DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Aprendizaje por Refuerzos

2012 **Aprendizaje por Refuerzos**

Integrante	LU	Correo electrónico
De Sousa Mariano	389/08	marian_sabianaa@hotmail.com
Mariano Bianchi	92/08	marianobianchi08@gmail.com
Pablo Brusco	527/08	pablo.brusco@gmail.com

1. Problema

Para este trabajo, se utilizo una versión simplificada del Tower Blocks, el cual se basa en un edificio de bloques al que hay que agregarle nuevos pisos utilizando una grúa. La dificultad radica en que el jugador no puede manejar la grúa, solo puede ejecutar la acción de soltar un nuevo piso sobre el edificio ya construido. De acuerdo a la presición con que se deposite el piso, la torre gana o pierde estabilidad.

Ya que el juego tiene una gran variedad de niveles, se opto por un modelo simple en el cual tenemos que agregar pisos (bloques) a un edificio intentando llegar a una cierta cantidad y sin haber hecho que se derrumbe. Para ello existe una grúa que se mueve de manera **constante** sobre el edificio y nos permite ejecutar la acción de tirar o no tirar.

2. Ambiente

En esta etapa se implementaron todos los objetos necesarios para que el ambiente sea lo más similar al juego real. Estos objetos son:

- Una grúa que tiene un movimiento lineal entre dos posiciones (en este caso -49 y +49), a velocidad constante (1). Se mueve en ambas direcciones (izquierda y derecha). Es la encargada de sostener los pisos que se van a ir agregando al edificio. El agente le dará la orden de sostener o tirar el piso en un momento dado.
- Un edificio que posee un movimiento pendular que va variando en velocidad de acuerdo a que tan desbalanceado está. El desbalanceo está dado por que tan cerca o lejos del centro del edificio fueron cayendo los pisos arrojados por el agente desde la grúa. Por la forma en que lo modelamos, cuanto más alto sea el edificio, mayor estabilidad tendrá y por lo tanto será más difícil que se derrumbe.

El ambiente, mantiene un estado visible para el exterior que contiene, la posición y velocidad de la torre, y además, la posición y dirección de la grúa.

Dados los parametros que mantiene el ambiente, la cantidad de estados está dada por:

$$\#estados = \#posiciones\_grua * \#direcciones\_grua * \#posiciones\_torre * \#velocidades\_torre$$

Que dada la configuración que se utilizo, serian:

$$\#estados = 99 * 2 * 99 * 11 = 215622$$

Aunque no todos ellos son alcanzables dependiendo de otro factor. Este otro factor es el ángulo máximo que se admite que tenga la torre antes de colapsar y que fuimos variando para encontrar buena dinamica, dejandolo en 30 grados.

3. Agentes

Para implementar los jugadores, utilizamos 2 tipos de agentes. Un agente que aprende utilizando la tecnica de Q-Learning y otro que aprende usando el algoritmo Sarsa-λ (ambos utilizando los algoritmos vistos en clase).

Estos agentes, reciben estímulos según sus posibles acciones:

- Tirar
- Pasar

En donde el ambiente puede responder con distintos refuerzos que contemplaban los siguientes casos:

- Refuerzo por pasar (negativo chico)
- Refuerzo por tirar y no golpear a la torre (negativo medio)
- Refuerzo por pegarle a la torre y lograr que se caiga (negativo grande)
- Refuerzo por tiro exitoso (golpeó la torre y se agregó un nuevo piso)

3.1. Experimentos

Para responder a las distintas preguntas que uno se puede plantear con respecto a los algoritmos y sus parámetros, corrimos los siguientes experimentos, siempre el objetivo era llegar a los 20 pisos

Agente	ϵ	γ	α	Pisos desde
Q-Learning	0.001	0.8	0.7	1
Q-Learning	0.001	0.8	0.7	10
Q-Learning	0.001	0.8	0.7	20
Q-Learning	0.001	0.8	0.7	30
Q-Learning	0.001	0.8	0.7	50
Q-Learning	0.001	0.8	0.7	70

Cuadro 1: Distintas estabilidades en Q-Learning

Agente	ϵ	γ	α	λ	Pisos desde
Sarsa-λ	0.001	0.8	0.7	0.3	1
Sarsa-λ	0.001	0.8	0.7	0.3	10
Sarsa-λ	0.001	0.8	0.7	0.3	20
Sarsa-λ	0.001	0.8	0.7	0.3	30
Sarsa-λ	0.001	0.8	0.7	0.3	50
Sarsa-λ	0.001	0.8	0.7	0.3	70

Cuadro 2: Distintas estabilidades en Sarsa-λ

Luego de estos experimentos fijamos el piso desde el cual se comienza en 50.

Agente	ϵ	γ	α	λ
Q-Learning	0.001	0.8	0.7	-
Sarsa-λ	0.001	0.8	0.7	0.001
Sarsa-λ	0.001	0.8	0.7	0.3
Sarsa-λ	0.001	0.8	0.7	0.7

Cuadro 3: Qlearning vs Sarsa-λ

Agente	ϵ	γ	α
Q-Learning	0	0.8	0.7
Q-Learning	0.0001	0.8	0.7
Q-Learning	0.001	0.8	0.7
Q-Learning	0.01	0.8	0.7

Cuadro 4: Epsilons en Qlearning

Agente	ϵ	γ	α
Q-Learning	0.001	0.8	0
Q-Learning	0.001	0.8	0.2
Q-Learning	0.001	0.8	0.5
Q-Learning	0.001	0.8	0.8

Cuadro 5: Alphas en Qlearning

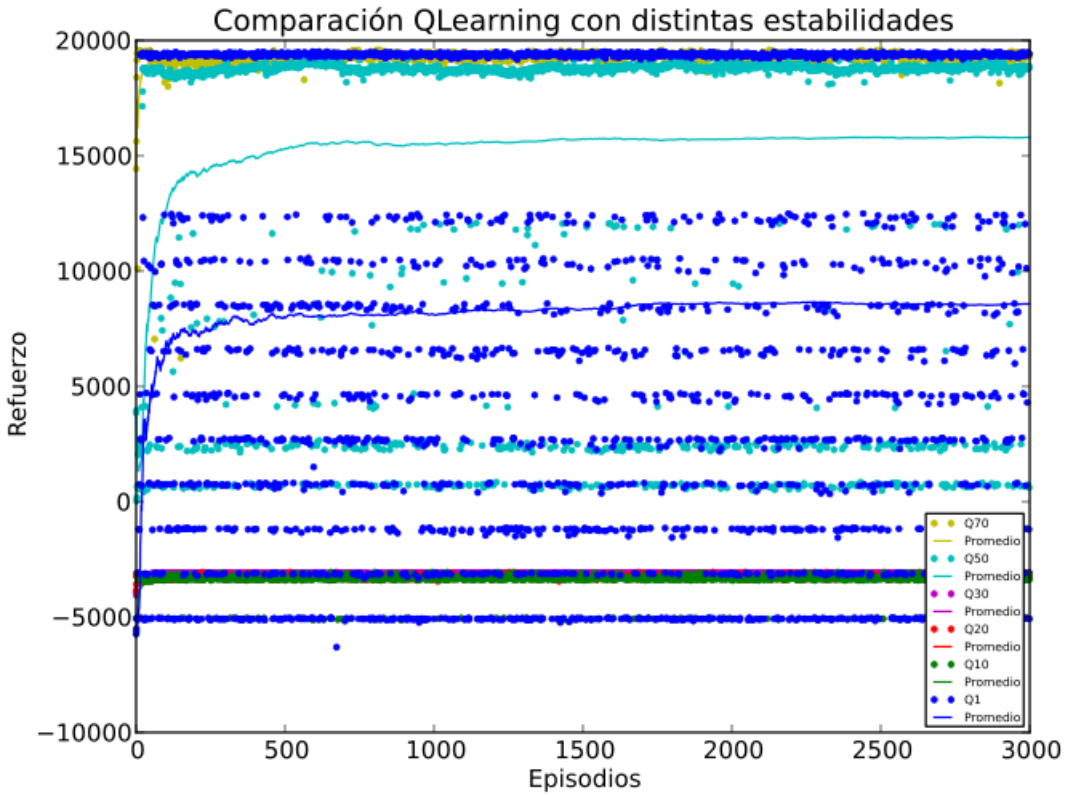
Agente	ϵ	γ	α
Q-Learning	0.0001	0.01	0.7
Q-Learning	0.0001	0.1	0.7
Q-Learning	0.0001	0.5	0.7
Q-Learning	0.0001	0.9	0.7

Cuadro 6: Gammas en Qlearning

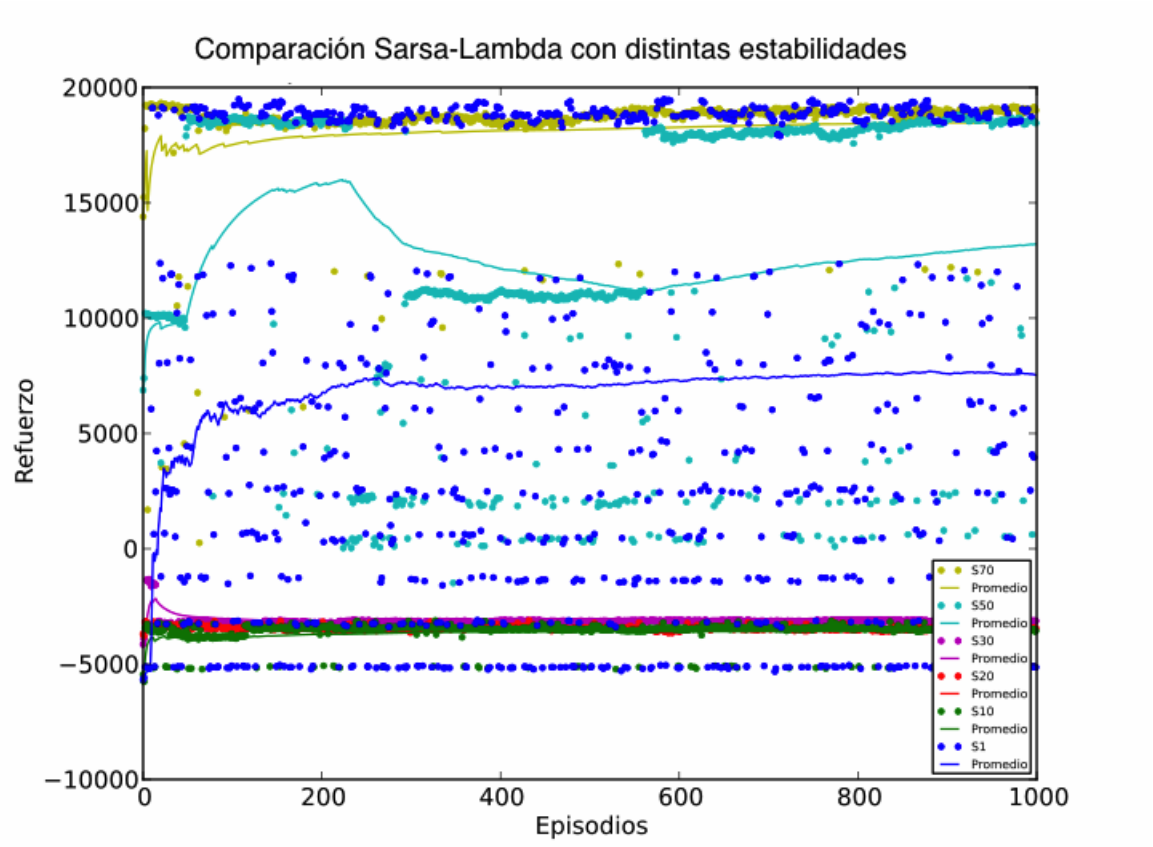
3.2. Resultados

Los siguientes resultados son en base a corridas en donde los refuerzos eran los siguientes:

- Refuerzo por pasar = -1
- Refuerzo por tirar y no golpear a la torre = -30
- Refuerzo por pegarle a la torre y lograr que se caiga = -5000
- Refuerzo por tiro exitoso (golpeó la torre y se agregó un nuevo piso) = 2000

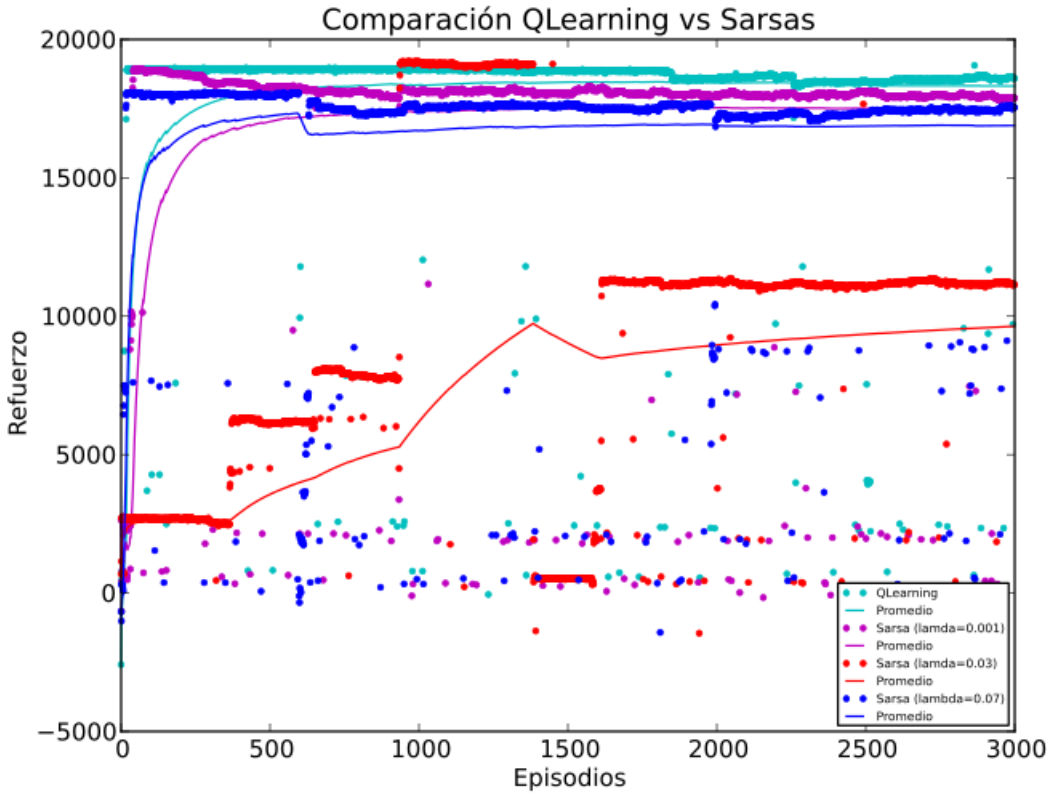


Como se puede ver en este primer resultado, a medida que el piso desde donde arranca a jugar aumenta, la dificultad disminuye, esto es adecuado ya que el péndulo que modelamos funciona de esa manera.

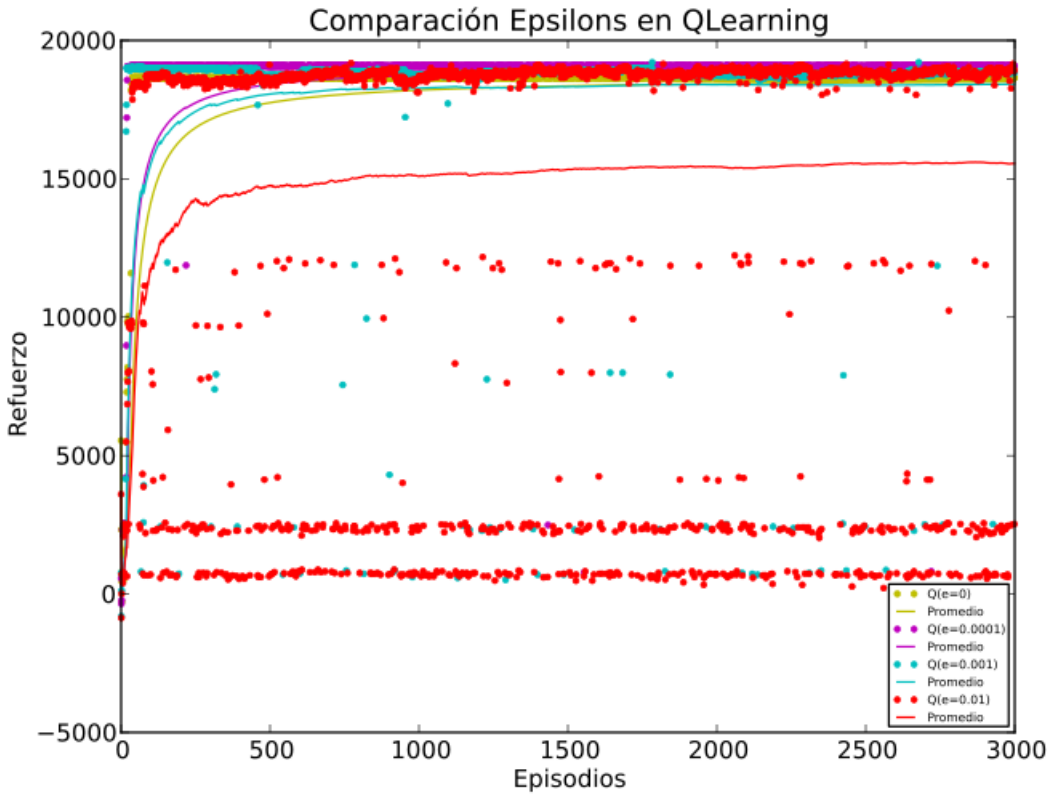


Con Sarsa-λ ocurre algo similar, aunque con una mayor variabilidad, en especial si se juega desde el piso 50. En este resultado podemos observar un comportamiento extrañõ del algoritmo, en el cual parece haberle dado importancia a una acción erronea luego de haber obtenido buenos resultados (ver S50)

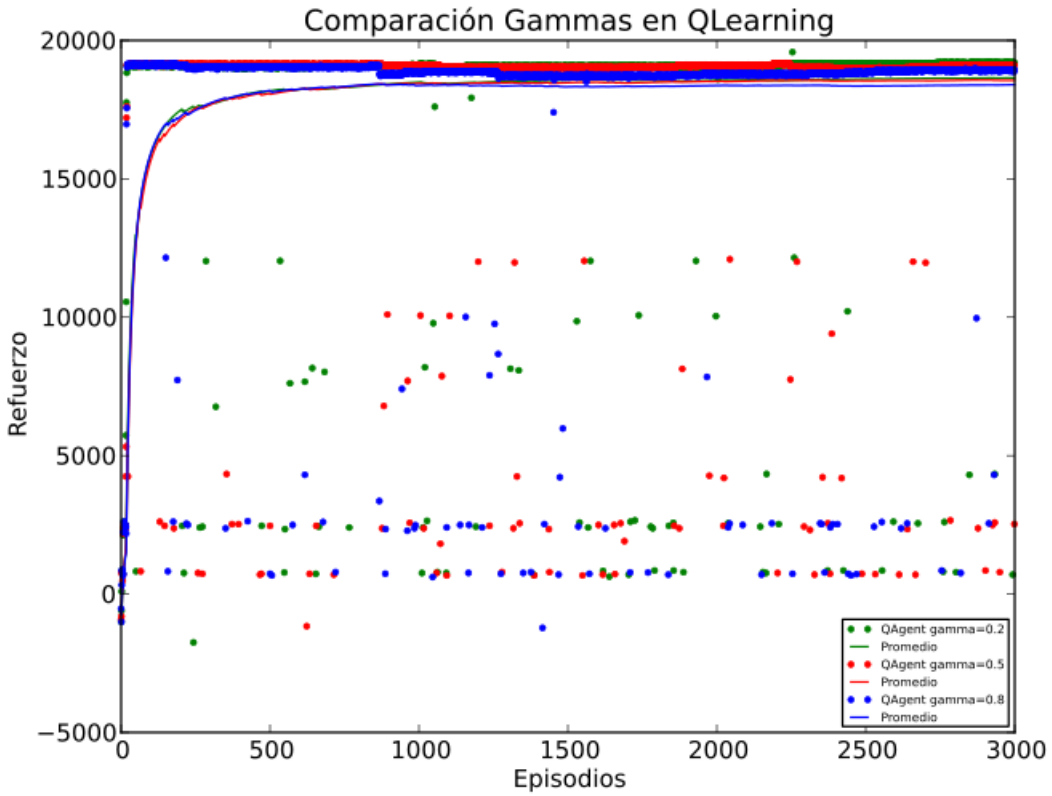
Recordar que los próximos resultados se obtuvieron jugando a partir del piso 50 y hasta lograr 20 pisos.



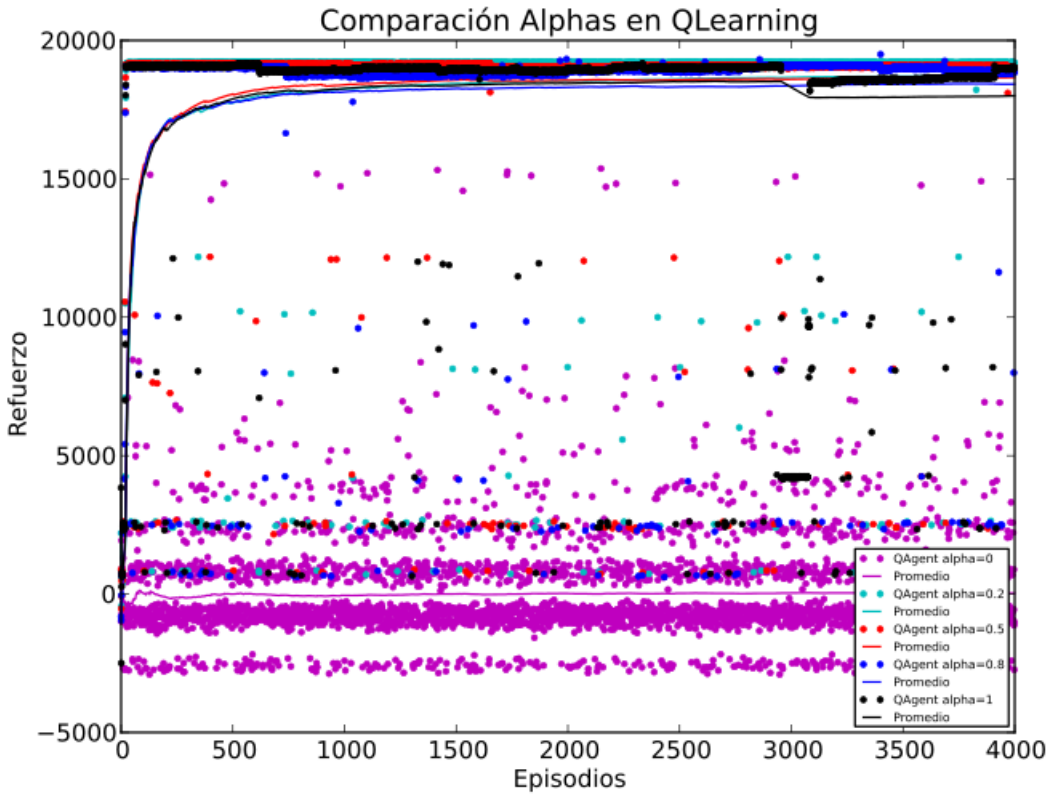
El comportamiento de los algoritmos en este caso fue un poco caótico, en especial Sarsa con $\lambda = 0.03$, en donde se alcanzo el valor optimo durante un tiempo, pero luego volvió a bajar a la mitad. Por otro lado, Q-Learning se mantuvo con muy buenos resultados.



En este experimento, se ve que mientras epsilon es mas chico, funciona mejor (con resultados muy parecidos para epsilons muy chicos). Esto puede deberse a la esencia del problema, el cual parece funcionar bien con algoritmos semi-golosos, en donde un poco de aleatoriedad ayuda, pero en cantidades mas grandes parece perjudicar. Creemos que esto tiene que ver con la forma del problema en donde una vez que se consigue agregar un bloque, conviene seguir prestando atención al mismo tipo de estados, y no investigar nuevos.



En este ejemplo, y dados los resultados anteriores (en donde parece que un algoritmo goloso es la mejor solución) vemos que los cambios en γ no afectan al resultado.



Al igual que el ejemplo anterior, el factor α no afecta en gran medida, excepto cuando el 0, lo cual tiene sentido ya que un α tendiendo a cero significa no darle importancia al aprendizaje calculado.

3.3. Conclusiones

Luego de analizar los resultados, llegamos a la conclusión de que resulto fácil para el agente implementado usando Q-Lerning encontrar soluciones al problema cuando la estabilidad del edificio ayuda, ya que el problema parece solucionarse de buena manera con algoritmos Golosos.

Por otro lado, el algoritmo Sarsa-λ parece no funcionar muy bien para este tipo de problemas, ademas de demorar mucho más (hasta 10 veces más).

Algoritmicamente, el desafió mas grande fue modelar un ambiente que modele el juego con algo de realismo. Queda para trabajo futuro, intentar diseñar una interfaz gráfica que muestre como se comportan los algoritmos e intentar aplicar otros algoritmos, quizás mas directos para ver como funcionan.

4. Anexo A

4.1. Código de Q-Learning

```
def run_episode(self):
    state = self.environment.start()
    rewards = 0
    movements = 0

    while not (state.has_finished()):

        #usando una politica derivada de Q (eps-greedy en este caso)
        action = self.choose_action(state)

        new_state, reward = self.environment.make_action(action)
        max_action = self.max_action(new_state)
        key = (state,action)
        self.set_q_value(
            key,
            (1-self.alpha) * self.q_value((state,action)) +
            self.alpha * (reward + self.gamma *
                                self.q_value((new_state, max_action)))
        )

        state = deepcopy(new_state)
        rewards += reward
        movements += 1
    return rewards
```

4.2. Código de Sarsa-Lambda

```
def run_episode(self):
    state = self.environment.start()
    rewards = 0
    movements = 0

    #usando una politica derivada de Q (eps-greedy en este caso)
    action = self.choose_action(state)

    while not (state.has_finished()):
        new_state, reward = self.environment.make_action(action)
        new_action = self.choose_action(new_state)
        delta = reward + self.gamma * self.q_value((new_state, new_action))
                            - self.q_value((state,action))

        self.set_e_value((state,action),1.0)
        for k in self.elegibilities.keys():
```

```
            self.set_q_value(
                k,
                self.q_value(k) + self.alpha*delta*self.e_value(k)
            )

            self.set_e_value(
                k,
                self.e_value(k)*self.gamma*self.lambda_val
            )

        state = deepcopy(new_state)
        action = new_action
        rewards += reward
        movements += 1

    return rewards
```