTIONS})#estados terminales nulos
32
33
34     sum_rewards_per_episode = []
35     #Eleccion de acción de acuerdo a Q(S,A)
36     for i in range(max_iterations): #bucle convergencia
37         S = init_state
38         A = elige_accion(epsilon,Q, S)

```

```

39     agent.setInitialState(init_state)
40     if(debug): print('Iteración:', i+1)
41     sum_rewards_per_episode.append(0)
42     #counter_hole = 0
43
44     while(agent.world.map[S] != 1): #S terminal
45         S_prima,R = agent.executeAction(A)
46         S_prima = (S_prima[0], S_prima[1])
47         A_prima = elige_accion(epsilon,Q, S_prima)
48         sum_rewards_per_episode[i] = sum_rewards_per_episode[i] + R
49         #Update model
50         Q[(S,A)] = Q[(S,A)] + alpha*( R + ( gamma*Q[(S_prima,A_prima)] ) - Q[(S,A)] )
51
52         # indica que ha caido en obstaculo, y no puede moverse mas
53         if (agent.world.map[S_prima] == -1):
54             break
55
56         S = S_prima
57         A = A_prima
58
59
60     return Q,sum_rewards_per_episode
61
62

```

Definimos una función para movernos:

Siguiendo la política basada en valores obtenida con SARSA/Q-Learning.

```

1
2 def move_with_value_function(agent, Q, init_state = (0,0), verbose = True, max_steps =
3     ''
4     Returns:
5         number of steps walked
6     ''
7
8     S = init_state
9     agent.setInitialState(S)
10
11    A = choose_max_Q(S, Q)
12    if(verbose): print('Initial S:', S, '--(A:', A,DICCIONARIO[A],') -->')
13    #print('Init action', A)
14
15    # Mostrar cada nuevo estado y su recompensa
16    i = 0
17    while(agent.world.map[S] != 1): #Hasta estado terminal, sigue moviendose
18        S, R = agent.executeAction(A)
19        S = (S[0], S[1])
20        A = choose_max_Q(S, Q)
21        if(verbose): print('Now in S:', S, '--(A:', A,DICCIONARIO[A],') -->')
22        if( agent.world.map[(S[0], S[1])] == -1):
23            print('\tTRAPPED IN OBSTACLE!!!')
24            break;
25        i = i+1

```

```

26     if(i>max_steps):break
27 print('END: ', i , 'steps.')
28 return i

```

Ejecutamos un minitest:

```

1 w = World((10, 10), [(9, 9)], [(2, 4), (4, 2)], [(0, 2), (9, 7)])
2 a = Agent(w, (0, 0))
3 #Q_test = sarsa(a,debug = True)
4 Q_test = sarsa(a)

1 a.setInitialState((0, 0))
2 Q_final = move_with_value_function(a ,Q_test)

Initial S: (0, 0) --(A: (0, 1) Derecha ) -->
Now in S: (0, 1) --(A: (0, 1) Derecha ) -->
Now in S: (9, 7) --(A: (0, 1) Derecha ) -->
Now in S: (9, 8) --(A: (0, 1) Derecha ) -->
Now in S: (9, 9) --(A: (0, 1) Derecha ) -->
END

```

▼ Q-Learning:

Q-Learning es el método más conocido para resolver problemas de aprendizaje por refuerzo mediante un esquema basado en el valor. Este algoritmo recibe su nombre directamente de $Q(S, A)$, la función de valor que va actualizando a lo largo de su ejecución. *Q-Learning* es muy parecido a SARSA, pero tiene una función de actualización diferente:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

En este caso, la acción A_{t+1} en S_{t+1} se toma buscando el máximo valor, en lugar de poder elegir entre exploración o explotación.

El algoritmo *Q-Learning* sigue el siguiente esquema:

1. Inicializar $Q(S, A)$ para todos los estados y acciones
2. **Bucle** (repetir 3 – 8 hasta la convergencia):
3. Inicializar S_t
4. **Bucle** (repetir 6 – 8 hasta que S_t sea terminal):
5. Elegir A_t en S_t siguiendo la política derivada de $Q(S, A)$
6. Tomar la acción A_t en S_t y observar R_t y S_{t+1}
7. Actualizar el valor $Q(S_t, A_t)$ con la función de actualización
8. Tomar S_{t+1} como el nuevo S_t

Ejercicio 2:

Implementad el algoritmo Q-Learning para el agente y entorno definidos anteriormente.

```

1 import random
2
3
4
5 def choose_max_Q(S, Q):
6
7
8     actions_S = [ ( A, Q[(S,A)]) for A in ACTIONS ]
9     max_value = max(actions_S, key = lambda x: x[1])[1]
10    possible_actions = [A[0] for A in actions_S if A[1] == max_value]
11    return random.choice(possible_actions)
12
13
14
15 def elige_accion(epsilon,Q, S):
16    rand_num = np.random.rand(1)[0]
17    if(rand_num <= epsilon):
18        return random.choice(ACTIONS)
19    else:
20        return choose_max_Q(S, Q)
21
22 def qlearning(agent,init_state = (0,0), alpha = 0.5,gamma = 0.9,   epsilon = 0.1 , max_
23
24     assert alpha >= 0 and alpha <= 1, 'Alpha debe de estar entre 0 y 1'
25     assert gamma >= 0 and gamma <= 1, 'Gamma debe de estar entre 0 y 1'
26
27     size = agent.world.size
28
29     #Incializando valores
30     Q = {}
31     for i in range(size[0]):
32         for j in range(size[1]):
33             Q.update( { ((i,j),a):(0 if agent.world.map[(i,j)] == 1 else -1) for a in ACTION
34             #Q.update( { ((i,j),a):0. for a in ACTIONS})#estados terminales nulos
35
36
37     sum_rewards_per_episode = []
38     #Elección de acción de acuerdo a Q(S,A)
39     for i in range(max_iterations): #bucle convergencia
40         S = init_state
41         agent.setInitialState(init_state)
42         sum_rewards_per_episode.append(0)
43         if(debug): print('Iteración:', i+1)
44
45
46         while(agent.world.map[S] != 1): #S terminal
47             A = elige_accion(epsilon,Q, S)
48             S_prima,R = agent.executeAction(A)
49             S_prima = (S_prima[0], S_prima[1])
50             sum_rewards_per_episode[i] = sum_rewards_per_episode[i] + R
51
52
53         #Update model

```

```

54     Q[(S,A)] = Q[(S,A)] + alpha*( R + ( gamma*Q[(S_prima,choose_max_Q(S_prima, Q))] )
55
56     #if(R == REWARD_OBSTACLE):#R = -1 indica que ha caido en obstaculo, y no puede mc
57     if (agent.world.map[S_prima] == -1):
58         break
59
60     S = S_prima
61
62
63 return Q,sum_rewards_per_episode
64

```

▼ Comparativa

Hacemos una comparativa con una selección de los mundos para ver si ambos algoritmos son capaces de resolver los laberintos.

Creamos una función para **comparar los rendimientos como la suma del numero de rewards por episodio frente al numero de episodios**.

Incluimos una **opción de promediar cada mean_each_n episodios**, para evitar picos dado el valor tan alto de refuerzos que hemos elegido.

Incluimos también la opción de graficar lo anterior **de manera acumulada**, de modo que vaya sumando las rewards de cada episodio de manera acumulada, con el objetivo de **estudiar la convergencia** de los algoritmos para los ejemplos.

```

1 import plotly.graph_objects as go
2
3 def plot_performance(rewards_sarsa, rewards_q1, size = None, mean_each_n = None,acumula
4
5     if(size is None):
6         r_sarsa = rewards_sarsa
7         r_q1 = rewards_q1
8     else:
9         r_sarsa = rewards_sarsa[0:size]
10        r_q1 = rewards_q1[0:size]
11
12    if(mean_each_n is not None): #Media cada n elementos, para evitar picos
13        r_sarsa = np.array(r_sarsa)
14        r_q1 = np.array(r_q1)
15
16        r_sarsa = np.average(r_sarsa.reshape(-1, mean_each_n), axis=1).tolist()
17        r_q1 = np.average(r_q1.reshape(-1, mean_each_n), axis=1).tolist()
18
19        r_sarsa = [i/mean_each_n for i in r_sarsa ]
20        r_q1 = [i/mean_each_n for i in r_sarsa ]
21        title = 'Sum of reward for each episode(averaged with each ' + str(mean_each_n) + '
22        x_title = 'Number of episodes(averaged with each ' + str(mean_each_n) + ' episodes)'
23    else:
24        title = 'Sum of reward for each episode'
25        x_title = 'Number of episodes'

```

```

26
27
28 if(accumulated):
29     title = title + ' ACUMULATED'
30     r_sarsa = np.cumsum(r_sarsa).tolist()
31     r_q1 = np.cumsum(r_q1).tolist()
32
33 fig = go.Figure()
34 fig.add_trace(go.Scatter(x=[i for i in range(len(r_sarsa))], y=r_sarsa,
35                         mode='lines+markers',
36                         name='SARSA'))
37 fig.add_trace(go.Scatter(x=[i for i in range(len(r_q1))], y=r_q1,
38                         mode='lines+markers',
39                         name='Q-Learning'))
40
41
42
43 fig.update_layout(
44     title=title,
45     xaxis_title=x_title,
46     yaxis_title="Sum of rewards during episode",
47     legend_title="Legend Title",
48     font=dict(
49         family="Courier New, monospace",
50         color="RebeccaPurple"
51     )
52 )
53
54 fig.show()

```

Procedemos ahora a resolver varios laberintos, escogiendo aquellos que nos parecían más relevantes y representativos para estudiar los algoritmos.

▼ World1p

```
1 printMap(w1p)
```

```
[ 0 X 0 0 0
 0 X 0 X 0
 0 X 0 X 0
 0 X 0 X 0
 0 0 0 X F ]
```

▼ SARSA

```

1 #SARSA
2 agent_w1p_sarsa = Agent(w1p, (0, 0))
3 Q_w1p_sarsa, rewards_w1p_sarsa = sarsa(agent_w1p_sarsa, init_state = (0,0), alpha = 0.5,
4 move_with_value_function(agent_w1p_sarsa, Q_w1p_sarsa, max_steps = 50)

```

```

Initial S: (0, 0) --(A: (1, 0) Abajo ) -->
Now in S: (1, 0) --(A: (1, 0) Abajo ) -->
Now in S: (2, 0) --(A: (1, 0) Abajo ) -->
Now in S: (3, 0) --(A: (1, 0) Abajo ) -->
Now in S: (4, 0) --(A: (0, 1) Derecha ) -->
Now in S: (4, 1) --(A: (0, 1) Derecha ) -->
Now in S: (4, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (3, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (2, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (1, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (0, 2) --(A: (0, 1) Derecha ) -->
Now in S: (0, 3) --(A: (0, 1) Derecha ) -->
Now in S: (0, 4) --(A: (1, 0) Abajo ) -->
Now in S: (1, 4) --(A: (1, 0) Abajo ) -->
Now in S: (2, 4) --(A: (1, 0) Abajo ) -->
Now in S: (3, 4) --(A: (1, 0) Abajo ) -->
Now in S: (4, 4) --(A: (-1, 0) Arriba ) -->
END: 16 steps.
16

```

▼ Q-LEARNING

```

1 #Q-LEARNING
2 agent_w1p_ql = Agent(w1p, (0, 0))
3 Q_w1p_ql, rewards_w1p_ql = qlearning(agent_w1p_ql, init_state = (0,0), alpha = 0.5, gamma = 0.9, max_steps = 50)
4 move_with_value_function(agent_w1p_ql,Q_w1p_ql, max_steps = 50)

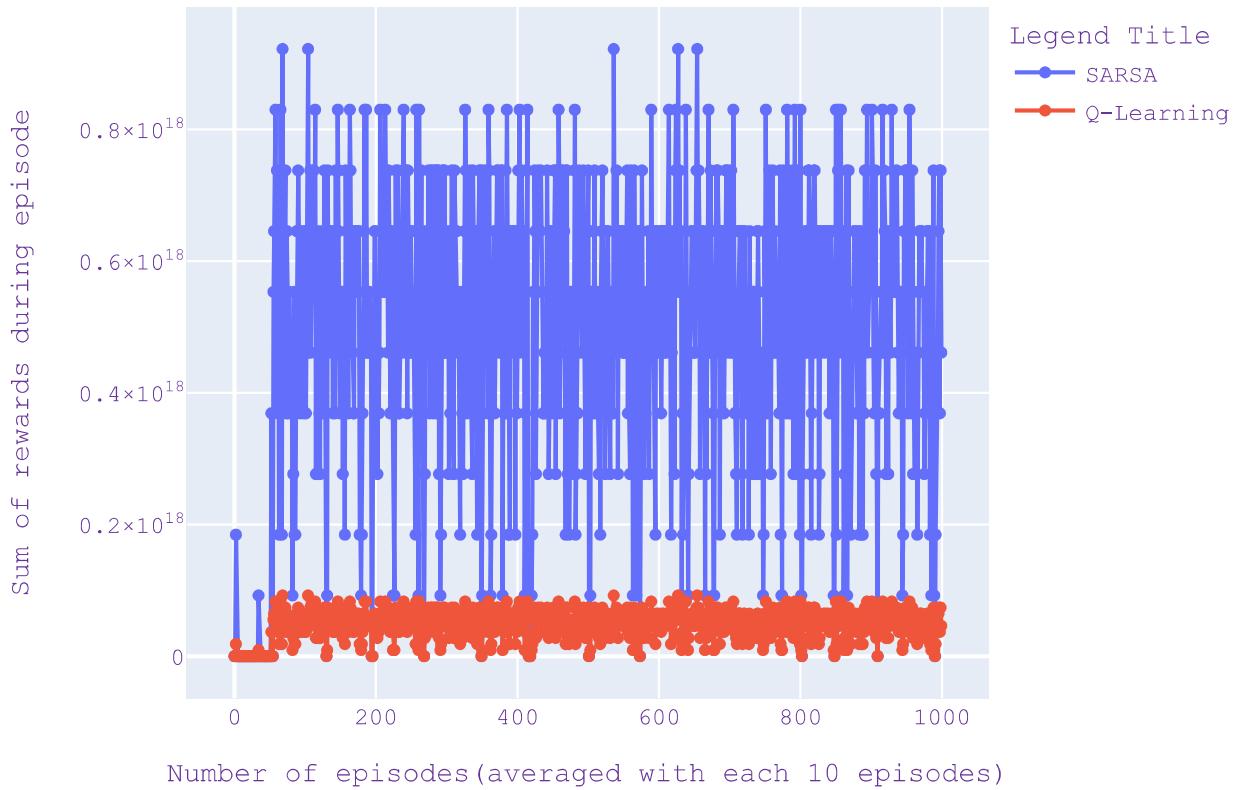
Initial S: (0, 0) --(A: (1, 0) Abajo ) -->
Now in S: (1, 0) --(A: (1, 0) Abajo ) -->
Now in S: (2, 0) --(A: (1, 0) Abajo ) -->
Now in S: (3, 0) --(A: (1, 0) Abajo ) -->
Now in S: (4, 0) --(A: (0, 1) Derecha ) -->
Now in S: (4, 1) --(A: (0, 1) Derecha ) -->
Now in S: (4, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (3, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (2, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (1, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (0, 2) --(A: (0, 1) Derecha ) -->
Now in S: (0, 3) --(A: (0, 1) Derecha ) -->
Now in S: (0, 4) --(A: (1, 0) Abajo ) -->
Now in S: (1, 4) --(A: (1, 0) Abajo ) -->
Now in S: (2, 4) --(A: (1, 0) Abajo ) -->
Now in S: (3, 4) --(A: (1, 0) Abajo ) -->
Now in S: (4, 4) --(A: (0, -1) Izquierda ) -->
END: 16 steps.
16

```

▼ Rendimiento

```
1 plot_performance(rewards_w1p_sarsa, rewards_w1p_ql, size = 10000, mean_each_n = 10)
```

Sum of reward for each episode (averaged with each 10 episodes)



Graficamos ahora la misma gráfica pero de manera acumulada:

```
1 plot_performance(rewards_w1p_sarsa, rewards_w1p_ql, size = 10000, mean_each_n = 10, acumul)
```

Sum of reward for each episode (averaged with each 10 episodes)

▼ World1m

```
1 printMap(w1m)

[ 0 X 0 0 0 X 0 0 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 X 0 X 0 X 0 X 0
 0 0 0 X 0 0 0 X F ]
```

▼ SARSA

```
1 agent_w1m_sarsa = Agent(w1m, (0, 0))
2 Q_w1m_sarsa, rewards_w1m_sarsa = sarsa(agent_w1m_sarsa, init_state = (0,0), alpha = 0.5,
3 move_with_value_function(agent_w1m_sarsa, Q_w1m_sarsa, max_steps = 100)
```

```
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (1, 0) Abajo ) -->
```

101

Vemos como SARSA no consigue resolver dicho laberinto, ya que se queda atascado en un bucle.

▼ Q-LEARNING

```
1 agent_w1m_q1 = Agent(w1m, (0, 0))
2 Q_w1m_q1, rewards_w1m_q1= qlearning(agent_w1m_q1,init_state = (0,0), alpha = 0.5,gamma
3 move_with_value_function(agent_w1m_q1,Q_w1m_q1, max_steps = 100)

Initial S: (0, 0) --(A: (1, 0) Abajo ) -->
Now in S: (1, 0) --(A: (1, 0) Abajo ) -->
Now in S: (2, 0) --(A: (1, 0) Abajo ) -->
Now in S: (3, 0) --(A: (1, 0) Abajo ) -->
Now in S: (4, 0) --(A: (1, 0) Abajo ) -->
Now in S: (5, 0) --(A: (1, 0) Abajo ) -->
Now in S: (6, 0) --(A: (1, 0) Abajo ) -->
Now in S: (7, 0) --(A: (1, 0) Abajo ) -->
Now in S: (8, 0) --(A: (0, 1) Derecha ) -->
Now in S: (8, 1) --(A: (0, 1) Derecha ) -->
Now in S: (8, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (7, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (6, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (5, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (3, 2) --(A: (-1, 0) Arriba ) -->
```

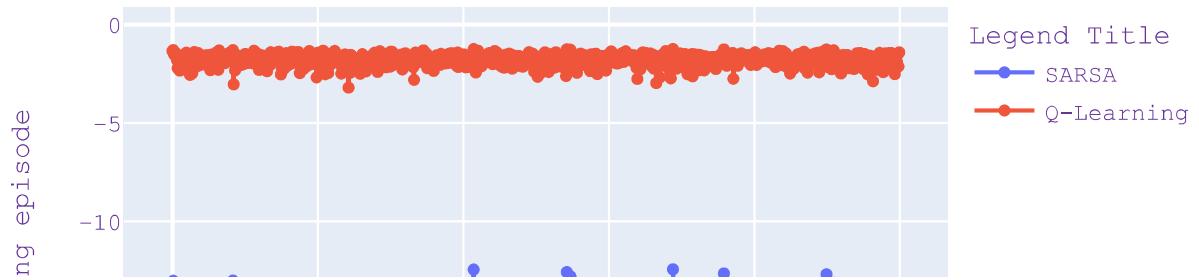
```
Now in S: (2, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (1, 2) --(A: (-1, 0) Arriba ) -->
Now in S: (0, 2) --(A: (0, 1) Derecha ) -->
Now in S: (0, 3) --(A: (0, 1) Derecha ) -->
Now in S: (0, 4) --(A: (1, 0) Abajo ) -->
Now in S: (1, 4) --(A: (1, 0) Abajo ) -->
Now in S: (2, 4) --(A: (1, 0) Abajo ) -->
Now in S: (3, 4) --(A: (1, 0) Abajo ) -->
Now in S: (4, 4) --(A: (1, 0) Abajo ) -->
Now in S: (5, 4) --(A: (1, 0) Abajo ) -->
Now in S: (6, 4) --(A: (1, 0) Abajo ) -->
Now in S: (7, 4) --(A: (1, 0) Abajo ) -->
Now in S: (8, 4) --(A: (0, 1) Derecha ) -->
Now in S: (8, 5) --(A: (0, 1) Derecha ) -->
Now in S: (8, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (7, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (6, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (5, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (4, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (3, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (2, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (1, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (0, 6) --(A: (0, 1) Derecha ) -->
Now in S: (0, 7) --(A: (0, 1) Derecha ) -->
Now in S: (0, 8) --(A: (1, 0) Abajo ) -->
Now in S: (1, 8) --(A: (1, 0) Abajo ) -->
Now in S: (2, 8) --(A: (1, 0) Abajo ) -->
Now in S: (3, 8) --(A: (1, 0) Abajo ) -->
Now in S: (4, 8) --(A: (1, 0) Abajo ) -->
Now in S: (5, 8) --(A: (1, 0) Abajo ) -->
Now in S: (6, 8) --(A: (1, 0) Abajo ) -->
Now in S: (7, 8) --(A: (1, 0) Abajo ) -->
Now in S: (8, 8) --(A: (0, 1) Derecha ) -->
END: 48 steps.
```

48

▼ Rendimiento

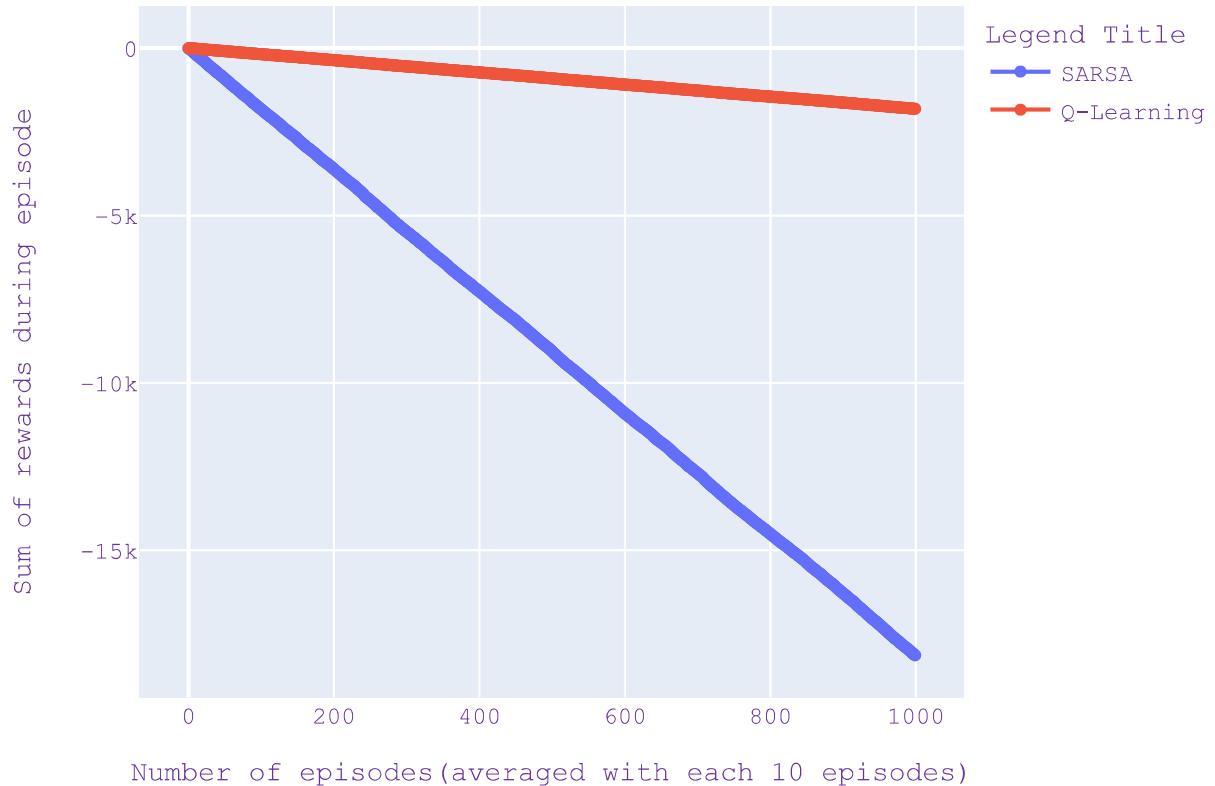
```
1 plot_performance(rewards_w1m_sarsa, rewards_w1m_ql, size = 10000, mean_each_n = 10)
```

Sum of reward for each episode (averaged with each 10 episodes)



```
1 plot_performance(rewards_w1m_sarsa, rewards_w1m_ql, size = 10000, mean_each_n = 10, acumul
```

Sum of reward for each episode (averaged with each 10 episodes)



```
Number of episodes (averaged with each 10 episodes)
```

▼ World2m

```
1 printMap(w2m)
```

```
[ 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  X  0  0  0  0  0  0  0
 0  T  X  0  0  0  0  0  0  0
 0  0  0  0  0  0  X  0  0  0
```

```

0 X X 0 0 0 0 0 0 0
0 0 0 0 0 0 X 0 0 0
0 0 0 0 0 X 0 X 0 0
0 0 0 0 0 0 T 0 X 0
0 0 0 0 0 0 0 0 0 F ]

```

▼ SARSA

```

1 agent_w2m_sarsa = Agent(w2m, (0, 0))
2 Q_w2m_sarsa, rewards_w2m_sarsa = sarsa(agent_w2m_sarsa, init_state = (0,0), alpha = 0.5,
3 move_with_value_function(agent_w2m_sarsa, Q_w2m_sarsa, max_steps = 100)

Initial S: (0, 0) --(A: (1, 0) Abajo ) -->
Now in S: (1, 0) --(A: (1, 0) Abajo ) -->
Now in S: (2, 0) --(A: (0, 1) Derecha ) -->
Now in S: (2, 1) --(A: (1, 0) Abajo ) -->
Now in S: (8, 6) --(A: (1, 0) Abajo ) -->
Now in S: (9, 6) --(A: (0, 1) Derecha ) -->
Now in S: (9, 7) --(A: (0, 1) Derecha ) -->
Now in S: (9, 8) --(A: (0, 1) Derecha ) -->
Now in S: (9, 9) --(A: (1, 0) Abajo ) -->
END: 8 steps.
8

```

▼ Q-Learning

```

1 agent_w2m_ql = Agent(w2m, (0, 0))
2 Q_w2m_ql, rewards_w2m_ql = qlearning(agent_w2m_ql, init_state = (0,0), alpha = 0.5, gamma = 0.95,
3 move_with_value_function(agent_w2m_ql, Q_w2m_ql, max_steps = 100)

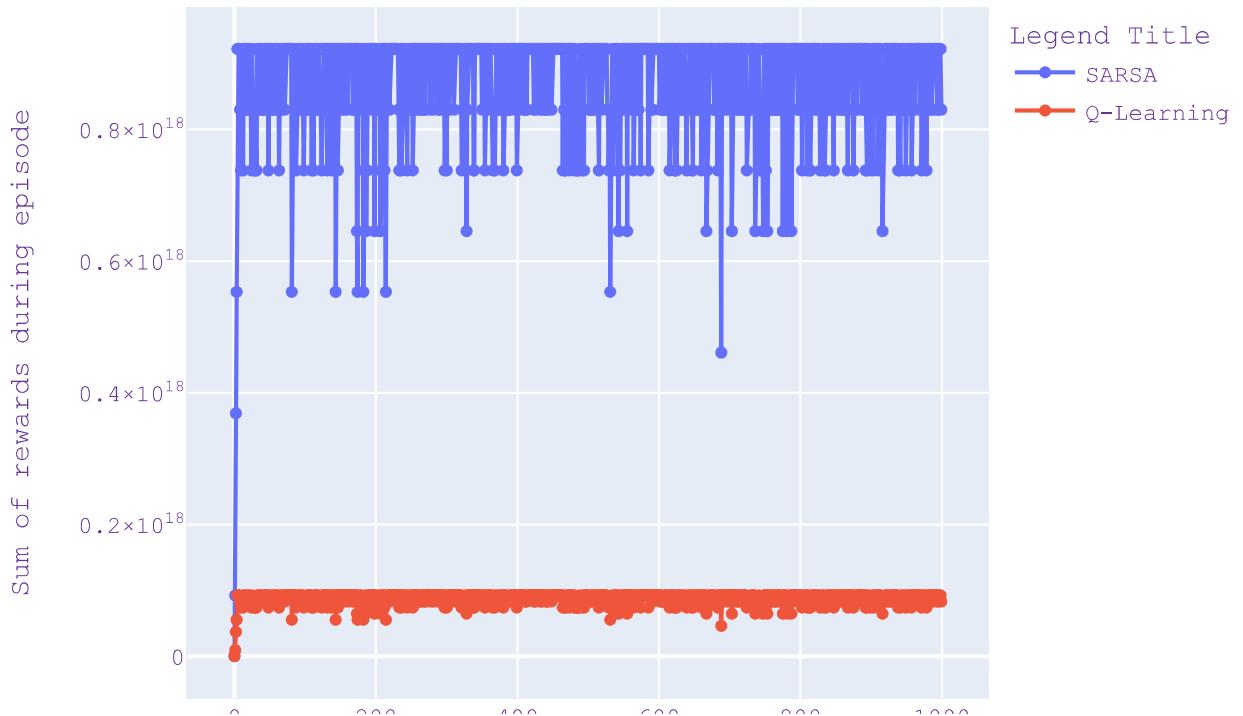
Initial S: (0, 0) --(A: (0, 1) Derecha ) -->
Now in S: (0, 1) --(A: (1, 0) Abajo ) -->
Now in S: (1, 1) --(A: (1, 0) Abajo ) -->
Now in S: (2, 1) --(A: (1, 0) Abajo ) -->
Now in S: (8, 6) --(A: (1, 0) Abajo ) -->
Now in S: (9, 6) --(A: (0, 1) Derecha ) -->
Now in S: (9, 7) --(A: (0, 1) Derecha ) -->
Now in S: (9, 8) --(A: (0, 1) Derecha ) -->
Now in S: (9, 9) --(A: (-1, 0) Arriba ) -->
END: 8 steps.
8

```

▼ Rendimiento

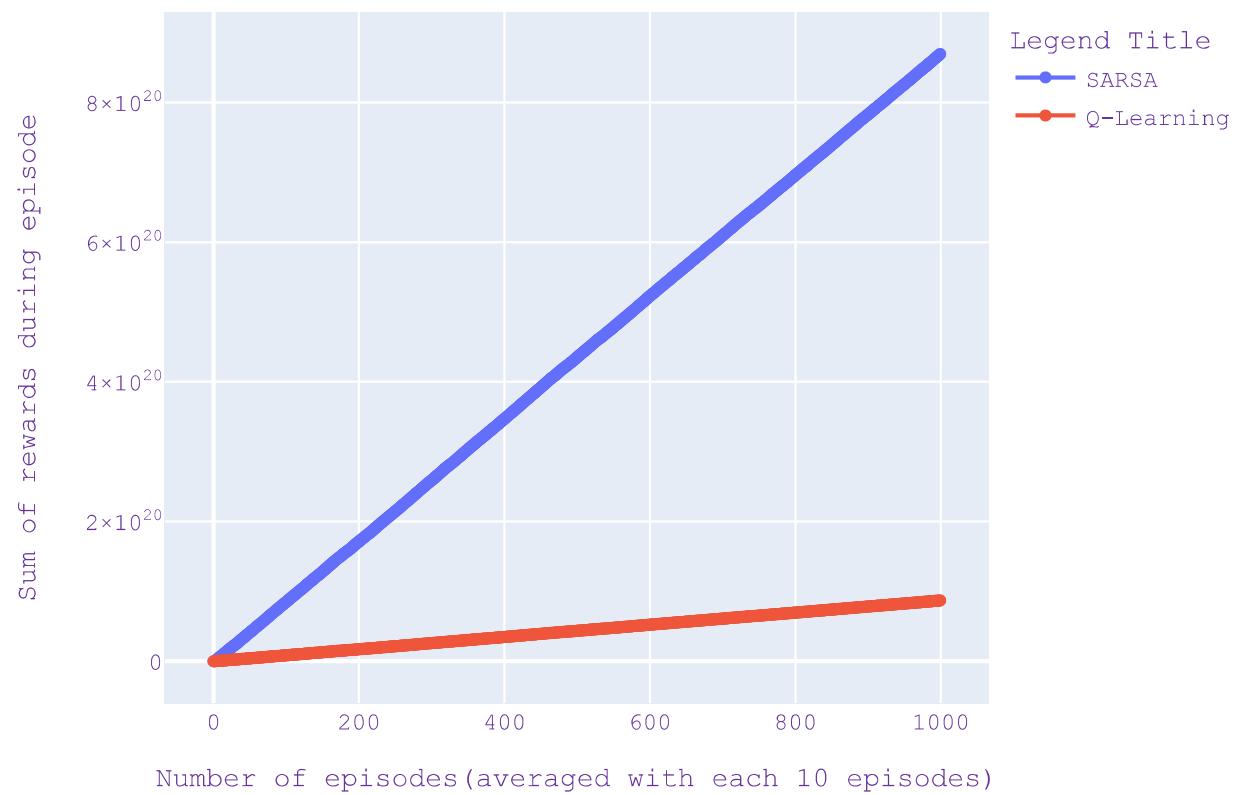
```
1 plot_performance(rewards_w2m_sarsa, rewards_w2m_ql, size = 10000, mean_each_n = 10)
```

Sum of reward for each episode (averaged with each 10 episodes)



```
1 plot_performance(rewards_w2m_sarsa, rewards_w2m_ql, size = 10000,mean_each_n = 10,acumul
```

Sum of reward for each episode (averaged with each 10 episodes)



▼ World3m

```
1 printMap(w3m)

[ 0  0  0  0  0  0  0  0  0  0
 0  0  0  X  0  0  0  0  T  0
 0  X  0  0  0  0  0  0  0  0
 0  0  0  0  0  X  0  0  X  0
 0  0  0  X  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  X  0  0  0  0  0  0  0  0
 0  0  0  0  X  0  0  0  0  0
 0  T  0  0  X  0  X  0  X  0
 0  0  0  0  0  0  0  0  F  ]
```

▼ SARSA

```
1 agent_w3m_sarsa = Agent(w3m, (0, 0))
2 Q_w3m_sarsa, rewards_w3m_sarsa = sarsa(agent_w3m_sarsa, init_state = (0,0), alpha = 0.5,
3 move_with_value_function(agent_w3m_sarsa, Q_w3m_sarsa, max_steps = 100)

Initial S: (0, 0) --(A: (1, 0) Abajo ) -->
Now in S: (1, 0) --(A: (1, 0) Abajo ) -->
Now in S: (2, 0) --(A: (1, 0) Abajo ) -->
Now in S: (3, 0) --(A: (1, 0) Abajo ) -->
Now in S: (4, 0) --(A: (1, 0) Abajo ) -->
Now in S: (5, 0) --(A: (0, 1) Derecha ) -->
Now in S: (5, 1) --(A: (0, 1) Derecha ) -->
Now in S: (5, 2) --(A: (0, 1) Derecha ) -->
Now in S: (5, 3) --(A: (0, 1) Derecha ) -->
Now in S: (5, 4) --(A: (0, 1) Derecha ) -->
Now in S: (5, 5) --(A: (0, 1) Derecha ) -->
Now in S: (5, 6) --(A: (1, 0) Abajo ) -->
Now in S: (6, 6) --(A: (0, 1) Derecha ) -->
Now in S: (6, 7) --(A: (0, 1) Derecha ) -->
Now in S: (6, 8) --(A: (0, 1) Derecha ) -->
Now in S: (6, 9) --(A: (1, 0) Abajo ) -->
Now in S: (7, 9) --(A: (1, 0) Abajo ) -->
Now in S: (8, 9) --(A: (1, 0) Abajo ) -->
Now in S: (9, 9) --(A: (0, 1) Derecha ) -->
END: 18 steps.
18
```

▼ Q-Learning

```
1 agent_w3m_ql = Agent(w3m, (0, 0))
2 Q_w3m_ql, rewards_w3m_ql = qlearning(agent_w3m_ql, init_state = (0,0), alpha = 0.5, gamma
3 move_with_value_function(agent_w3m_ql, Q_w3m_ql, max_steps = 100)

Initial S: (0, 0) --(A: (0, 1) Derecha ) -->
```

```

Now in S: (0, 1) --(A: (1, 0) Abajo ) -->
Now in S: (1, 1) --(A: (0, 1) Derecha ) -->
Now in S: (1, 2) --(A: (1, 0) Abajo ) -->
Now in S: (2, 2) --(A: (1, 0) Abajo ) -->
Now in S: (3, 2) --(A: (0, 1) Derecha ) -->
Now in S: (3, 3) --(A: (0, 1) Derecha ) -->
Now in S: (3, 4) --(A: (1, 0) Abajo ) -->
Now in S: (4, 4) --(A: (1, 0) Abajo ) -->
Now in S: (5, 4) --(A: (0, 1) Derecha ) -->
Now in S: (5, 5) --(A: (1, 0) Abajo ) -->
Now in S: (6, 5) --(A: (0, 1) Derecha ) -->
Now in S: (6, 6) --(A: (1, 0) Abajo ) -->
Now in S: (7, 6) --(A: (0, 1) Derecha ) -->
Now in S: (7, 7) --(A: (0, 1) Derecha ) -->
Now in S: (7, 8) --(A: (0, 1) Derecha ) -->
Now in S: (7, 9) --(A: (1, 0) Abajo ) -->
Now in S: (8, 9) --(A: (1, 0) Abajo ) -->
Now in S: (9, 9) --(A: (-1, 0) Arriba ) -->
END: 18 steps.

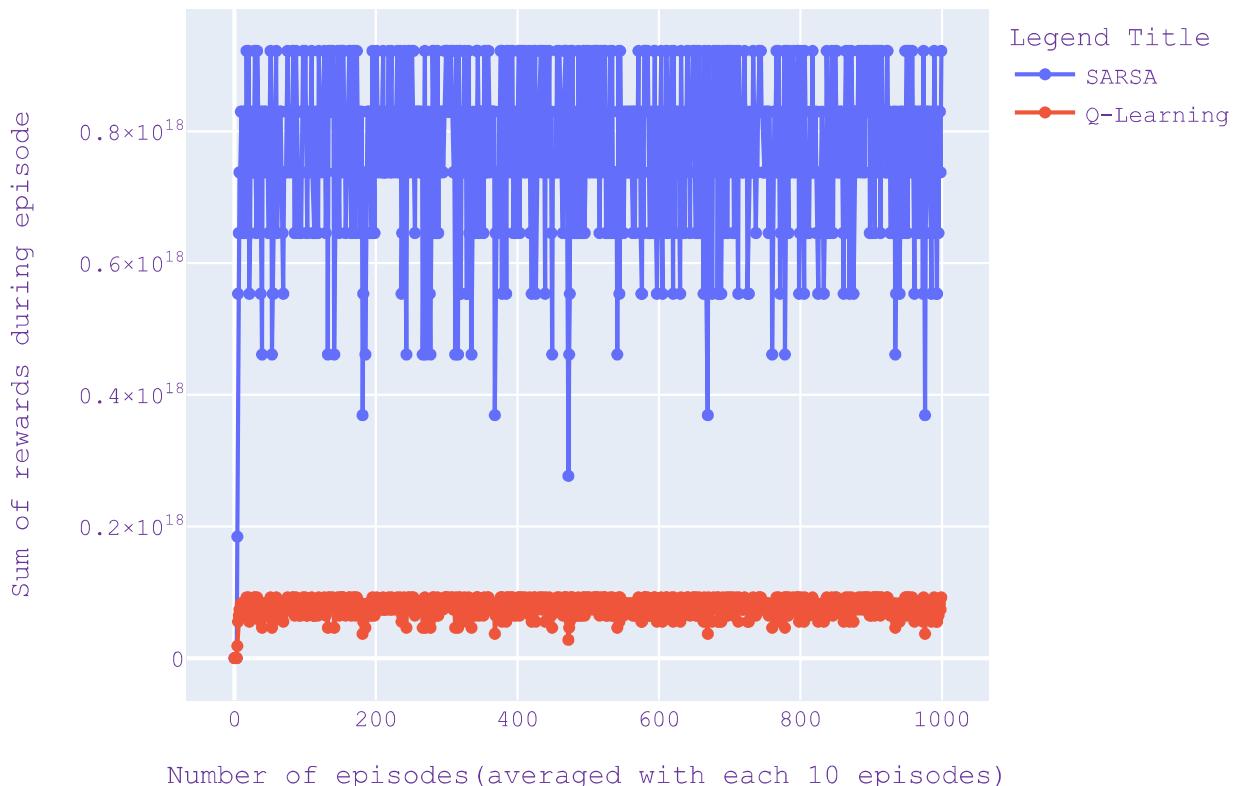
```

18

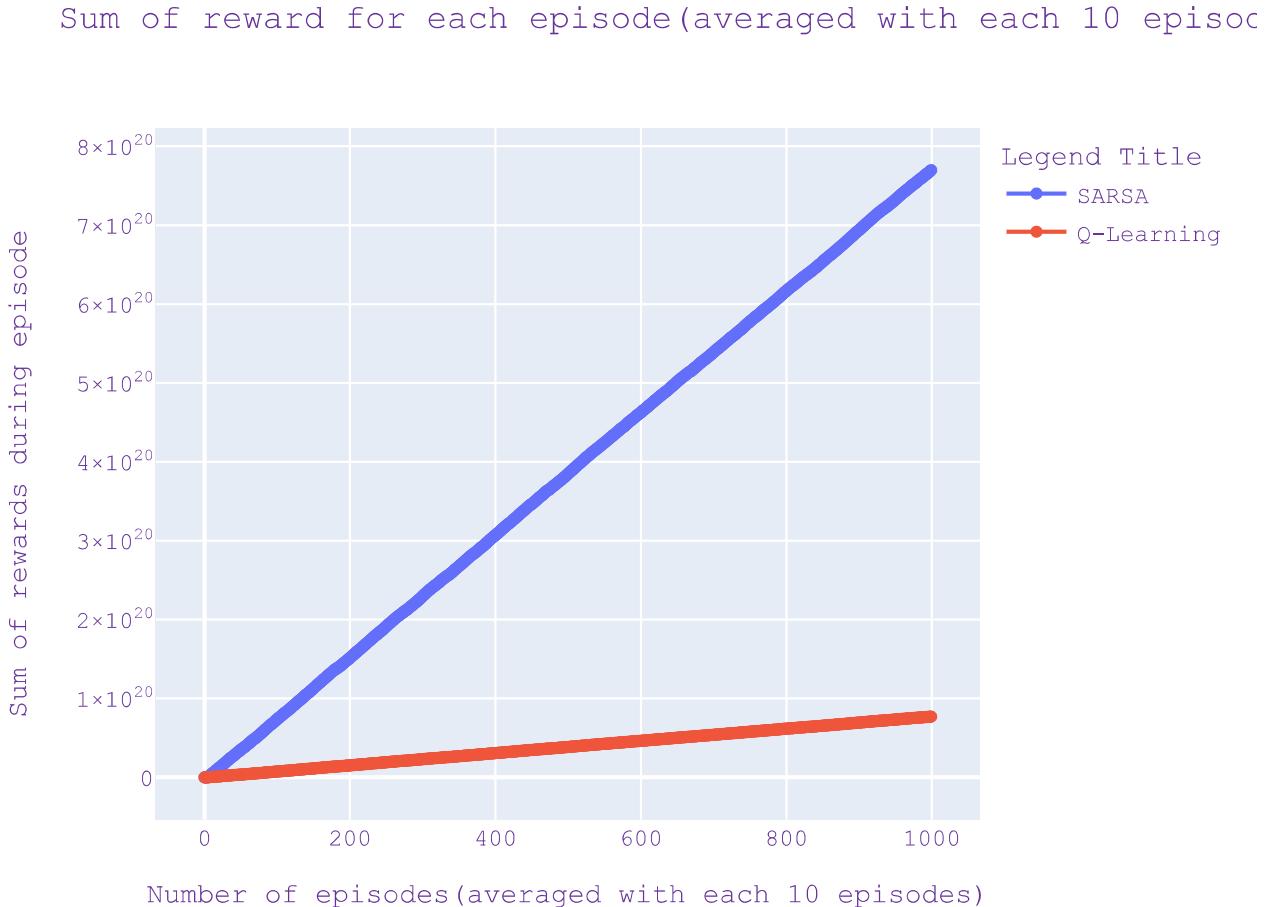
▼ Rendimiento

```
1 plot_performance(rewards_w3m_sarsa, rewards_w3m_ql, size = 10000, mean_each_n = 10)
```

Sum of reward for each episode (averaged with each 10 episodes)



```
1 plot_performance(rewards_w3m_sarsa, rewards_w3m_ql, size = 10000, mean_each_n = 10, acumul]
```



▼ World4

```
1 printMap(w4)
```

```
[ 0 X 0 X 0 0 0 0 0 0 X 0 0 0 0 0 X X X 0 X 0
 0 X 0 X X X X X 0 X X X X X 0 0 0 X 0 X 0
 0 X 0 0 0 0 0 0 0 X 0 0 0 0 X 0 X X X 0 X 0
 0 X 0 X 0 X 0 X 0 X 0 X 0 0 0 0 0 X 0 0 X 0
 0 0 0 X 0 X 0 X X X X X X X 0 X 0 X 0 X X 0
 X X X X 0 X 0 0 0 X 0 0 0 0 0 0 0 0 X 0 0 0 0
 0 0 0 0 0 X X X 0 X X X X X X 0 X X X 0 X 0
 X X X X 0 X 0 X 0 X 0 0 0 0 0 0 0 0 0 0 0 X 0
 0 0 0 X 0 0 0 X X X 0 0 0 X X X X X X X X X 0
 0 X 0 X 0 X 0 X 0 0 0 X X 0 0 0 0 0 0 0 0 0 X X
 0 X 0 X 0 X X X 0 X 0 X 0 0 0 X X X X X X X 0 0
 0 X 0 X 0 X 0 0 0 X 0 X 0 X 0 X X 0 0 0 X X X 0
 0 X 0 0 0 X X 0 X X 0 X 0 X 0 0 0 0 0 0 0 0 X 0
 0 X X X X X 0 0 X 0 0 0 0 0 0 0 0 X X X 0 X 0 X 0
 0 0 0 0 X 0 0 X X 0 X 0 X 0 X X 0 X 0 0 0 0 X 0
 X X X 0 0 0 X X X 0 0 X 0 0 X X X 0 X X X 0 X X X
 0 0 X X 0 X X X X X X 0 0 0 0 0 X 0 X 0 0 0 X 0 0
 X 0 0 X 0 X 0 0 0 X 0 0 0 X 0 0 0 X X X 0 X 0 X 0
 X X 0 0 0 X X X 0 X X X 0 0 0 X 0 0 0 X 0 0 0 X 0
 0 X X 0 X X X X X X 0 0 0 0 0 X X X X X X X X 0 ]
```

```
0 0 0 0 0 0 X X X 0 X 0 0 0 0 0 0 X F ]
```

▼ SARSA

```
1 agent_w4_sarsa = Agent(w4, (0, 0))
2 Q_w4_sarsa, rewards_w4_sarsa = sarsa(agent_w4_sarsa, init_state = (0,0), alpha = 0.5,gan
3 move_with_value_function(agent_w4_sarsa,Q_w4_sarsa, max_steps = 1000)
```

```
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
Now in S: (4, 0) --(A: (0, -1) Izquierda ) -->
```

Vemos como SARSA se queda atascado y **no consigue resolver este laberinto**

▼ Q-Learning

```

1 agent_w4_q1 = Agent(w4, (0, 0))
2 Q_w4_q1, rewards_w4_q1 = qlearning(agent_w4_q1, init_state = (0,0), alpha = 0.5, gamma =
3 move_with_value_function(agent_w4_q1, Q_w4_q1, max_steps = 1000)

```

```
Now in S: (8, 1) --(A: (0, -1) Izquierda ) -->
Now in S: (8, 0) --(A: (1, 0) Abajo ) -->
Now in S: (9, 0) --(A: (1, 0) Abajo ) -->
Now in S: (10, 0) --(A: (1, 0) Abajo ) -->
Now in S: (11, 0) --(A: (1, 0) Abajo ) -->
Now in S: (12, 0) --(A: (1, 0) Abajo ) -->
Now in S: (13, 0) --(A: (1, 0) Abajo ) -->
Now in S: (14, 0) --(A: (0, 1) Derecha ) -->
Now in S: (14, 1) --(A: (0, 1) Derecha ) -->
Now in S: (14, 2) --(A: (0, 1) Derecha ) -->
Now in S: (14, 3) --(A: (1, 0) Abajo ) -->
Now in S: (15, 3) --(A: (0, 1) Derecha ) -->
Now in S: (15, 4) --(A: (0, 1) Derecha ) -->
Now in S: (15, 5) --(A: (-1, 0) Arriba ) -->
Now in S: (14, 5) --(A: (0, 1) Derecha ) -->
Now in S: (14, 6) --(A: (-1, 0) Arriba ) -->
Now in S: (13, 6) --(A: (0, 1) Derecha ) -->
Now in S: (13, 7) --(A: (-1, 0) Arriba ) -->
Now in S: (12, 7) --(A: (-1, 0) Arriba ) -->
Now in S: (11, 7) --(A: (0, 1) Derecha ) -->
Now in S: (11, 8) --(A: (-1, 0) Arriba ) -->
Now in S: (10, 8) --(A: (-1, 0) Arriba ) -->
Now in S: (9, 8) --(A: (0, 1) Derecha ) -->
Now in S: (9, 9) --(A: (0, 1) Derecha ) -->
Now in S: (9, 10) --(A: (1, 0) Abajo ) -->
Now in S: (10, 10) --(A: (1, 0) Abajo ) -->
Now in S: (11, 10) --(A: (1, 0) Abajo ) -->
Now in S: (12, 10) --(A: (1, 0) Abajo ) -->
Now in S: (13, 10) --(A: (0, 1) Derecha ) -->
Now in S: (13, 11) --(A: (0, 1) Derecha ) -->
Now in S: (13, 12) --(A: (0, 1) Derecha ) -->
Now in S: (13, 13) --(A: (0, 1) Derecha ) -->
Now in S: (13, 14) --(A: (-1, 0) Arriba ) -->
Now in S: (12, 14) --(A: (0, 1) Derecha ) -->
Now in S: (12, 15) --(A: (0, 1) Derecha ) -->
Now in S: (12, 16) --(A: (0, 1) Derecha ) -->
Now in S: (12, 17) --(A: (0, 1) Derecha ) -->
```

```
NOW in S: (12, 18) --(A: (1, 0) Abajo ) -->
Now in S: (13, 18) --(A: (1, 0) Abajo ) -->
Now in S: (14, 18) --(A: (0, -1) Izquierda ) -->
Now in S: (14, 17) --(A: (0, -1) Izquierda ) -->
Now in S: (14, 16) --(A: (1, 0) Abajo ) -->
Now in S: (15, 16) --(A: (1, 0) Abajo ) -->
Now in S: (16, 16) --(A: (1, 0) Abajo ) -->
Now in S: (17, 16) --(A: (1, 0) Abajo ) -->
Now in S: (18, 16) --(A: (0, 1) Derecha ) -->
Now in S: (18, 17) --(A: (0, 1) Derecha ) -->
Now in S: (18, 18) --(A: (-1, 0) Arriba ) -->
Now in S: (17, 18) --(A: (-1, 0) Arriba ) -->
Now in S: (16, 18) --(A: (0, 1) Derecha ) -->
Now in S: (16, 19) --(A: (0, 1) Derecha ) -->
Now in S: (16, 20) --(A: (1, 0) Abajo ) -->
Now in S: (17, 20) --(A: (1, 0) Abajo ) -->
Now in S: (18, 20) --(A: (1, 0) Abajo ) -->
Now in S: (19, 20) --(A: (1, 0) Abajo ) -->
Now in S: (20, 20) --(A: (0, -1) Izquierda ) -->
END: 82 steps.
```

82

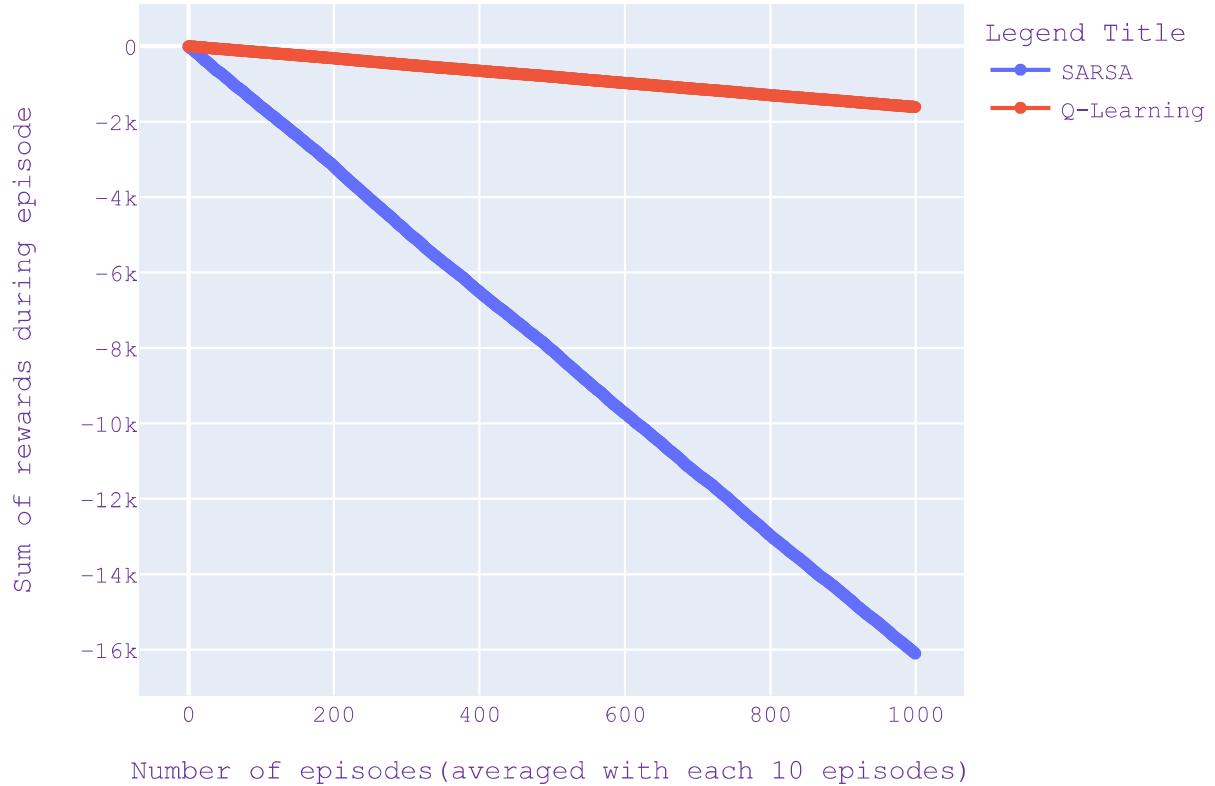
▼ Rendimiento

Vemos como este último laberinto Q-Learning sí ha sido capaz de resolverlo, pero SARSA no.

```
1 plot_performance(rewards_w4_sarsa, rewards_w4_ql, size = 10000, mean_each_n = 10)
```

```
1 plot_performance(rewards_w4_sarsa, rewards_w4_ql, size = 10000, mean_each_n = 10, acumula
```

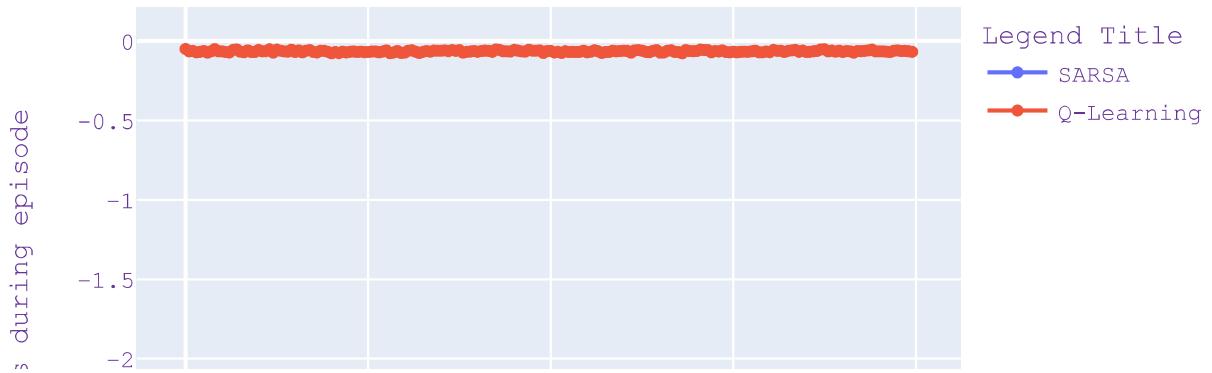
Sum of reward for each episode (averaged with each 10 episodes)



Promediamos ahora con 50 episodios, por ser un laberinto mucho más grande sobre el que hemos hecho 100k épocas:

```
1 plot_performance(rewards_w4_sarsa, rewards_w4_ql, size = 10000, mean_each_n = 50)
```

Sum of reward for each episode (averaged with each 50 episodes)



Análisis:

SARSA y Q-Learning son dos algoritmos muy parecidos, que se pueden aplicar en los mismos problemas y suelen encontrar soluciones similares. No obstante, los resultados de ambos algoritmos pueden diferir en ciertos problemas: por ejemplo, hay un problema llamado Cliffworld en el que SARSA encuentra soluciones más seguras y con menos valor, mientras que Q-Learning asume más riesgos y consigue más valor ([artículo interesante](#)).

Number of episodes (averaged with each 50 episodes)

▼ Ejercicio 3:

Analizad los resultados obtenidos por ambos algoritmos en los escenarios de prueba.

1. Comentad el rendimiento que observáis en ambos algoritmos. ¿Qué problemas son capaces de resolver? ¿En cuáles no encuentran la solución óptima? ¿A qué se puede deber este comportamiento?

Para comparar los rendimientos, **podemos comparar la suma de las recompensas por cada episodio, frente al incremento de número de episodios para cada algoritmo.**

Q-Learning es mejor para este tipo de problemas, ya que es el único que ha conseguido resolver todos los tipos de laberintos. En cambio SARSA los laberintos medianos/grandes en los que haya que hacer muchos giros debido a los obstáculos, observamos que no es capaz de resolverlo, y se acaba atascando en algún punto, haciendo movimientos repetitivos. Ambos consiguen resolver los laberintos de manera óptima(siempre que los resuelven para los ejemplos probados).

Esto es debido a que Q-Learning tiene un aprendizaje fuera de política: por lo tanto se asume que su política ya es óptima, ya que siempre escoge la acción que mayor valor le aporta. En cambio SARSA debe evaluar las acciones disponibles en cada paso bajo la restricción de su probabilidad en la política actual. Es decir, con SARSA un agente utilizará la política primero

para observar la probabilidad de tomar una acción, evaluará la recompensa de esa acción, y luego se comprometerá con otra acción siguiente para actualizar el valor Q de su par estado-acción actual en consecuencia.

2. Comentad las diferencias entre los algoritmos en los diferentes escenarios: ¿Cuál resuelve más escenarios? ¿Cuál converge más rápido? ¿Cuál genera más valor?

Nota: Las siguientes variables pueden ser interesantes para valorar los resultados: Diferencia entre la política resultante y la política óptima, número de iteraciones necesarias para converger, retorno total del problema y retorno obtenido en cada episodio.

Para los escenarios más grandes vemos como necesitamos más épocas: para el escenario 4 hemos utilizado 100k, mientras que para el resto nos ha bastado con 10k para que los algoritmos convergieran.

Cabe destacar que para los **ejemplos en los que ambos algoritmos han coseguido resolver los laberintos**, ambos lo hacen en el **mismo número de pasos** y de manera** óptima. **Además, en estos casos, si observamos las gráficas de **rendimiento**, es decir, la suma de recompensas por episodio, observamos que el rendimiento de **SARSA siempre es mayor**. Esto es debido, a que SARSA suele dedicar más tiempo a **explorar con pasos subóptimos**, y por tanto, las recompensas positivas acaban sumando más. Aunque también las negativas, de ahí que su gráfica tenga muchos más picos.

En cambio, en los **casos en los que SARSA no consigue resolver los laberintos**: World4 y World1m, vemos como su rendimiento está como es de esperar por debajo del de Q-Learning. Este comportamiento se puede ver además en la gráfica de "**suma acumulada de recompensas**": Mientras que cuando los algoritmos son capaces de resolver los laberintos, esta gráfica es creciente, para SARSA en estos dos mundos, **su linea es decreciente, indicando que no ha sido capaz de converger el algoritmo**. Como Q-Learning consigue resolverlos todos, todas sus gráficas aquí son crecientes.

Es decir, mientras que la gráfica de **suma de recompensas por episodios** tiene una interpretabilidad equiparable al "**rendimiento de los algoritmos**", la gráfica "**suma acumulada de recompensas**", tiene una interpretabilidad de **convergencia o no** de los algoritmos.

3. Comentad las diferencias cuando se aplica una mayor exploración (ϵ más alto) y una mayor explotación (ϵ más bajo). ¿Cuál converge más rápido? ¿Cuál obtiene más valor? ¿Qué estrategia piensas que podría usarse para explorar y explotar de forma más inteligente?

Un ϵ muy alto, en el que el algoritmo se dedique en prioridad a explorar, probablemente no garantice la convergencia del mismo, ya que tardará en encontrar una solución optima.

En cambio si escogemos un ϵ muy bajo, la prioridad del algoritmo será demasiado codiciosa y tampoco convergerá en laberintos medianamente complicados, ya que probablemente entre en

bucles entre casillas, debido a que se dedica básicamente a maximizar valores, y se olvida de explorar.

La estrategia más inteligente resulta de dedicar la mayor parte del esfuerzo a explotar, y una pequeña fracción a explorar. **Es decir, un valor de ϵ pequeño.** Por ejemplo una valor de epsilon que nos ha ofrecido buen rendimiento es del 0.1, es decir, que dedique un 10% de las veces que decide tomar una acción a explorar el mapa, y el 90% restante a explotar y optimizar los valores.

4. Comentad las diferencias cuando se varían otros parámetros como el número de episodios, el ratio de aprendizaje (α) o el factor de descuento (γ). ¿Qué valores dan mejores resultados?

Variar el valor de los refuerzos, es otro hiperparámetro que ya describimos más arriba.

Para el número de episodios es un valor que depende de la complejidad de cada problema. En nuestro caso la convergencia la hemos hecho en base a un número máximo de episodios, lo cual no ha obligado a tener que prefijar y probar distintos números de episodios hasta ver si los algoritmos conseguían o no resolver los laberintos(aunque este no era el único factor del que dependía la convergencia como explicábamos antes, ya que un numero más alto de episodios no hacía que SARSA convergiera en algunos laberintos). Para laberintos pequeños y medianos nos han valido con 10k episodios, para el World4 eran ya insuficientes. Bien es cierto que también depende del algoritmo, podemos verlo en las gráficas de convergencia: Q-Learning lo hace a los pocos episodios y se estabiliza muy rápidamente, mientras que la convergencia en SARSA es mucho más gradual.

Para el ratio de aprendizaje un valor en torno al 0.5 nos ha arrojado un buen rendimiento. Si lo tomamos muy alto, aprenderán demasiado rápido los algoritmos y serán demasiado sensibles a los cambios, provocando un comportamiento errático. Valores muy pequeños en cambio provocarán que no se actualicen apenas los valores, lo que disparará el número necesario de épocas para que los algoritmos aprendan a resolver los laberintos, lo cual no parece la mejor de las opciones, si queremos una convergencia.

Para el factor de descuento un valor alto (0.9 - 1) nos arroja un mejor rendimiento que valores más pequeños. Esto es debido, a que si tomamos un valor muy bajo, los algoritmos no tendrán mucho en cuenta el valor de los estados contiguos según las acciones tomadas, y eso es precisamente lo que se busca aprender con las políticas basadas en valor.

✓ 0s completed at 5:36 AM



Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.