# IIC3675: Métodos Tabulares

## Introducción

En esta tarea resolverás problemas de decisión utilizando model-free methods (basados en TD learning) y model-based methods (Dyna y RMax). Para ello, te recomendamos estudiar la materia del curso y los capítulos 6, 7 y 8 del libro: "*Reinforcement Learning: An introduction.*"

El código base incluye varios tipos de dominios. Existen algunos dominios simples – muy parecidos a los de la tarea 2. Pero también hay dominios más complejos, que incluyen múltiples objetivos, múltiples agentes y observación parcial. Además hay una carpeta con distintos tipos de memorias que te ayudarán a resolver los dominios con observación parcial.

En esta tarea pondremos a nuestros agentes de RL en situaciones incómodas... y veremos cómo les va :)

### Dominios simples

Todos los dominios se encuentran en la carpeta `Environments`. Al igual que en la tarea 2, todo ambiente hereda de `AbstractEnv`. Esta herencia asegura que tengan implementados los siguientes métodos:

- `action_space`: Es un property que retorna la lista de acciones disponibles para el agente.
- `reset()`: Reinicia el ambiente y retorna el estado inicial del problema.
- `step(action)`: Ejecuta `action` en el ambiente y, como resultado, retorna una tupla con 3 elementos: (1) el siguiente estado, (2) la recompensa inmediata 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.

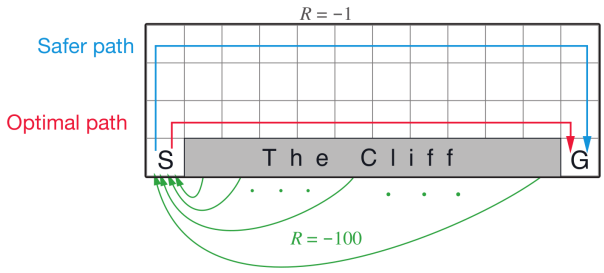
Estos métodos permiten programar agentes genéricos que funcionan con cualquier ambiente que herede de `AbstractEnv`. Por ejemplo, el siguiente código permite interactuar (manualmente) con cualquiera de los ambientes simples usando las teclas `asdw`.

```
def play_simple_env(simple_env):
    key2action = {'a': "left", 'd': "right", 'w': "up", 's': "down"}
    s = simple_env.reset()
    show(simple_env, s)
    done = False
    while not done:
        print("Action: ", end="")
        action = get_action_from_user(key2action)
        s, r, done = simple_env.step(action)
        show(simple_env, s, r)
```

Los dos ambientes simples son grillas donde el agente se puede mover en las 4 direcciones cardinales. El estado incluye la posición del agente y, a veces, la posición de otros elementos en la grilla. Las posiciones se codifican usando tuplas `(i, j)`, donde `i` es la fila y `j` es la columna. La posición `(0, 0)` es la esquina superior izquierda.

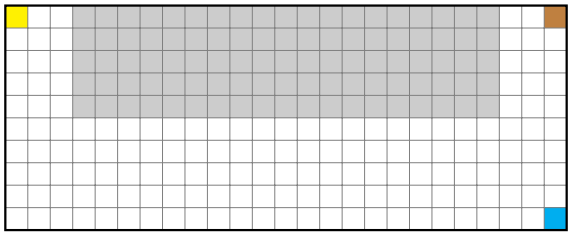
#### CliffEnv

Este ambiente implementa el dominio *Cliff Walking* descrito en el *Example 6.6* del libro de Sutton & Barto. El agente parte en el estado `S` y el episodio termina al llegar a `G`. La recompensa es `-1` en cada paso y `-100` si el agente cae al *cliff*. Al caer al *cliff* el agente vuelve al punto de partida.




**EscapeRoomEnv**

En esta grilla el objetivo es escapar de una habitación que contiene clavos (representados con cuadrados grises), una llave (representada con un cuadrado amarillo) y una puerta (representada con un cuadrado café). Para escapar el agente debe obtener la llave y abrir la puerta. Sin la llave la puerta no abre.



Abrir la puerta termina el episodio. La recompensa es `-10` por pisar un clavo y `-1` en otro caso. El estado consiste en la posición del agente y un booleano que indica si el agente tiene la llave. Al inicio de cada episodio, el agente comienza en el cuadrado celeste sin la llave.

## Dominios multi objetivo

*Multi-Goal RL* es un área de investigación donde el objetivo es crear agentes que puedan resolver distintas tareas en el mismo ambiente. Existen muchas maneras de formalizar este problema pero acá veremos la más básica, donde la tarea consiste en que el agente llegue a cierto estado objetivo.

Para ello, se define un *environment* $\mathcal{E} = (\mathcal{S}, \mathcal{A}, p_t, \gamma)$ como un MDP sin función de recompensa. Es decir, contiene un set finito de estados $\mathcal{S}$, acciones $\mathcal{A}$, un factor de descuento $\gamma$ y una función de transición Markoviana $p_t(s'|s, a)$ – que define la probabilidad de llegar a $s'$ si ejecutamos la acción $a$ en $s$. Además existe un conjunto de estados objetivos $\mathcal{G} \subseteq \mathcal{S}$.

La tarea consiste en aprender una política óptima $\pi_*(a|s, g)$ que para cualquier estado del ambiente $s \in \mathcal{S}$ y objetivo $g \in \mathcal{G}$ seleccione acciones tal que se llegue desde $s$ hasta $g$ de manera óptima.

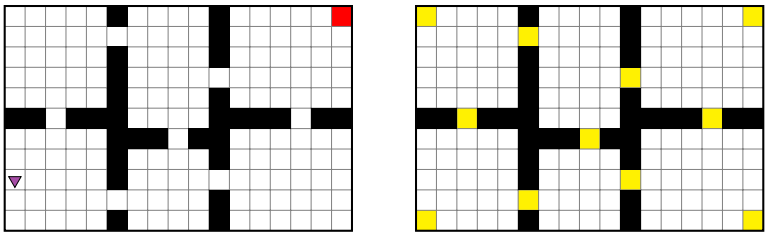
Para aprender $\pi_*(a|s, g)$ generalmente se trabaja sobre un MDP extendido. Donde el estado pasa a ser un par $(s, g) \in \mathcal{S} \times \mathcal{G}$. Al inicio de cada episodio se fija un objetivo $g \in \mathcal{G}$ de manera aleatoria. Luego la función de recompensa es `+1` si y solo si se llega a un estado $(s, g)$ en que $s = g$ (la recompensa es `0` en otro caso).

El código base incluye un ambiente multi objetivo: `RoomEnv`. Este ambiente consiste en una grilla con 6 habitaciones. En cada episodio se define un estado objetivo al que el agente debe llegar. El estado del agente es una tupla donde el primer valor es la posición del agente y el segundo valor es la posición del estado objetivo. La recompensa es `+1` cuando al agente llega a la posición objetivo y `0` en otro caso.

El conjunto de estados objetivos posibles (marcados en amarillo) son las cuatro esquinas de la grilla y todas las casillas que conectan dos habitaciones.

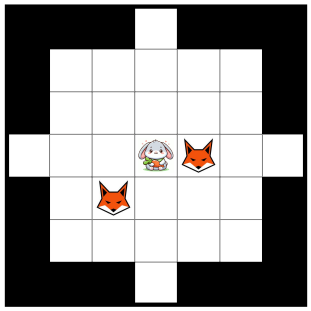
`RoomEnv`, además de implementar los métodos de `AbstractEnv`, implementa la siguiente property:

- `goals`: Retorna una lista con todas las posiciones de casillas que pueden ser muestreadas como objetivo.



## Dominios multi agente

En este dominio existen dos cazadores y una presa. Los cazadores reciben recompensa positiva por comerse a la presa. La presa recibe recompensa negativa al ser comida :(



En *Multi-Agent RL* el objetivo es definir agentes que puedan resolver problemas en conjunto con otros agentes. Aquí existen varios tipos de problemas. Hay problemas que son *cooperative* donde varios agentes deben cooperar para resolver una tarea. En este escenario los agentes comparten una misma función de recompensa por lo que existe un incentivo para colaborar. Por otro lado, hay problemas que son *competitive* donde los agentes compiten entre ellos. En estos casos solo puede existir un ganador (que recibe recompensa positiva) y el resto son perdedores (que reciben recompensas negativas).

Además de la distinción entre cooperative y competitive, también existen problemas que son *centralized* y otros que son *decentralized*. En el caso centralized todos los agentes son controlados por un único agente de manera centralizada. Se puede pensar como un usuario que controla a todas sus unidades en juego como *StarCraft*. El caso decentralized cada agente tiene una mente propia y decide qué hacer de manera independiente – sin claridad de lo que hará el resto.

El código base incluye tres versiones del dominio de cazadores y presas:

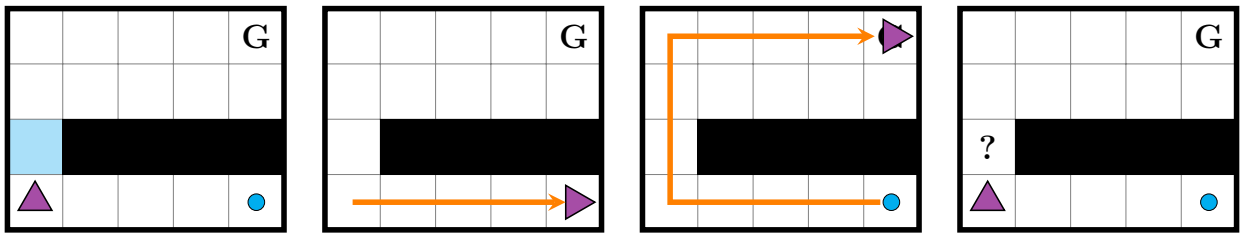
- `CentralizedHunterEnv`: En este caso la presa sigue una política fija (se aleja de los cazadores) y los cazadores son controlador por un único agente que decide – de manera conjunta – las acciones que realiza cada cazador. Es decir, las acciones son una tupla $(a_1, a_2)$ donde $a_1$ es la acción que ejecuta el cazador 1 y $a_2$ es la acción que ejecuta el cazador 2.
- `HunterEnv`: Este dominio es idéntico a `CentralizedHunterEnv`, pero acá cada cazador es un agente independiente. Por lo mismo, la función `step` retornará dos recompensas – una por agente.
- `HunterAndPreyEnv`: Este dominio es igual al dominio anterior pero la presa ya no sigue una política fija. En este dominio, la presa es otro agente de RL que aprende una política que le permita sobrevivir. La función `step` recibe una tupla con 3 acciones (la acción del cazador 1, cazador 2 y de la presa). Retorna (i) un estado que es idéntico para los 3 agentes, (ii) una tupla con 3 recompensas (la primera es la recompenza del cazador 1, la segunda es la recompensa del cazador 2 y la tercera es la recompensa de la presa) y (iii) un bool que indica si el episodio terminó.

Estos tres ambientes heredan de `AbstractMultiAgentEnv`. Esto asegura que, además de implementar los métodos de un `AbstractEnv`, también implementan los siguientes métodos:

- `single_agent_action_space`: Es un property que retorna la lista de acciones que un agente individual puede ejecutar. Notar que en estos ambientes `action_space` retorna una lista con todas las combinaciones de acciones que se pueden realizar en conjunto en el ambiente.
- `num_of_agents`: Es un property que retorna el número de agentes en el ambiente.

### Dominios con observación parcial

Considera el siguiente dominio. Se trata de una grilla donde el objetivo es llegar a la posición `G`. Sin embargo, existe una puerta de vidrio que le bloquea el paso al agente. Para abrir la puerta el agente puede presionar un botón que se encuentra en la esquina inferior derecha. Volver a apretar el botón vuelve a cerrar la puerta (y así sucesivamente). Como tal, la política óptima es apretar el botón una vez y luego ir a `G`. Sin embargo, el agente no puede ver la puerta de vidrio. Por lo que no sabe si está abierta o cerrada.




En concreto, el estado real del MDP incluye la posición del agente y un booleano que indica si la puerta está abierta. Sin embargo, lo que observa el agente es solo su posición. Este problema es interesante porque incluye información parcial. De hecho, a diferencia de un problema clásico, una política óptima requiere memoria. El agente tiene que aprender a recordar que presionó el botón para saber si la puerta está abierta. De lo contrario, cuando el agente esté frente a la puerta, no tendrá información suficiente para saber si la acción óptima es ir hacia `G` o hacia el botón.

Cuando consideramos agentes interactuando en entornos realistas, generalmente estos agentes tienen una observación parcial del entorno. Esto complica considerablemente las cosas para los agentes de aprendizaje reforzado. De partida, ahora requieren memoria para actuar de manera óptima.

Acá daremos un paso hacia entender las implicancias de tener que resolver un problema con observación parcial. El dominio a resolver se llama `InvisibleDoorEnv`. Para que sea factible resolverlo, incluimos distintos tipos de memorias que serán explicados más adelante. Además pusimos un tiempo límite de 5000 pasos para resolver el problema. Este tiempo límite lo agradecerás... créeme.

## 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 (o tablas) con los experimentos realizados y responder las siguientes preguntas (justifica tus respuestas):

a) **[1 punto]** ¿Por qué Q-Learning se considera un off-policy method mientras que Sarsa se considera un on-policy method?

b) **[1 punto]** Considera el caso en que la selección de acciones es greedy. ¿Son Sarsa y Q-Learning equivalentes en esta caso (i.e., eligen las mismas acciones y realizan las mismas actualizaciones en todo momento)? Si la respuesta es sí, demuestra que son equivalentes. Si es no, muestra un caso en que no sean equivalentes.

c) **[2 puntos]** Compara el rendimiento de Q-learning, Sarsa y 4-step Sarsa en el dominio `CliffEnv`. Utiliza $\epsilon = 0.1$ para explorar, un learning rate de $\alpha = 0.1$ y descuento $\gamma = 1.0$. Inicializa los Q-values en 0. Corre

100 veces cada experimento y grafica el retorno promedio por episodio para los primeros 500 episodios. En tu gráfico, fuerza a que el eje-Y parta en -200 para que sea más fácil ver la parte más interesante de los resultados. Explica en detalle por qué los resultados que obtuviste están bien sustentados por la teoría. Por ejemplo, digamos que Q-learning le gana a Sarsa y Sarsa le gana a 4-step Sarsa. Explica por qué Q-learning le debería ganar a Sarsa y luego explica por qué Sarsa le debería ganar a 4-step Sarsa.

d) **[3 puntos]** Compara el rendimiento de Dyna y RMax en el dominio `EscapeRoomEnv`. Corre cada método 5 veces por 20 episodios. Usa $\gamma = 1.0$. Para Dyna utiliza $\alpha = 0.5$, $\epsilon = 0.1$ e inicializa los Q-values en 0. Reporta en una tabla el retorno medio por episodio de RMax y Dyna utilizando 0, 1, 10, 100, 1000 y 10000 pasos de planning. ¿Es verdad que Dyna se vuelve equivalente a RMax cuando el número de pasos de planning es suficientemente grande? Justifica tu respuesta.

e) **[3 puntos]** Ahora resolveremos el dominio multi objetivo `RoomEnv`. Un dominio multi objetivo es un MDP como cualquier otro, por lo que puede ser resuelto usando cualquier algoritmo de RL. Sin embargo, también es posible aprovechar su estructura para diseñar agentes que los resuelvan más rápido.

Recordemos que en un problema multi objetivo el estado se compone de dos partes: $\langle s, g \rangle$ – donde $s$ es el estado en el ambiente y $g$ es el objetivo actual. En este caso, asumiremos que la recompensa es siempre cero salvo cuando $s = g$ (donde la recompensa es 1).

Consideremos qué ocurre cuando resolvemos un problema multi objetivo usando Q-learning. Digamos que en el episodio actual el objetivo es $g_c$. El estado actual en el ambiente es $s$, se ejecuta la acción $a$ y se llega al estado $s'$. Producto de esta experiencia, Q-learning realizará la siguiente actualización:

$$Q(\langle s, g_c \rangle, a) = \begin{cases} Q(\langle s, g_c \rangle, a) + \alpha \left[ \gamma \max_{a' \in \mathcal{A}} Q(\langle s', g_c \rangle, a') - Q(\langle s, g_c \rangle, a) \right], & \text{si } s' \neq g_c \\ Q(\langle s, g_c \rangle, a) + \alpha \left[ 1 - Q(\langle s, g_c \rangle, a) \right], & \text{si } s' = g_c \end{cases}$$

Lo interesante es que, independiente del objetivo actual del agente $g_c$, esta actualización es válida para todo $g \in \mathcal{G}$. Por lo mismo, una versión *multi-goal* de Q-learning puede usar la experiencia $(s, a, s')$ para actualizar todos los Q-values $Q(\langle s, g \rangle, a)$ para todos los objetivos posibles $g \in \mathcal{G}$. Y la misma idea se podría aplicar a Sarsa sobre experiencias $(s, a, r, s', a')$.

Resuelve el dominio `RoomEnv` usando Q-learning, Sarsa, 8-step Sarsa y versiones multi-goal de Q-learning y Sarsa. Utiliza $\alpha = 0.1$, $\epsilon = 0.1$, $\gamma = 0.99$ e inicializa los Q-values en 1.0. Corre cada método 100 veces por 500 episodios cada uno. Luego grafica el largo promedio de los episodios de cada algoritmo.

¿Son las versiones multi-goal efectivamente mejores que los baselines (i.e., las versiones estándar de Q-learning, Sarsa y 8-step Sarsa)? ¿Entre utilizar multi-goal Q-learning vs multi-goal Sarsa, cuál es mejor? ¿Por qué?

f) **[4 puntos]** Ahora trabajaremos con dominios multi agente. Cuando tenemos varios agentes interactuando con un ambiente hay muchos settings distintos. El más simple se le conoce como *Centralized Cooperative Multi-Agent RL*. En este setting tenemos un solo agente que controla varios agentes al mismo tiempo. Como tal, se trata de un MDP estándar pero con muchas (en general demasiadas) acciones. Cada acción en el MDP consiste en $n$ acciones (una por agente).

Comienza resolviendo `CentralizedHunterEnv` usando Q-learning. Este ambiente consiste en dos cazadores que tienen que cazar una presa. La presa sigue una política fija y las acciones consisten en todos los pares de acciones que pueden realizar los cazadores en conjunto (es un dominio centralizado). Utiliza $\gamma = 0.95$, $\epsilon = 0.1$, $\alpha = 0.1$ e inicializa los Q-values en 1.0. Corre Q-learning 30 veces por 50000 episodios. Reporta el largo promedio de cada episodio (puedes reportar el valor promedio cada 100 episodios para que el gráfico no quede tan sobrecargado).

El segundo setting se conoce como *Decentralized Cooperative Multi-Agent RL*. En este caso ambos agentes comparten la misma función de recompensa, pero seleccionan acciones de manera independiente.

Resuelve el dominio `HunterEnv` usando dos agentes de Q-learning que controlen, de manera independiente, a cada cazadores. Utiliza los mismos hyperparámetros que en el experimento anterior. También corre el experimento 30 veces por 50000 episodios y reporta el largo promedio de los episodios (cada 100 episodios).

El tercer setting se conoce como *Decentralized Competitive Multi-Agent RL*. En este caso, los agentes compiten y sus funciones de recompensa son opuestas.

Resuelve el dominio `HunterAndPreyEnv`. Este dominio es igual al `HunterEnv` pero acá la presa también aprende. Es decir, debes utilizar 3 agentes de Q-learning independientes. Uno para cada cazador y uno para la presa. Usando los mismos hyperparámetros que en los casos anteriores, corre 30 veces el experimento por 50000 episodios y reporta el largo promedio de cada episodio (cada 100 episodios).

Reporta las tres curvas en un solo gráfico. Explica en detalle las tendencias que se ven. Por ejemplo, explica subidas, bajadas y por qué en cierto setting se aprende más rápido que en otro.

g) **[4 puntos]** Ahora resolveremos dominios con observación parcial. Cuando existe observación parcial, ya sea por oclusión o cualquier otro motivo, deja de ser cierto que existe una política Markovina (es decir, sin memoria) que resuelva de forma óptima la tarea:

$$\pi_*(a_{t+1}|o_{t+1}) \ll \pi_*(a_{t+1}|o_0, o_1, o_2, \cdots, o_t).$$

De hecho, se suele dar que la mejor *memoryless policy* $\pi_*(a_{t+1}|o_{t+1})$ es mucho peor que la mejor *history-based policy* $\pi_*(a_{t+1}|o_0, \cdots, o_t)$. Pero eso no es todo. Cuando tenemos observación parcial las dinámicas del ambiente dejan de ser Markovianas desde el punto de vista del agente. Es decir:

$$p(o_{t+1}, r_{t+1}|o_0, a_0, o_1, a_1, \cdots, o_t, a_t) \neq p(o_{t+1}, r_{t+1}|o_t, a_t).$$

Como resultado, las ecuaciones de Bellman (y TD learning en general) se rompen.

Desde un punto de vista teórico, un problema con observación parcial equivale a utilizar una función de aproximación lineal sobre un conjunto de features $\boldsymbol{x}(s)$ que esconden parte de la información contenida en $s$. Como tal, todo teorema discutido sobre métodos aproximados aplica al caso de observación parcial.

Comienza evaluando el rendimiento de Q-learning, Sarsa y 16-step Sarsa en el dominio `InvisibleDoorEnv`. Utiliza $\gamma = 0.99$, $\alpha = 0.1$, $\epsilon = 0.01$ e inicializa los Q-values en 1. Corre cada método por 1000 episodios 30 veces (cada uno). Luego reporta el largo promedio de cada episodio. Analiza los resultados obtenidos y explica por qué son como son.

Para ayudar al agente vamos a darle un poco de memoria. Comenzaremos con la memoria más típica – conocida como *k-order memory*. Esta memoria consiste en un buffer que guarda las últimas $k$ observaciones. Luego el estado del agente pasa a ser las $k$ observaciones que se encuentran en el buffer. Por ejemplo, una *2-order memory* significa que el agente aprenderá una política $\pi(a_t|o_{t-1}, o_t)$.

Vuelve a correr Q-learning, Sarsa y 16-step Sarsa pero esta vez utiliza una *2-order memory*. Esta memoria ya viene implementada en el código base. Para usar una *2-order memory* puedes utilizar el siguiente código:

```
memory_size = 2
env = InvisibleDoorEnv()
env = KOrderMemory(env, memory_size)
```

En más detalle, las memorias funcionan como `wrappers` sobre ambientes. Reciben un ambiente en su constructor y redefinen las funciones `reset()` y `step(action)` para incluir la memoria.

```
class KOrderMemory(AbstractEnv):

    def __init__(self, env, memory_size):
        self.__env = env
        self.__memory_size = memory_size
        self.__memory = None
```

```
    def reset(self):
        s = self.__env.reset()
        self.__memory = deque(maxlen=self.__memory_size)
        self.__memory.append(s)
        return tuple(self.__memory)

    def step(self, action):
        s, r, done = self.__env.step(action)
        self.__memory.append(s)
        return tuple(self.__memory), r, done
```

La ventaja de modelar las memorias como wrappers es que es fácil combinarlas con cualquier ambiente. Además, para el agente le es indiferente si interactúa con un ambiente con o sin memoria. Pues sigue seleccionando acciones a partir de observaciones. La diferencia está en que si el ambiente incluye una $k$-order memory el estado retornado por el ambiente pasa a ser una tupla con las últimas $k$ observaciones.

En tus experimentos con 2-order memory usa los mismos parámetros y reporta los resultados de la misma forma que en el experimento sin memoria. Comenta tus resultados y explica por qué hacen sentido. En particular, explica por qué una 2-order memory es suficiente (o insuficiente) para resolver este problema.

Repite el experimento anterior pero utilizando una *binary memory* en vez de una k-order memory. Las memorias binarias consisten en $n$ bits que dejamos que el agente cambie utilizando un conjunto separado de acciones. Para usar una memoria binarias de 1 bit puedes hacer lo siguiente:

```
num_of_bits = 1
env = InvisibleDoorEnv()
env = BinaryMemory(env, num_of_bits)
```

A diferencia de las k-order memories, las binary memories también cambian el espacio de acciones. Las acciones del agente pasan a ser una tupla $(a, w)$ donde $a$ es la acción que se ejecuta en el ambiente y $w$ es el nuevo valor que tendrá la memoria binaria.

En tu experimento, utiliza una memoria de 1 bit. Comenta tus resultados y explica por qué hacen sentido. En particular, explica por qué esta memoria es suficiente (o insuficiente) para resolver este problema.

Finalmente, debes implementar un nuevo tipo de memoria. Esta memoria se podría considerar una variante de una k-order memory. El problema de las k-order memories es que no pueden recordar eventos que ocurrieron hace más de $k$ observaciones. Para remediar esto, dejaremos que el agente decida si quiere guardar (o no) la observación actual en su *k-order buffer*.

En más detalle, le daremos un buffer de tamaño $k$ al agente. Además le daremos un acción extra que le permitirá decidir si quiere guardar (o no) la observación actual en el buffer. Es decir, ahora las acciones del agente serán una tupla $(a, w)$, donde $a$ es la que se ejecuta en el ambiente y $w$ puede ser "*save*" o "*ignore*". Si el agente selecciona *save* la observación actual (i.e., antes de ejecutar la acción en el ambiente) es guardada en el buffer. Si el buffer está lleno, se elimina la observación más antigua guardada (igual que en una $k$-order memory). Si la acción seleccionada por el agente es "*ignore*" entonces no se hace nada respecto al buffer. La observación entregada al agente será igual a la observación actual del ambiente concatenada con todas las observaciones que se encuentren en el buffer.

Repite el mismo experimento anterior pero esta vez utilizando esta nueva forma de memoria con un buffer de tamaño 1. Comenta tus resultados y explica por qué hacen sentido. En particular, explica por qué esta nueva memoria es suficiente (o insuficiente) para resolver este problema.

Considerando todos los experimentos, responde las siguientes preguntas:

- ¿Existió alguna memoria que sea mejor que otra? ¿A qué se debe esto?
- ¿Existió algún algoritmo de RL que sea mejor resolviendo estos problemas? ¿A qué se debe esto?