DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Aprendizaje por Refuerzos

2012 **Aprendizaje por Refuerzos**

Integrante	LU	Correo electrónico
Mariano De Sousa	??	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 y ¿variaciones?, 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

Para la implementación del ambiente La grúa es un objeto que tiene un movimiento lineal entre dos posiciones (en este caso -49 y +49), a velocidad constante (1) se mueve en una u otra dirección, de donde se tirará el nuevo piso a agregar.

Por otro lado, el edificio, posee un movimiento pendular que va variando en velocidad de acuerdo a que tan desbalanceado está, debido a los tiros previos del jugador.

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, recibían estímulos según sus posibles acciones:

- Tirar
- Pasar

En donde el ambiente podía 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)

Veamos algunas comparaciones en función de ciertos parametros que fuimos variando para estudiar

3.1. Experimentos

Para todos los experimentos normalizamos el ambiente para que sea un edificio de 20 pisos al que hay que llegar como objetivo del juego.

Para responder a las distintas preguntas que uno se puede plantear con respecto a los algoritmos y sus parametros, corrimos los siguientes experimentos:

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 1: 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 2: Epsilons en Qlearning

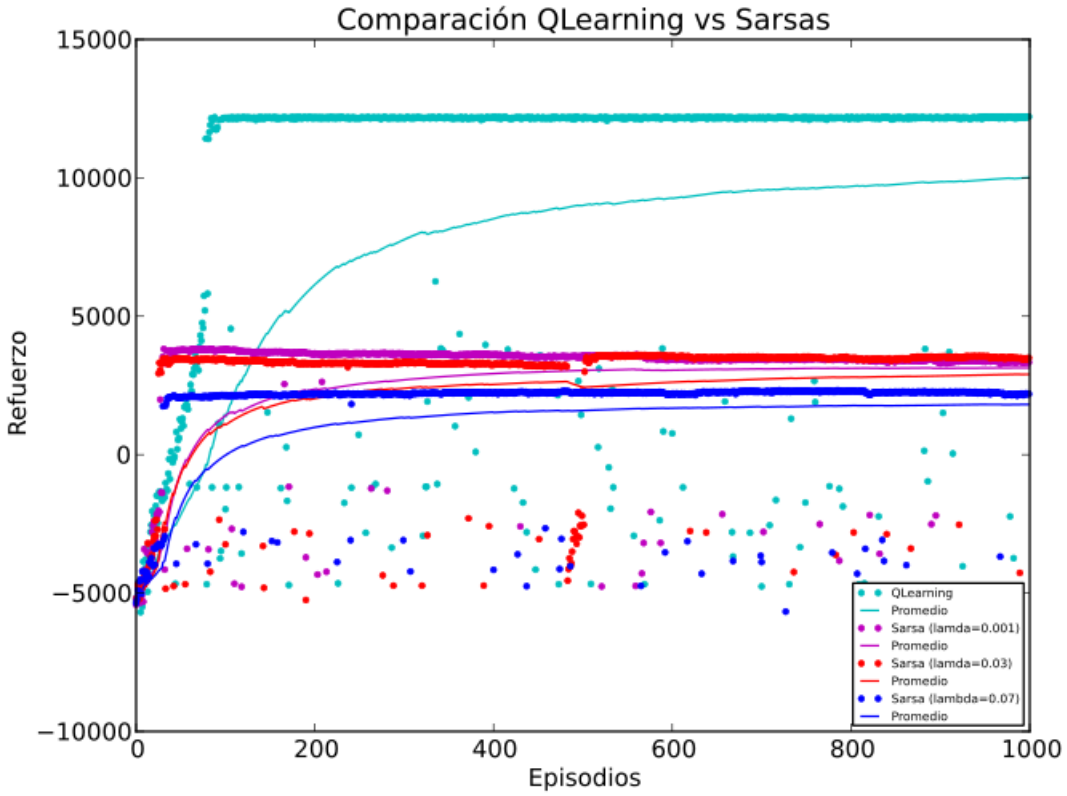
Agente	ϵ	γ	α
Q-Learning	0.001	0.8	0
Q-Learning	0.001	0.8	0.2
Q-Learning	0.001	0.8	0.5
Q-Learning	0.001	0.8	0.8

Cuadro 3: Gammas en Qlearning

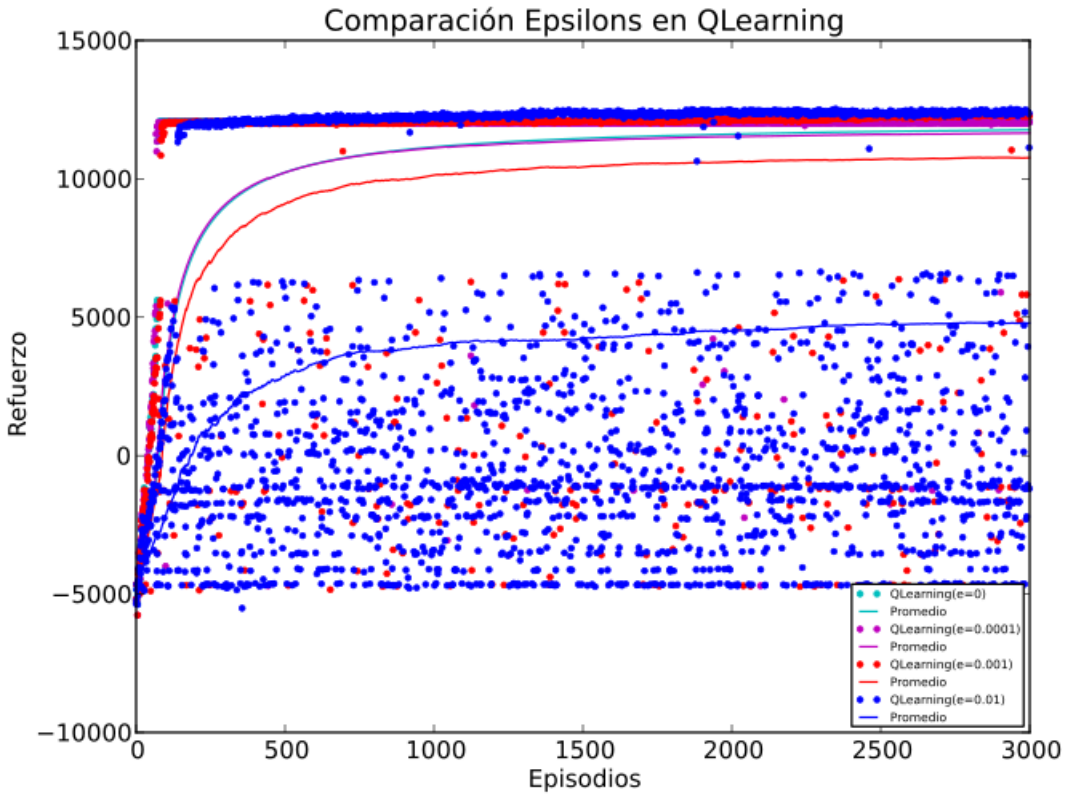
Agente	ϵ	γ	α
Q-Learning	0.0001	0.01	0.7
Q-Learning	0.0001	0.1	0.7
Q-Learning	0.0001	0.5	0.7
Q-Learning	0.0001	0.9	0.7

Cuadro 4: Gammas en Qlearning

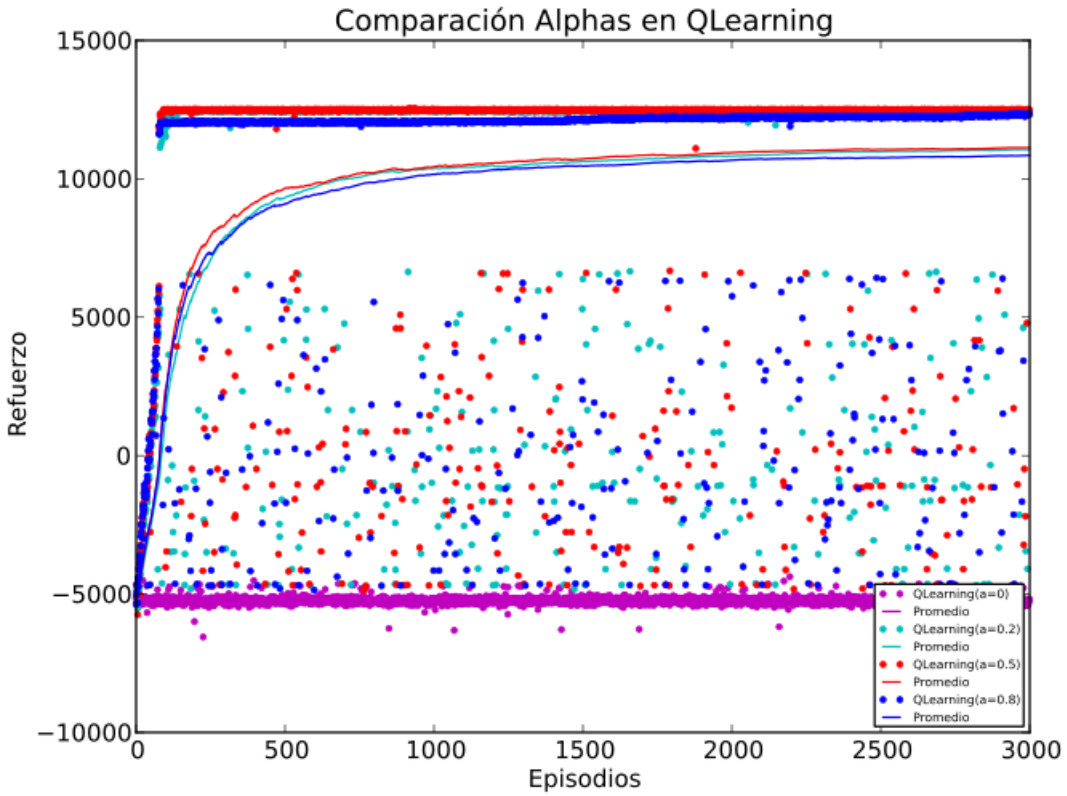
3.2. Resultados y Conclusiones



Analizando este gráfico podemos ver que Q-learning tuvo resultados muy por encima de Sarsa-λ para distintos lambdas, y consistentemente con este resultado, un lambda mas chico influye en un mejor resultado.



Ya que un epsilon mas chico significa proximidad a un algoritmo *Goloso*, y dado que mientras mas Goloso, dio mejores resultados, podemos comenzar a pensar que este problema funcionaría bien con este tipo de soluciones. Es decir, para el agente que se acerca a mejores resultados, no fue tan necesario explorar, sino explotar las soluciónes encontrada. Quizás esto nos de una pista de por que funciona mejor Qlearning que Sarsa-λ .



En este resultado, podemos ver que el factor α, que determina el peso que se le da al aprendizaje, no fue determinante para estos resultados (exepto con $\alpha = 0$, en donde esto significa que no aprende nada).

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() or movements > 300):
        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
```