Laboratorio 6

Task 1

1. Este es un método que acelera la convergencia hacia la política óptima al enfocarse en actualizar primero los estados más relevantes. En ambientes determinísticos, donde las transiciones entre estados son predecibles, asigna prioridades a los estados según cuánto cambiarían sus valores si se actualizan. Aquellos estados con mayores prioridades se actualizan antes, utilizando una cola de prioridades que guía el proceso de aprendizaje.

2. Este es un método en el cual el agente recolecta muestras de su interacción con el ambiente, conocidas como trayectorias, para aprender y mejorar su política. Una trayectoria es una secuencia de estados, acciones y recompensas que el agente experimenta. En lugar de evaluar todas las posibles acciones en cada estado, el agente utiliza estas trayectorias para actualizar sus estimaciones de valores o mejorar su política.

3. Este es un algoritmo que se utiliza en la búsqueda de Monte Carlo para la toma de decisiones en tiempo real, como en juegos de estrategia. UCT equilibra la exploración y la explotación mediante el uso del criterio de Upper Confidence Bounds. Durante la construcción del árbol de decisiones, el algoritmo selecciona qué nodos explorar basándose en una combinación del valor esperado y la incertidumbre asociada. Esto permite que UCT construya un árbol de decisiones de manera eficiente, maximizando las recompensas a largo plazo mientras explora opciones potencialmente mejores.

Task 2

In [ ]:
```
import gymnasium as gym
import math
import numpy as np
import time
import matplotlib.pyplot as plt
from collections import defaultdict
import uuid
import random
from math import sqrt, log
```

MCTS

In [ ]:
```
env = gym.make('FrozenLake-v1', is_slippery=True)
```

In [ ]:
```
import gym
import numpy as np
from collections import defaultdict

class MCTSAgent:
    def __init__(self, env, max_depth=10, n_simulations=100, gamma=0.99):
        self.env = env
        self.max_depth = max_depth
        self.n_simulations = n_simulations
        self.gamma = gamma
        self.values = defaultdict(float)
        self.visits = defaultdict(int)

    def select_action(self, state):
        def uct(state_action):
            state, action = state_action
            if self.visits[state, action] == 0:
                return float('inf')
            return self.values[state, action] / self.visits[state, action] + np.sqr
        state_action_pairs = [(state, action) for action in range(self.env.action_s
        best_action = max(state_action_pairs, key=uct)
        return best_action[1]

    def rollout(self, state):
        done = False
        total_reward = 0
        epsilon = 0.5  # Ajusta el valor de epsilon según sea necesario

        for steps in range(self.max_depth):
            if np.random.random() < epsilon:
                action = self.env.action_space.sample()  # Exploración
            else:
                # Explotación basada en el valor actual de Q
                action = np.argmax([self.values[(state, a)] for a in range(self.env

            state, reward, done, _, _ = self.env.step(action)
            total_reward += reward * (self.gamma ** steps)
            if done:
                break

        return total_reward

    def backpropagate(self, state_action_sequence, total_reward):
        for i in range(len(state_action_sequence)):
            state, action = state_action_sequence[i]
            self.values[state, action] += total_reward
            self.visits[state, action] += 1
            self.visits[state] += 1
            total_reward *= self.gamma

    def run(self, state):
        for _ in range(self.n_simulations):
            state_action_sequence = []
            cur_state = state
            for __ in range(self.max_depth):
```

```
                action = self.select_action(cur_state)
                state_action_sequence.append((cur_state, action))
                cur_state, reward, done, _, _ = self.env.step(action)
                if done:
                    break
            total_reward = self.rollout(cur_state)
            self.backpropagate(state_action_sequence, total_reward)

        return self.select_action(state)
```

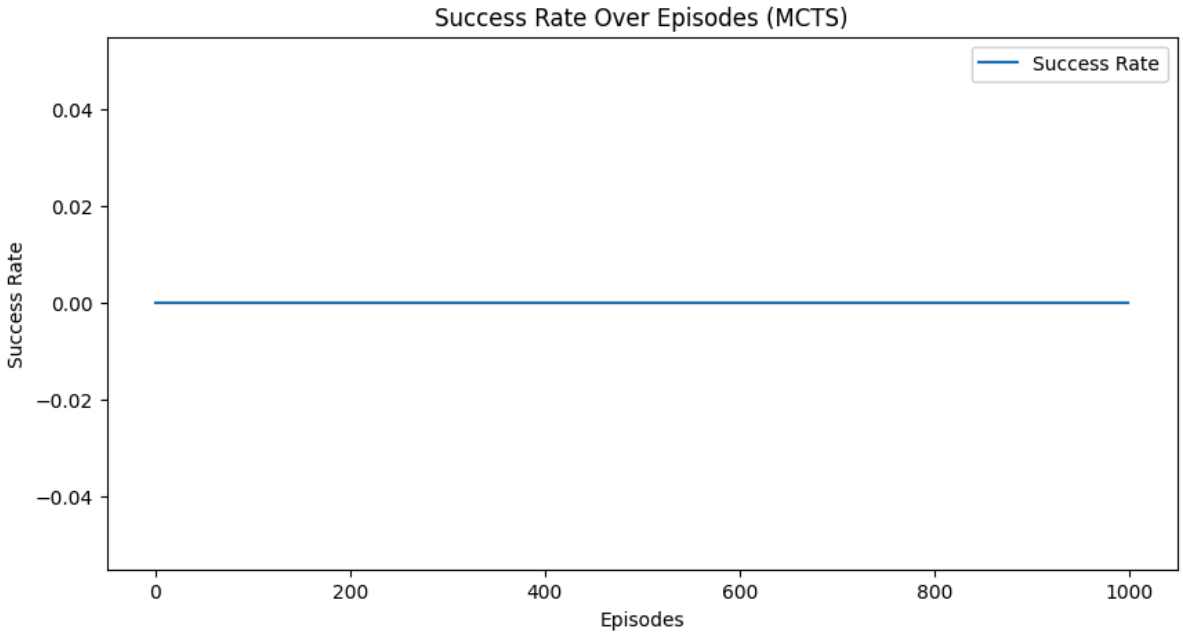
In [ ]:
```
success_rate = []
avg_reward = []
steps_to_goal = []
total_successes = 0
total_reward = 0
total_steps = 0
```

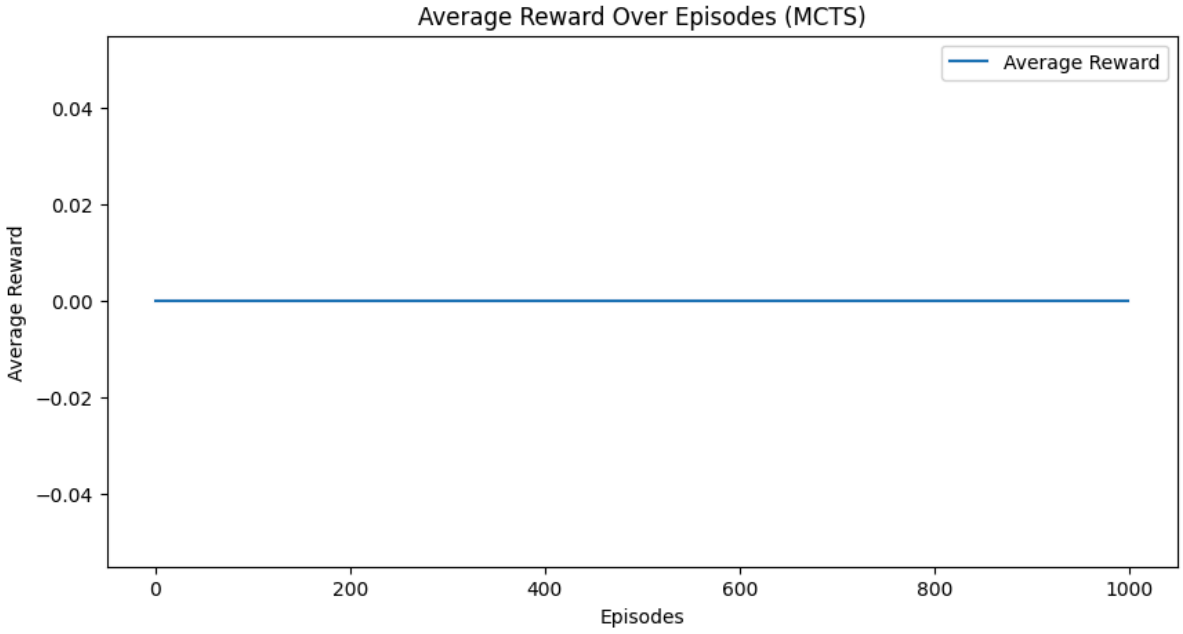
In [ ]:
```
episodes = 1000
```

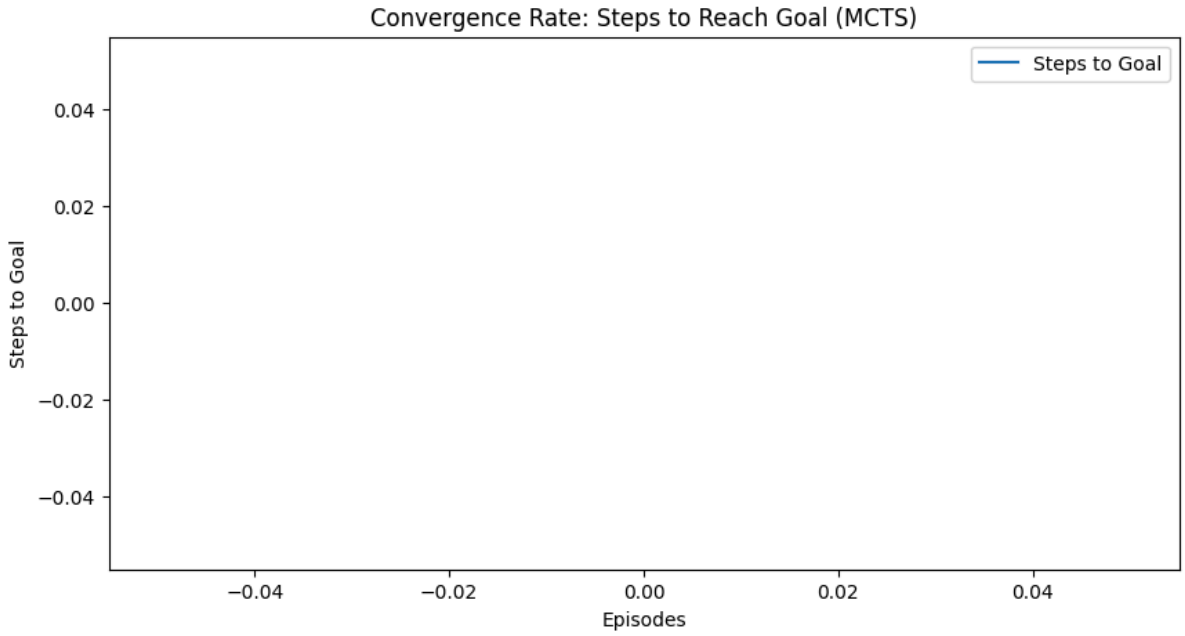
In [ ]:
```
agent = MCTSAgent(env)

for episode in range(episodes):
    state, _ = env.reset()
    done = False
    episode_reward = 0
    steps = 0
    reward = 0
    while not done:
        action = agent.run(state)
        next_state, reward, done, _, _ = env.step(action)
        state = next_state

        episode_reward += reward
        steps += 1

    total_reward += episode_reward
    total_steps += steps
    if reward == 1:
        total_successes += 1
        steps_to_goal.append(steps)
    else:
        steps_to_goal.append(0)

    success_rate.append(total_successes / (len(success_rate) + 1))
    avg_reward.append(total_reward / (len(avg_reward) + 1))
```

In [ ]:
```
plt.figure(figsize=(10, 5))
plt.plot(success_rate, label="Success Rate")
plt.xlabel("Episodes")
plt.ylabel("Success Rate")
plt.title("Success Rate Over Episodes (MCTS)")
plt.legend()
plt.show()

# ii. Recompensa promedio por episodio
```

```
plt.figure(figsize=(10, 5))
plt.plot(avg_reward, label="Average Reward")
plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("Average Reward Over Episodes (MCTS)")
plt.legend()
plt.show()

# iii. Tasa de convergencia
plt.figure(figsize=(10, 5))
plt.plot([s for s in steps_to_goal if s > 0], label="Steps to Goal")
plt.xlabel("Episodes")
plt.ylabel("Steps to Goal")
plt.title("Convergence Rate: Steps to Reach Goal (MCTS)")
plt.legend()
plt.show()
```







DynaQ+

In [ ]:
```
env = gym.make('FrozenLake-v1', is_slippery=True)
```

In [ ]:
```
def epsilon_greedy(epsilon, Q, state):
    if np.random.random() < epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(Q[state])
```

In [ ]:
```
def increment_time(model):
    for state in model:
        for act in model[state]:
            model[state][act] = (model[state][act][0], model[state][act][1], model[
```

In [ ]:
```
def dyna_Q_plus(n_planning, kappa, gamma, alpha, epsilon, episodes, env):
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    model = defaultdict(lambda: defaultdict(lambda: (0, 0, 0)))
    time = defaultdict(lambda: defaultdict(lambda: 0))

    success_rate = []
    avg_reward = []
    steps_to_goal = []
    total_successes = 0
    total_steps = 0
    total_reward = 0

    for _ in range(episodes):
        s, _ = env.reset()
        done = False
        episode_reward = 0
        steps = 0

        while not done:
```

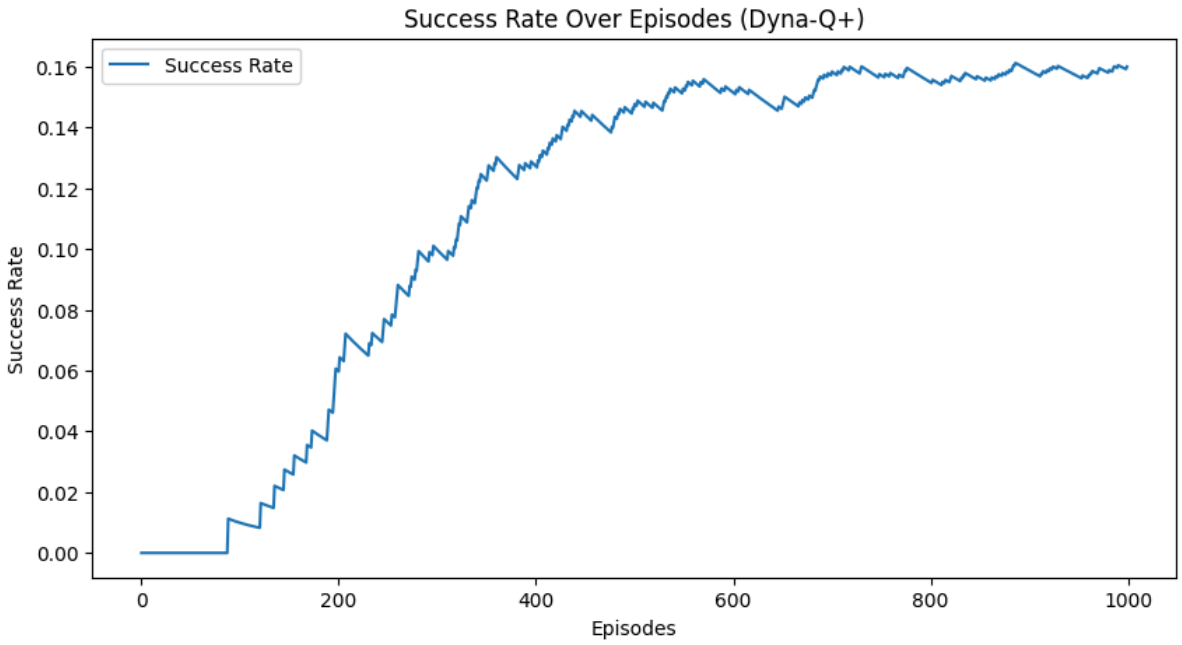
```
            action = epsilon_greedy(epsilon, Q, s)
            next_state, reward, done, _, _ = env.step(action)

            episode_reward += reward
            steps += 1

            best_action = np.argmax(Q[next_state])
            Q[s][action] += alpha * (reward + gamma * Q[next_state][best_action] -
            model[s][action] = (reward, next_state,0)

            time[s][action] = 0

            increment_time(model)

            for _ in range(n_planning):
                s = np.random.choice(list(model.keys()))
                a = np.random.choice(list(model[s]))
                r, s_, t = model[s][a]
                bonus = kappa * np.sqrt(t)
                best_action = np.argmax(Q[s_])
                Q[s][a] += alpha * (r + bonus + gamma * Q[s_][best_action] - Q[s][a

            s = next_state

        total_reward += episode_reward
        total_steps += steps

        if reward == 1:
            total_successes += 1
            steps_to_goal.append(steps)
        else:
            steps_to_goal.append(0)

        success_rate.append(total_successes / (len(success_rate) + 1))
        avg_reward.append(total_reward / (len(avg_reward) + 1))

    return success_rate, avg_reward, steps_to_goal
```

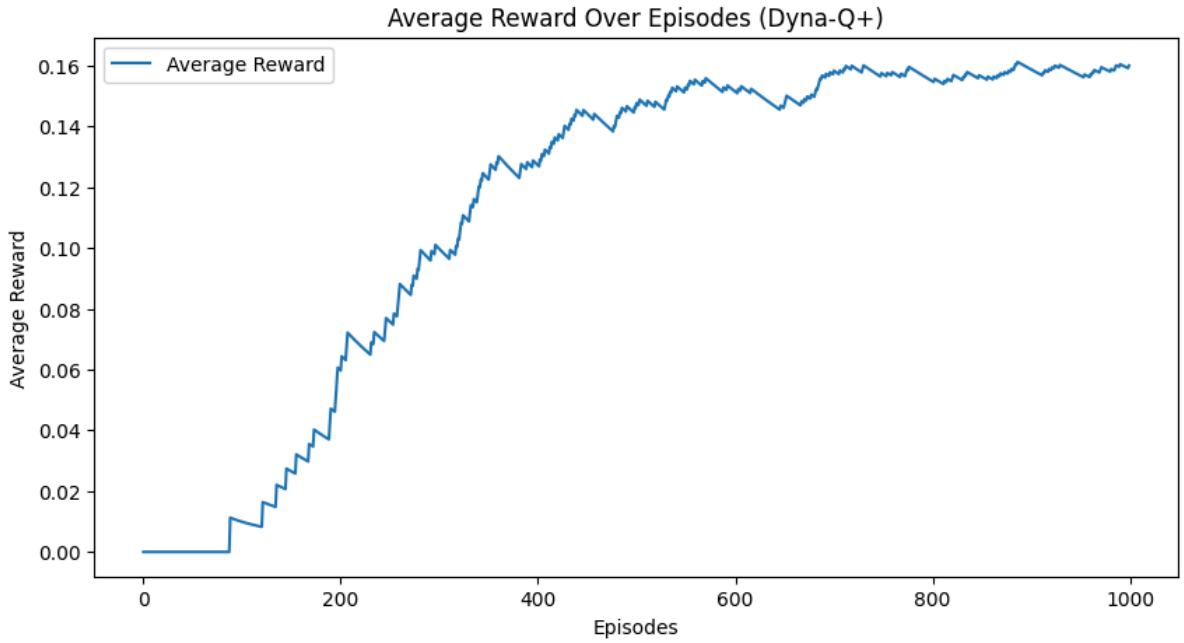
In [ ]:
```
n_planning = 5
kappa = 0.0001
gamma = 0.9
alpha = 0.1
epsilon = 0.1
episodes = 1000
```

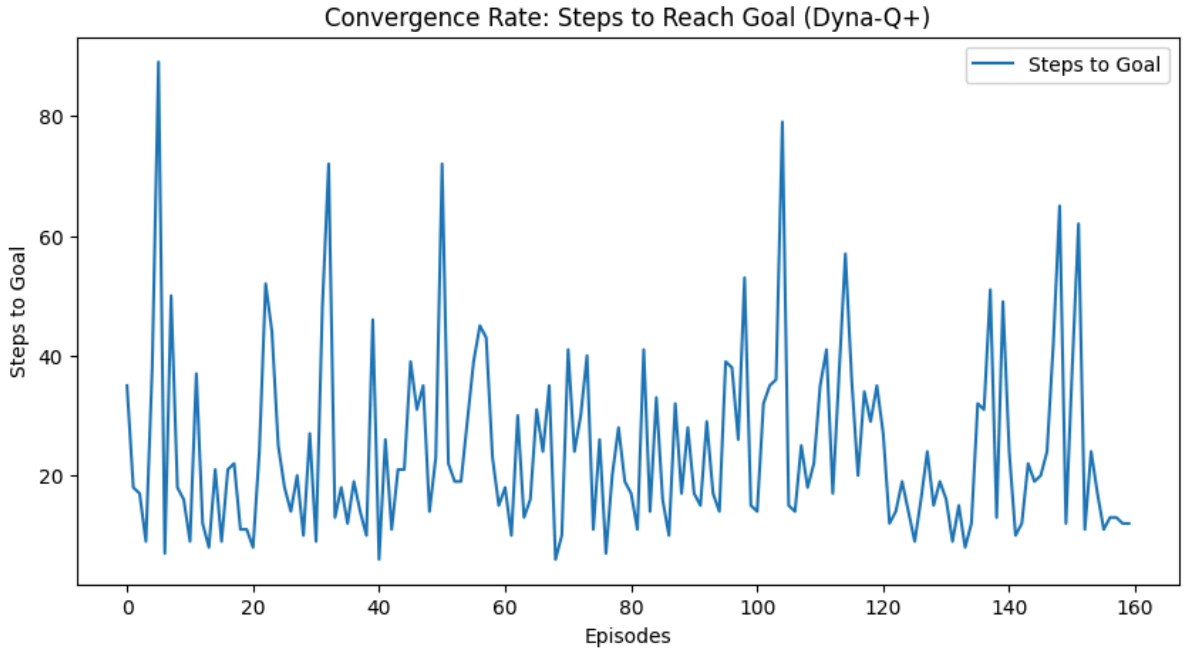
In [ ]:
```
success_rate, avg_reward, steps_to_goal = dyna_Q_plus(n_planning, kappa, gamma, alp
```

In [ ]:
```
# Gráficos
# i. Tasa de éxito en los episodios
plt.figure(figsize=(10, 5))
plt.plot(success_rate, label="Success Rate")
plt.xlabel("Episodes")
plt.ylabel("Success Rate")
plt.title("Success Rate Over Episodes (Dyna-Q+)")
plt.legend()
```

```
plt.show()

# ii. Recompensa promedio por episodio
plt.figure(figsize=(10, 5))
plt.plot(avg_reward, label="Average Reward")
plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("Average Reward Over Episodes (Dyna-Q+)")
plt.legend()
plt.show()

# iii. Tasa de convergencia
plt.figure(figsize=(10, 5))
plt.plot([s for s in steps_to_goal if s > 0], label="Steps to Goal")
plt.xlabel("Episodes")
plt.ylabel("Steps to Goal")
plt.title("Convergence Rate: Steps to Reach Goal (Dyna-Q+)")
plt.legend()
plt.show()
```



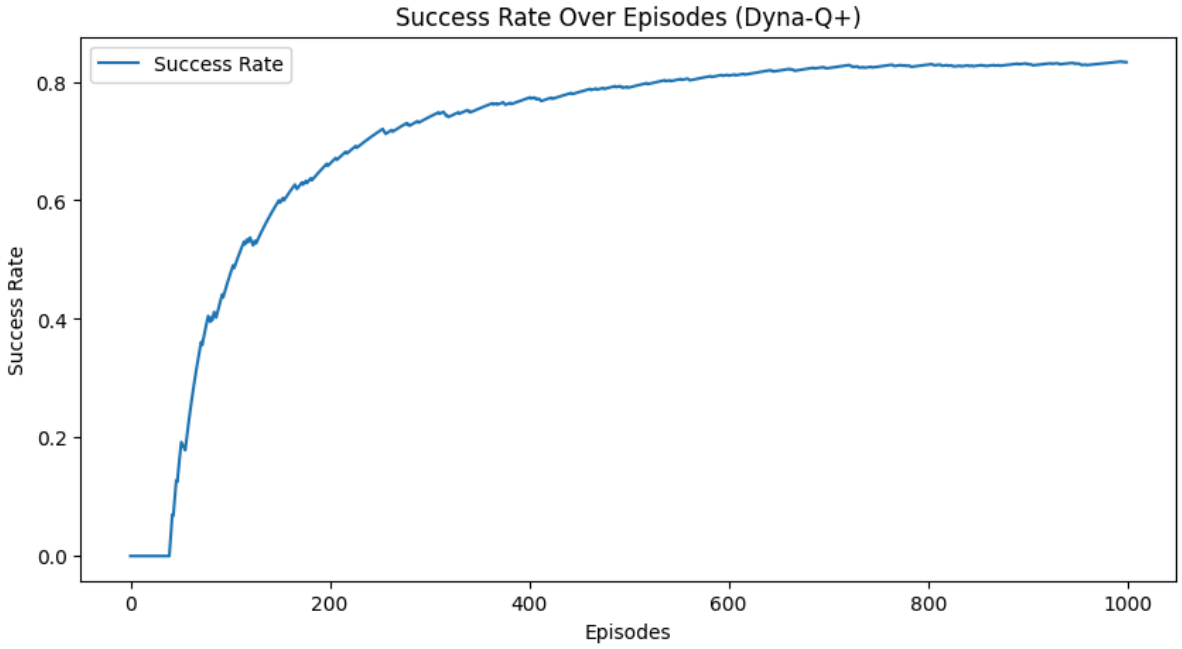


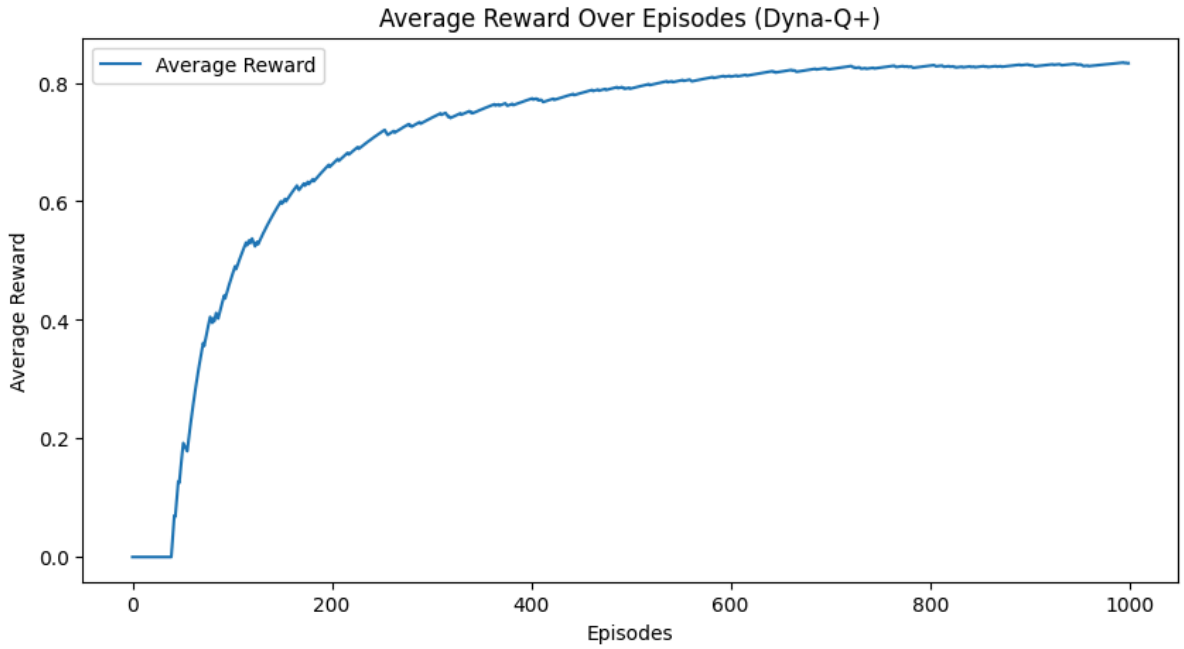


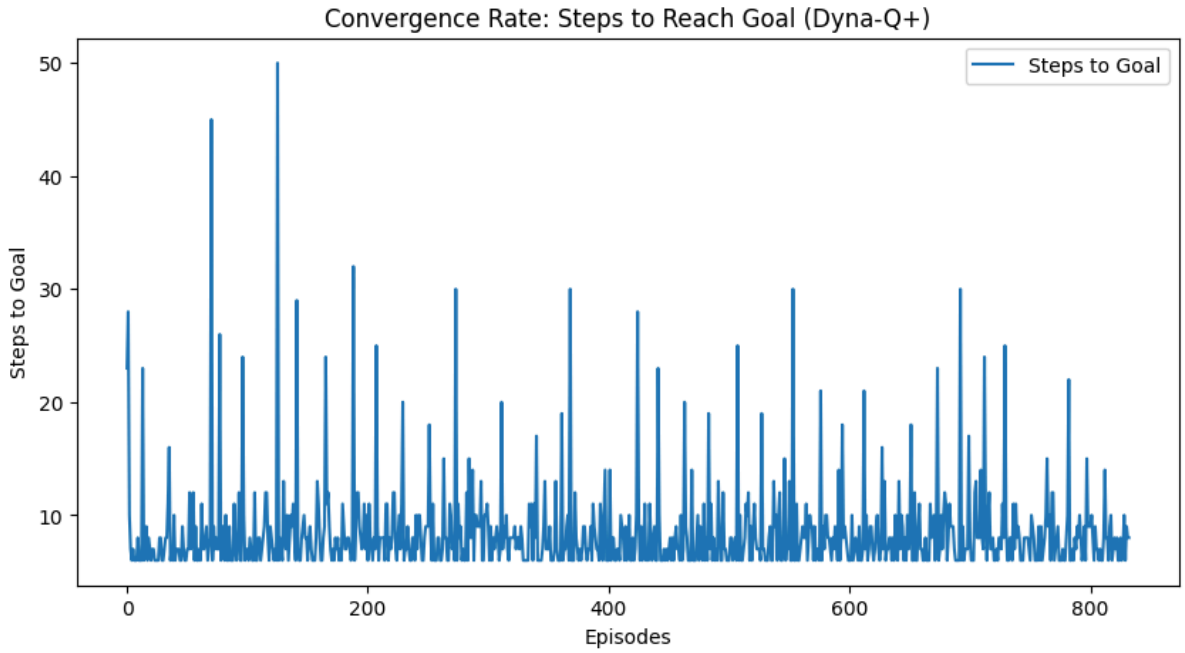
In [ ]:
```
n_planning = 10
kappa = 0.01
gamma = 0.9
alpha = 0.1
epsilon = 0.1
episodes = 1000

success_rate, avg_reward, steps_to_goal = dyna_Q_plus(n_planning, kappa, gamma, alp
```

In [ ]:
```
# Gráficos
# i. Tasa de éxito en los episodios
plt.figure(figsize=(10, 5))
plt.plot(success_rate, label="Success Rate")
plt.xlabel("Episodes")
plt.ylabel("Success Rate")
```

```
plt.title("Success Rate Over Episodes (Dyna-Q+)")
plt.legend()
plt.show()

# ii. Recompensa promedio por episodio
plt.figure(figsize=(10, 5))
plt.plot(avg_reward, label="Average Reward")
plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("Average Reward Over Episodes (Dyna-Q+)")
plt.legend()
plt.show()

# iii. Tasa de convergencia
plt.figure(figsize=(10, 5))
plt.plot([s for s in steps_to_goal if s > 0], label="Steps to Goal")
plt.xlabel("Episodes")
plt.ylabel("Steps to Goal")
plt.title("Convergence Rate: Steps to Reach Goal (Dyna-Q+)")
plt.legend()
plt.show()
```



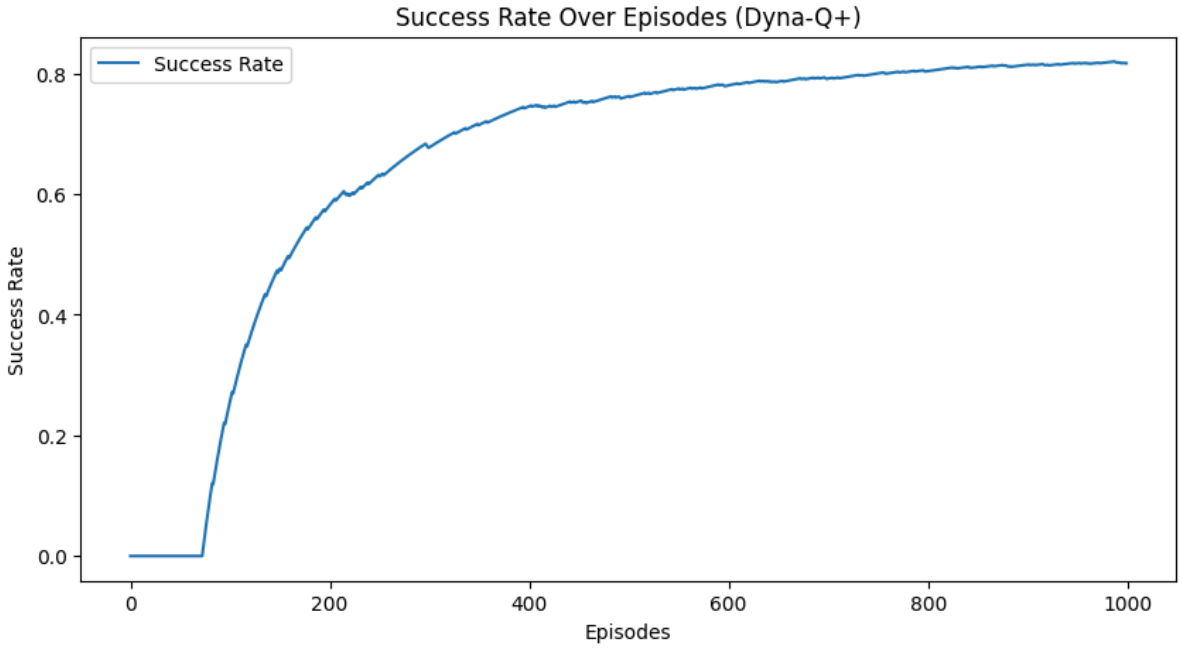


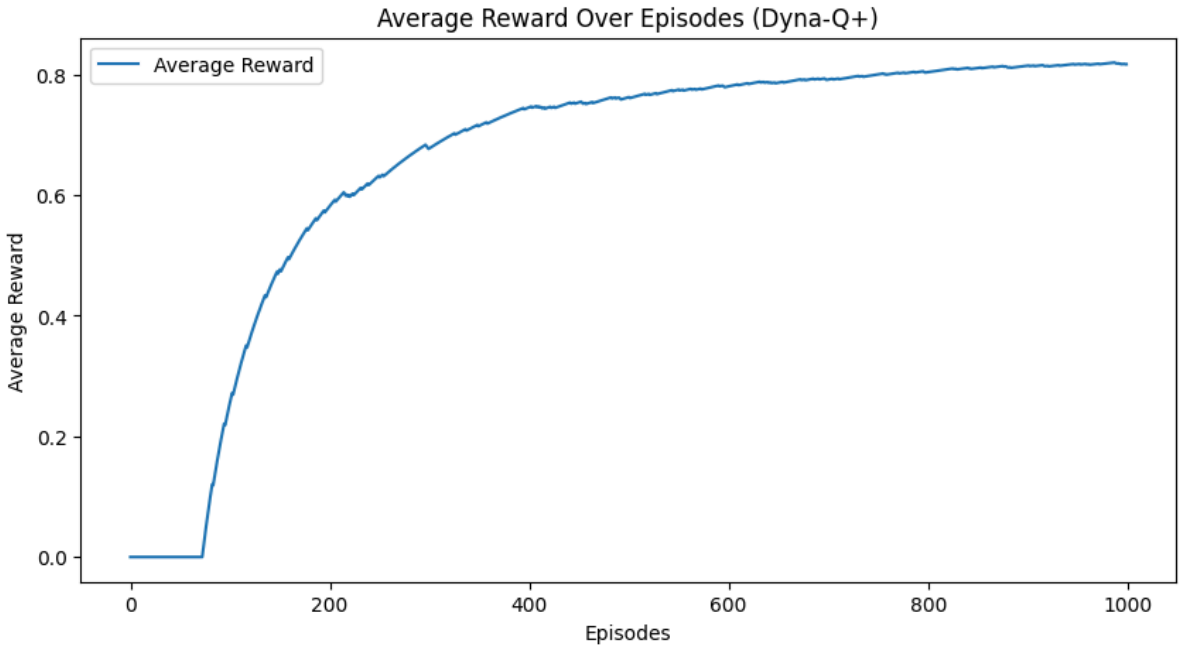


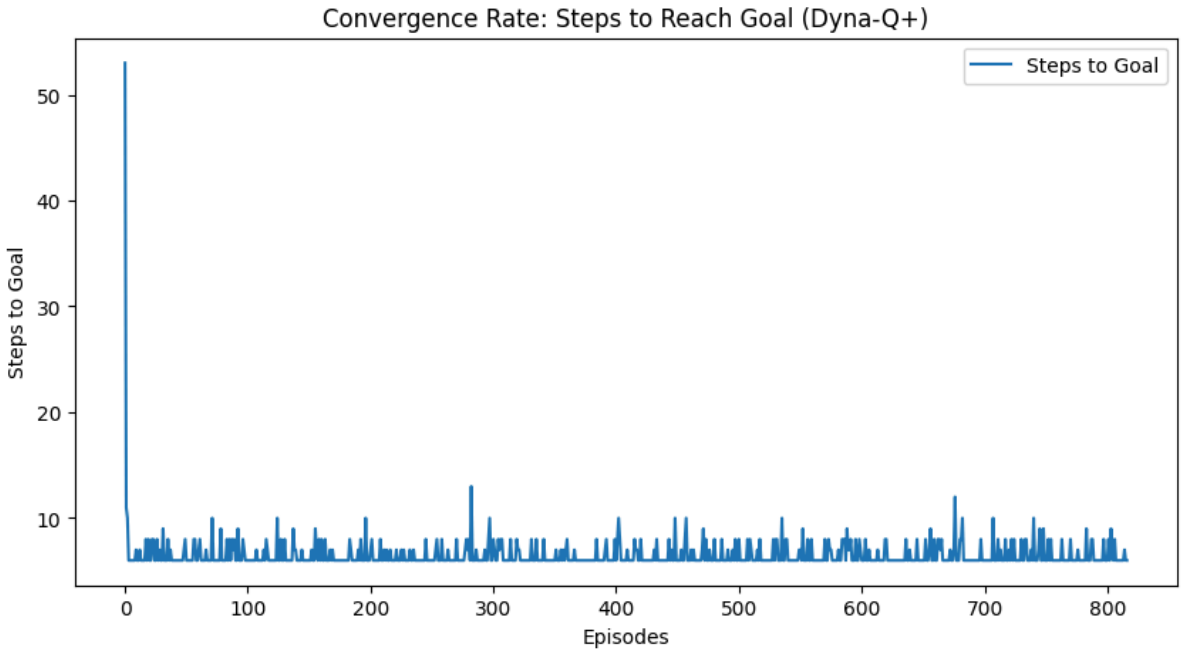
In [ ]:
```
n_planning = 20
kappa = 0.0001
gamma = 0.9
alpha = 0.1
epsilon = 0.1
episodes = 1000

success_rate, avg_reward, steps_to_goal = dyna_Q_plus(n_planning, kappa, gamma, alp
```

In [ ]:
```
# Gráficos
# i. Tasa de éxito en los episodios
plt.figure(figsize=(10, 5))
plt.plot(success_rate, label="Success Rate")
plt.xlabel("Episodes")
plt.ylabel("Success Rate")
```

```
plt.title("Success Rate Over Episodes (Dyna-Q+)")
plt.legend()
plt.show()

# ii. Recompensa promedio por episodio
plt.figure(figsize=(10, 5))
plt.plot(avg_reward, label="Average Reward")
plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("Average Reward Over Episodes (Dyna-Q+)")
plt.legend()
plt.show()

# iii. Tasa de convergencia
plt.figure(figsize=(10, 5))
plt.plot([s for s in steps_to_goal if s > 0], label="Steps to Goal")
plt.xlabel("Episodes")
plt.ylabel("Steps to Goal")
plt.title("Convergence Rate: Steps to Reach Goal (Dyna-Q+)")
plt.legend()
plt.show()
```







Análisis

a. Se puede observar a partir de los gráficos, que MCTS no logra llegar al objetivo en ningún episodio. Esto se puede deber a distintos factores. Por un lado, puede ser que no haya una exploración suficiente del espacio de acciones, lo cual lleva a no obtener recompensas positivas. También, el entorno al ser estocástico, puede llevar a caer muchas veces en agujeros. Que no sea determinístico el ambiente puede causar que el modelo no aprenda correctamente. Por otro lado DynaQ+, logra obtener bastantes recompensas positivas y en general tiene éxito, pudiéndose adaptar correctamente al ambiente.

b. MCTS tiene una capacidad de búsqueda en profundidad, lo cual le permite al agente evaluar a largo plazo las consecuencias de sus decisiones. Además, la exploración y

explotación balanceadas, le hace probar distintos caminos. No requiere un modelo exacto y es flexible. Sin embargo, en un entorno estocástico puede sufrir de alta varianza en los resultados de sus simulaciones, llevando a decisiones subóptimas. También puede ser muy costoso computacionalmente y depende de la calidad de las simulaciones.

Por otra parte, DynaQ+ al usar la planificación, puede tener un aprendizaje acelerado, permitiendo que el agente actualice su política de forma más rápida. Además, tener un bonus para exploración de estados, incentiva al agente a visitar estas áreas y adaptar su política a los cambios del entorno, lo cuál es especialmente útil en un entorno estocástico. Sin embargo, DynaQ+ requiere de un modelo del ambiente, lo cual puede ser costoso de obtener y mantener. Además, el bonus de exploración puede llevar a que el agente se quede en un loop de exploración y no logre converger a la política óptima.

c. MCTS sufre de alta varianza en sus simulaciones debido a la estocasticidad, lo que puede llevar a decisiones subóptimas al basarse en muestras aleatorias que no reflejan consistentemente la mejor acción.

En Dyna-Q+ la estocasticidad puede hacer que el modelo utilizado para las simulaciones sea impreciso, lo que puede conducir a actualizaciones incorrectas de la política y a una exploración menos efectiva, afectando la eficiencia del aprendizaje. Sin embargo, su bonus de exploración puede ayudar a mitigar este problema al incentivar la exploración de estados menos visitados.

Preguntas

Estrategias de exploración:
a. ¿Cómo influye la bonificación de exploración en Dyna-Q+ en la política en comparación con el equilibrio de exploración-explotación en MCTS? ¿Qué enfoque conduce a una convergencia más rápida en el entorno FrozenLake-v1?

La bonificación incentiva al agente a explorar acciones y estados visitados con menor frecuencia. Puede ayudar a descubrir rutas óptimas, con la desventaja de que puede ser muy lento si es muy estocástico el ambiente. Por otra parte, el uso de UCT que permite una exploración y explotación balanceada, puede llevar a una convergencia más rápida, ya que se prueban distintas rutas y se explora el espacio de acciones. Sin embargo, se puede observar que en un ambiente estocástico, MCTS no logra llegar al objetivo, lo cual puede ser un problema.

2. Rendimiento del algoritmo:

a. ¿Qué algoritmo, MCTS o Dyna-Q+, tuvo un mejor rendimiento en términos de tasa de éxito y recompensa promedio en el entorno FrozenLake-v1? Analice por qué uno podría superar al otro dada la naturaleza estocástica del entorno.

DynaQ+ tuvo un mejor rendimiento en términos de tasa de éxito y recompensa promedio en el entorno FrozenLake-v1. Esto se debe a que DynaQ+ utiliza la planificación para acelerar el aprendizaje y adaptarse al ambiente, mientras que MCTS se basa en la exploración y explotación de rutas, lo cual puede ser muy costoso computacionalmente y no lograr llegar al objetivo en un ambiente estocástico. Además, como se ha mencionado, DynaQ+ tiene un bonus de exploración que incentiva al agente a explorar rutas menos visitadas, lo cual puede ser útil en un ambiente estocástico.

3. Impacto de las transiciones estocásticas:

a. ¿Cómo afectan las transiciones probabilísticas en FrozenLake-v1 al proceso de planificación en MCTS en comparación con Dyna-Q+? ¿Qué algoritmo es más robusto a la aleatoriedad introducida por el entorno?

Las transiciones probabilísticas en FrozenLake-v1 afectan el proceso de planificación en MCTS al introducir incertidumbre en las simulaciones y en la evaluación de las acciones. Esto puede llevar a una alta varianza en los resultados de las simulaciones y a decisiones subóptimas. Por otro lado, Dyna-Q+ puede ser más robusto a la aleatoriedad introducida por el entorno al utilizar un bonus de exploración que incentiva al agente a explorar rutas menos visitadas y adaptar su política a los cambios del ambiente. Además, Dyna-Q+ utiliza la planificación para acelerar el aprendizaje y adaptarse al ambiente, lo cual puede ser útil en un entorno estocástico.

4. Sensibilidad de los parámetros:

a. En la implementación de Dyna-Q+, ¿cómo afecta el cambio de la cantidad de pasos de planificación n y la bonificación de exploración a la curva de aprendizaje y al rendimiento final? ¿Se necesitarían diferentes configuraciones para una versión determinista del entorno?

En la implementación de Dyna-Q+, aumentar el número de pasos de planificación n permite al agente aprender más rápidamente al aprovechar experiencias simuladas, lo que acelera la convergencia inicial. Sin embargo, esto puede incrementar el costo computacional y, si el modelo del entorno es inexacto, podría introducir errores en la política. Por otro lado, una bonificación de exploración alta incentiva al agente a explorar más, lo que mejora la cobertura del espacio de estados y puede conducir a un mejor rendimiento final, aunque podría ralentizar la convergencia si la exploración se vuelve excesiva.

En un entorno determinista, se puede reducir tanto n como la bonificación de exploración, ya que las transiciones son predecibles y la necesidad de explorar es menor. En cambio, en un entorno estocástico es preferible usar un mayor n y una bonificación de exploración más alta para asegurar que el agente explore adecuadamente y maneje la incertidumbre en las transiciones, lo que ayuda a encontrar una política más robusta.