

## Task 01

1. **Prioritized sweeping** es una técnica de planificación en ambientes determinísticos que prioriza la actualización de los pares estado-acción que tienen mayor impacto en el valor estimado. En lugar de seleccionar transiciones al azar para simular y actualizar, Prioritized sweeping mantiene una cola de prioridades basada en el cambio esperado en el valor de los estados. Así, se enfoca en propagar rápidamente la información relevante a través del modelo, acelerando el aprendizaje y la convergencia de los valores óptimos.
2. **Trajectory Sampling** es un método de planificación que consiste en simular trayectorias completas a partir del modelo del ambiente, siguiendo una política determinada. En vez de actualizar valores individuales de estados o acciones, se generan episodios ficticios y se utilizan las recompensas obtenidas para ajustar los valores estimados. Este enfoque permite capturar dependencias temporales y explorar secuencias de decisiones, siendo útil especialmente en ambientes donde las transiciones y recompensas dependen del historial.
3. **Upper Confidence Bounds para Árboles (UCT)** es una estrategia utilizada en algoritmos de búsqueda como Monte Carlo Tree Search (MCTS) para balancear la exploración y explotación al seleccionar acciones. UCT asigna a cada acción un valor que combina la recompensa promedio obtenida y un término de confianza que favorece acciones menos exploradas. Esto permite que el algoritmo explore nuevas acciones potencialmente prometedoras, mientras aprovecha el conocimiento adquirido, logrando una búsqueda eficiente y efectiva en espacios de decisión grandes.

## Task 02

```
In [2]: import math, numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
```

```
In [ ]: def make_env(seed=42, is_slippery=True, map_name="4x4"):
    env = gym.make("FrozenLake-v1", is_slippery=is_slippery, map_name=map_name)
    env.reset(seed=seed)
    P = env.unwrapped.P
    nS = env.observation_space.n
    nA = env.action_space.n
    return env, nS, nA, P

env, nS, nA, P = make_env(seed=42, is_slippery=True, map_name="4x4")
print(f"Entorno listo. nS={nS}, nA={nA}")
```

Entorno listo. nS=16, nA=4

```
In [ ]: def moving_average(x, w=20):
    x = np.asarray(x, dtype=float)
    if len(x) < w: return x
    c = np.cumsum(np.insert(x, 0, 0.0))
    return (c[w:] - c[:-w]) / float(w)

def plot_series(y, title, xlabel, ylabel):
    plt.figure()
    plt.plot(y)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.show()

def is_terminal_from_P(P, s):
    return all(out[3] for out in P[s][0])

def sample_from_P(P, s, a, rng):
    outcomes = P[s][a]
    probs = [o[0] for o in outcomes]
    idx = rng.choice(len(outcomes), p=probs)
    _, ns, r, done = outcomes[idx]
    return ns, r, bool(done)
```

## MCTS (UCT)

```
In [ ]: class MCTSNode:
    __slots__ = ("state", "parent", "children", "N", "W", "untried_actions", "is_terminal")
    def __init__(self, state, parent=None, nA=4, is_terminal=False):
        self.state = state
        self.parent = parent
        self.children = {}
        self.N = 0
        self.W = 0.0
        self.untried_actions = list(range(nA))
        self.is_terminal = is_terminal

    def uct_score(parent, child, c=1.414213562):
        if child.N == 0: return float("inf")
        return (child.W / child.N) + c * math.sqrt(math.log(parent.N + 1) / child.N)

    def mcts_select(node, c=1.414213562):
        cur = node
        while (not cur.is_terminal) and (len(cur.untried_actions) == 0) and (len(cur.children) == 0):
            cur = max(cur.children.items(), key=lambda kv: uct_score(cur, kv[1], c))
        return cur

    def mcts_expand(node, P, nA, rng):
        if node.is_terminal or not node.untried_actions:
            return node, 0.0
        a = node.untried_actions.pop(rng.integers(len(node.untried_actions)))
```

```

ns, r, done = sample_from_P(P, node.state, a, rng)
child = MCTSNode(ns, parent=node, nA=nA, is_terminal=done or is_terminal_from_P
node.children[a] = child
return child, r

def mcts_rollout(state, P, rng, max_depth=50, gamma=0.99):
    total, g, s = 0.0, 1.0, state
    for _ in range(max_depth):
        if is_terminal_from_P(P, s): break
        a = int(rng.integers(0, 4))
        ns, r, done = sample_from_P(P, s, a, rng)
        total += g * r
        g *= gamma
        s = ns
        if done: break
    return total

def mcts_backprop(node, reward):
    cur = node
    while cur is not None:
        cur.N += 1
        cur.W += reward
        cur = cur.parent

def mcts_action(P, root_state, nA, sims_per_move=200, c=1.4142, gamma=0.99, max_depth=50):
    rng = np.random.default_rng(seed)
    root = MCTSNode(root_state, parent=None, nA=nA, is_terminal=is_terminal_from_P)
    for _ in range(sims_per_move):
        node = mcts_select(root, c)
        if node.is_terminal:
            mcts_backprop(node, 0.0)
            continue
        new_node, r = mcts_expand(node, P, nA, rng)
        reward = r + gamma * mcts_rollout(new_node.state, P, rng, max_depth=max_depth)
        mcts_backprop(new_node, reward)
    if len(root.children) == 0:
        return int(np.random.default_rng().integers(nA))
    best_a = max(root.children.items(), key=lambda kv: kv[1].N)[0]
    return int(best_a)

```

## Dyna-Q+

```

In [ ]: def dyna_q_plus(env, nS, nA, episodes=500, alpha=0.1, gamma=0.99, epsilon=0.1,
                           planning_steps=20, kappa=1e-3, max_steps=200, seed=0):
    rng = np.random.default_rng(seed)
    Q = np.zeros((nS, nA), dtype=float)

    model_r = np.zeros((nS, nA), dtype=float)
    model_ns = np.zeros((nS, nA), dtype=int)
    model_done = np.zeros((nS, nA), dtype=bool)
    last_tried = np.zeros((nS, nA), dtype=int)

    visited_sa = np.zeros((nS, nA), dtype=bool)
    total_sa = nS * nA

```

```

rewards, success, steps_to_goal, visited_ratio = [], [], [], []
t_global = 0

for ep in range(episodes):
    s, _ = env.reset()
    ep_rew, steps, ok = 0.0, 0, 0

    for _ in range(max_steps):
        t_global += 1
        a = int(rng.integers(nA)) if (rng.random() < epsilon) else int(np.argmax(
            ns, r, terminated, truncated, _ = env.step(a)
        done = bool(terminated or truncated)

        td_target = r + gamma * (0.0 if done else np.max(Q[ns]))
        Q[s, a] += alpha * (td_target - Q[s, a])

        model_r[s, a] = r
        model_ns[s, a] = ns
        model_done[s, a] = done
        last_tried[s, a] = t_global
        visited_sa[s, a] = True

        s = ns
        ep_rew += r
        steps += 1

        seen = np.argwhere(last_tried > 0)
        for _p in range(min(planning_steps, len(seen))):
            sp, ap = seen[rng.integers(len(seen))]
            bonus = kappa * math.sqrt(max(0, t_global - last_tried[sp, ap]))
            rp = model_r[sp, ap] + bonus
            nsp = model_ns[sp, ap]
            td_target_p = rp + gamma * (0.0 if model_done[sp, ap] else np.max(Q[
                sp, ap] += alpha * (td_target_p - Q[sp, ap])

        if done:
            ok = 1 if r > 0 else 0
            break

    rewards.append(ep_rew)
    success.append(ok)
    steps_to_goal.append(steps if ok == 1 else np.nan)
    visited_ratio.append(np.count_nonzero(visited_sa) / total_sa)

return {
    "Q": Q,
    "rewards": np.array(rewards, float),
    "success": np.array(success, float),
    "steps_to_goal": np.array(steps_to_goal, float),
    "visited_ratio": np.array(visited_ratio, float),
}

```

In [7]: `def mcts_run(env, P, nS, nA, episodes=300, sims_per_move=150, c=1.2,
 gamma=0.99, max_depth=50, max_steps=200, seed=1):
 rewards, success, steps_to_goal = [], [], []`

```

for ep in range(episodes):
    s, _ = env.reset(seed=int(seed + ep))
    ep_rew, steps, ok = 0.0, 0, 0
    for _ in range(max_steps):
        a = mcts_action(P, s, nA, sims_per_move=sims_per_move, c=c, gamma=gamma
                        ns, r, terminated, truncated, _ = env.step(a))
        ep_rew += r
        steps += 1
        s = ns
        if terminated or truncated:
            ok = 1 if r > 0 else 0
            break
    rewards.append(ep_rew)
    success.append(ok)
    steps_to_goal.append(steps if ok == 1 else np.nan)
return {
    "rewards": np.array(rewards, float),
    "success": np.array(success, float),
    "steps_to_goal": np.array(steps_to_goal, float),
}

```

In [8]:

```

EPISODES_MCTS = 300
EPISODES_DYNA = 500

mcts_cfg = dict(episodes=EPISODES_MCTS, sims_per_move=150, c=1.2, gamma=0.99, max_d
dyna_cfg = dict(episodes=EPISODES_DYNA, alpha=0.15, gamma=0.99, epsilon=0.10, plann

print("Corriendo MCTS...")
res_mcts = mcts_run(env, P, nS, nA, **mcts_cfg)
print("Listo MCTS.")

print("Corriendo Dyna-Q+...")
res_dyna = dyna_q_plus(env, nS, nA, **dyna_cfg)
print("Listo Dyna-Q+.")

```

Corriendo MCTS...  
Listo MCTS.  
Corriendo Dyna-Q+...  
Listo Dyna-Q+.

In [9]:

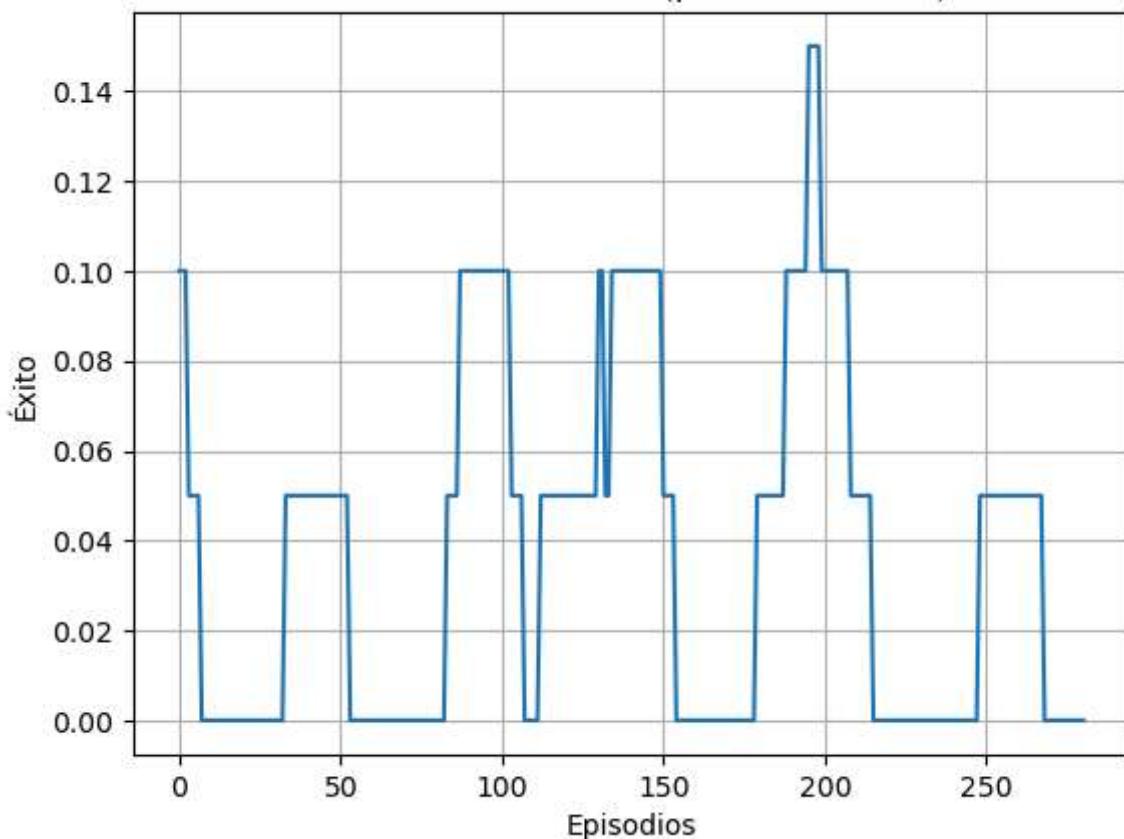
```

ma_win = 20
mcts_success_ma = moving_average(res_mcts["success"], w=ma_win)
dyna_success_ma = moving_average(res_dyna["success"], w=ma_win)

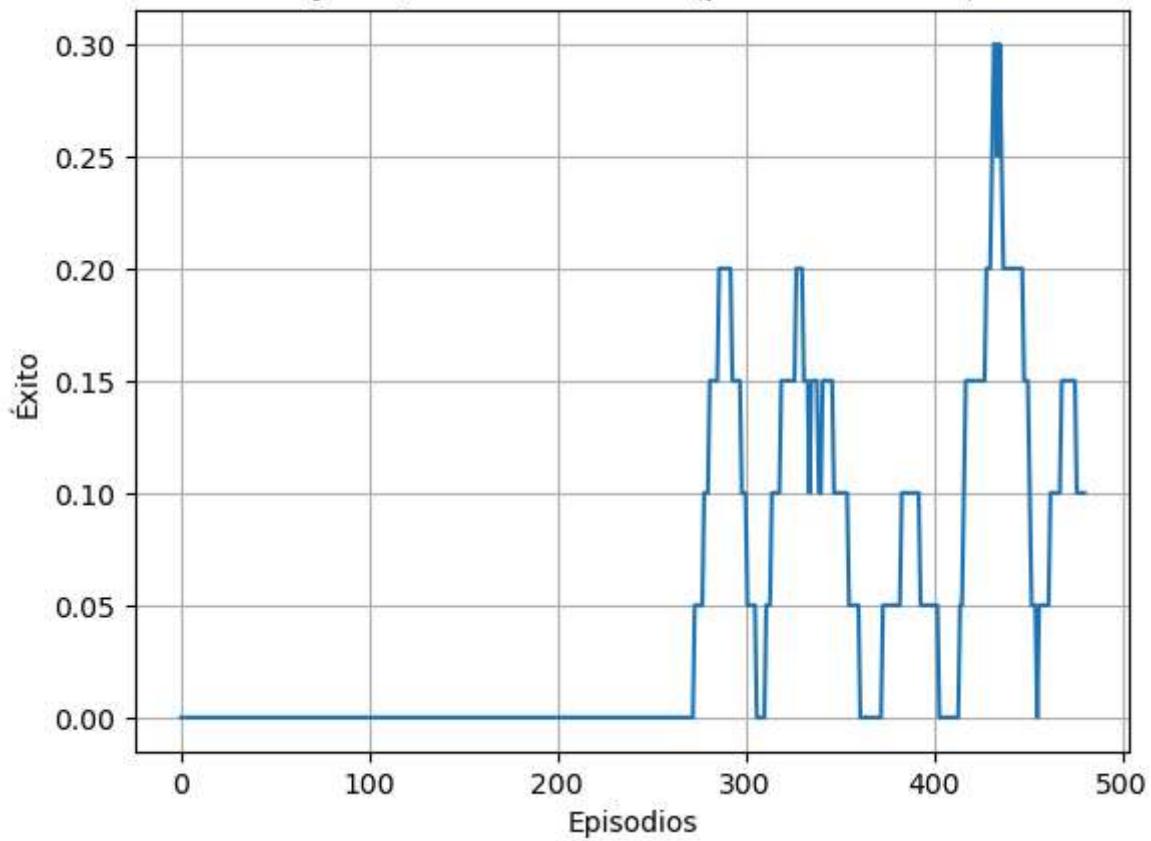
plot_series(mcts_success_ma, "MCTS: Tasa de éxito (promedio móvil)", "Episodios",
plot_series(dyna_success_ma, "Dyna-Q+: Tasa de éxito (promedio móvil)", "Episodios")

```

MCTS: Tasa de éxito (promedio móvil)

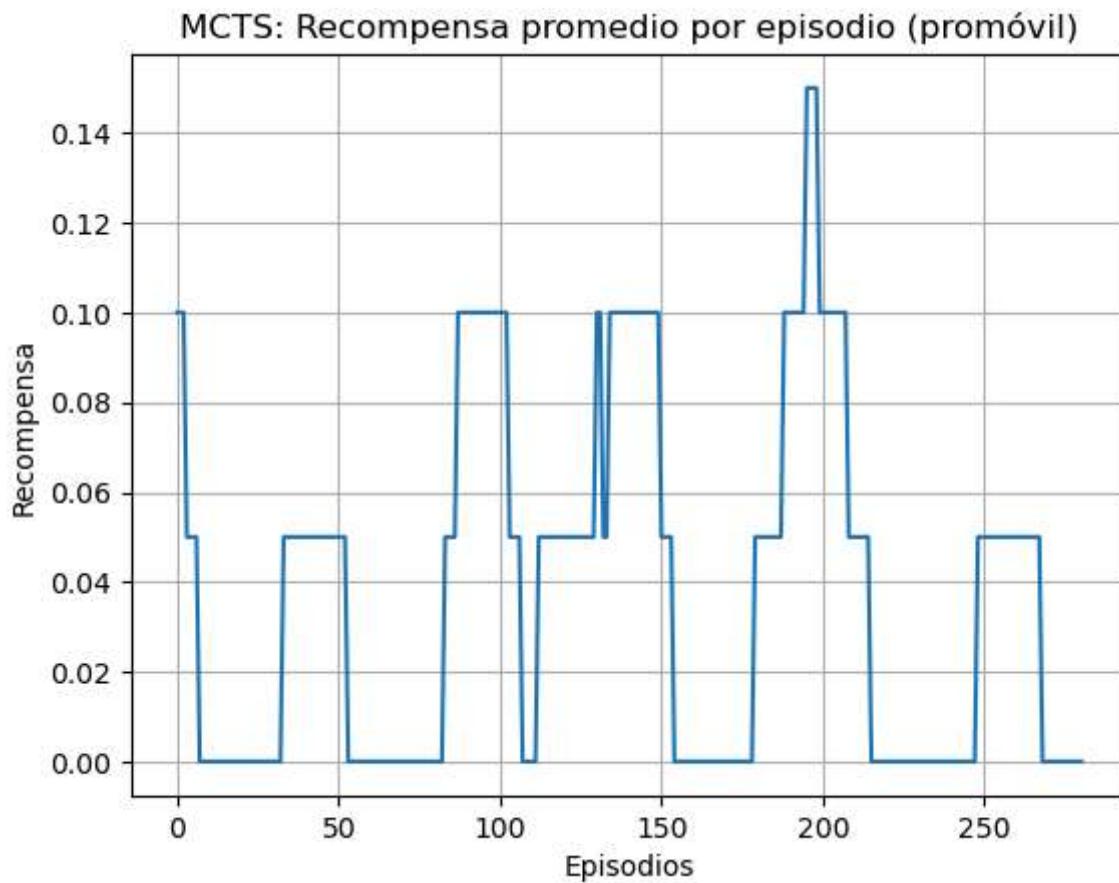


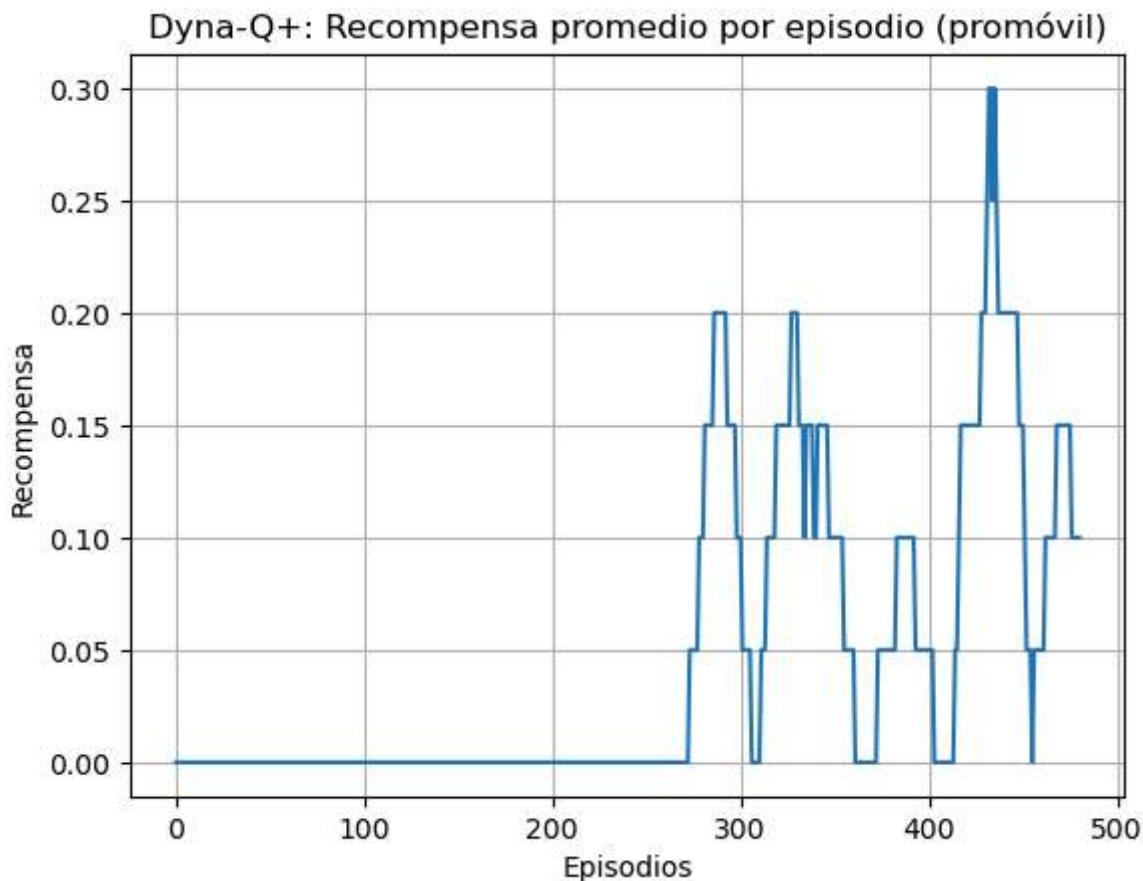
Dyna-Q+: Tasa de éxito (promedio móvil)



```
In [10]: mcts_rew_ma = moving_average(res_mcts["rewards"], w=ma_win)
dyna_rew_ma = moving_average(res_dyna["rewards"], w=ma_win)

plot_series(mcts_rew_ma, "MCTS: Recompensa promedio por episodio (promóvil)", "Episodios")
plot_series(dyna_rew_ma, "Dyna-Q+: Recompensa promedio por episodio (promóvil)", "Episodios")
```



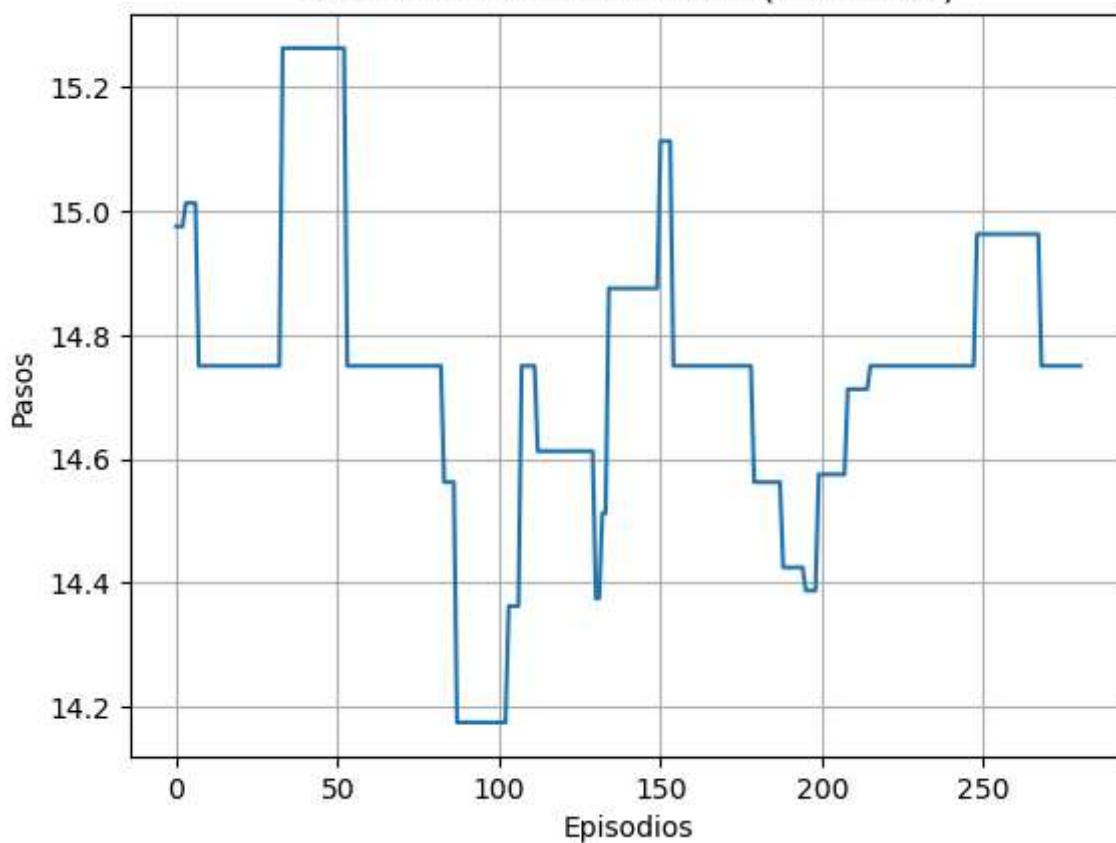


```
In [11]: def replace_nan_with_mean(y):
    y = np.array(y, dtype=float)
    if np.all(np.isnan(y)): return y
    mean_val = np.nanmean(y)
    return np.where(np.isnan(y), mean_val, y)

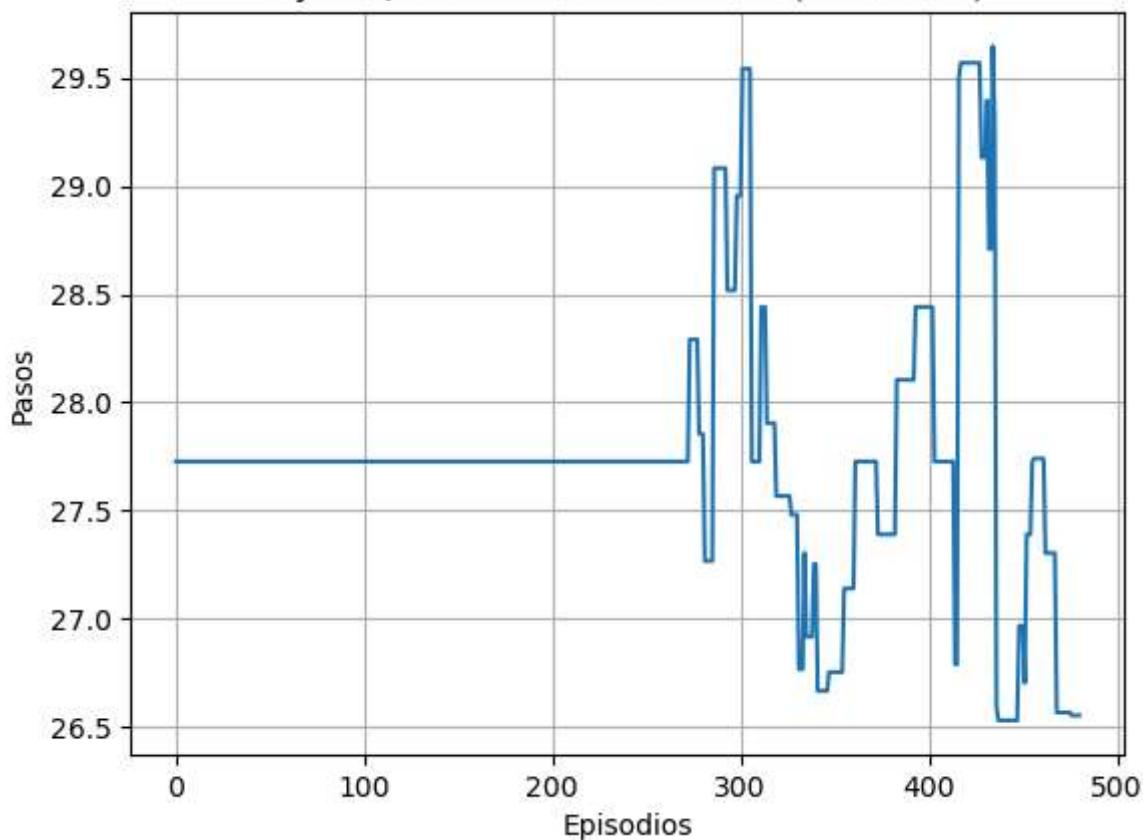
mcts_steps_ma = moving_average(replace_nan_with_mean(res_mcts["steps_to_goal"])), w=
dyna_steps_ma = moving_average(replace_nan_with_mean(res_dyna["steps_to_goal"])), w=

plot_series(mcts_steps_ma, "MCTS: Pasos hasta la meta (suavizado)", "Episodios", "P
plot_series(dyna_steps_ma, "Dyna-Q+: Pasos hasta la meta (suavizado)", "Episodios",
```

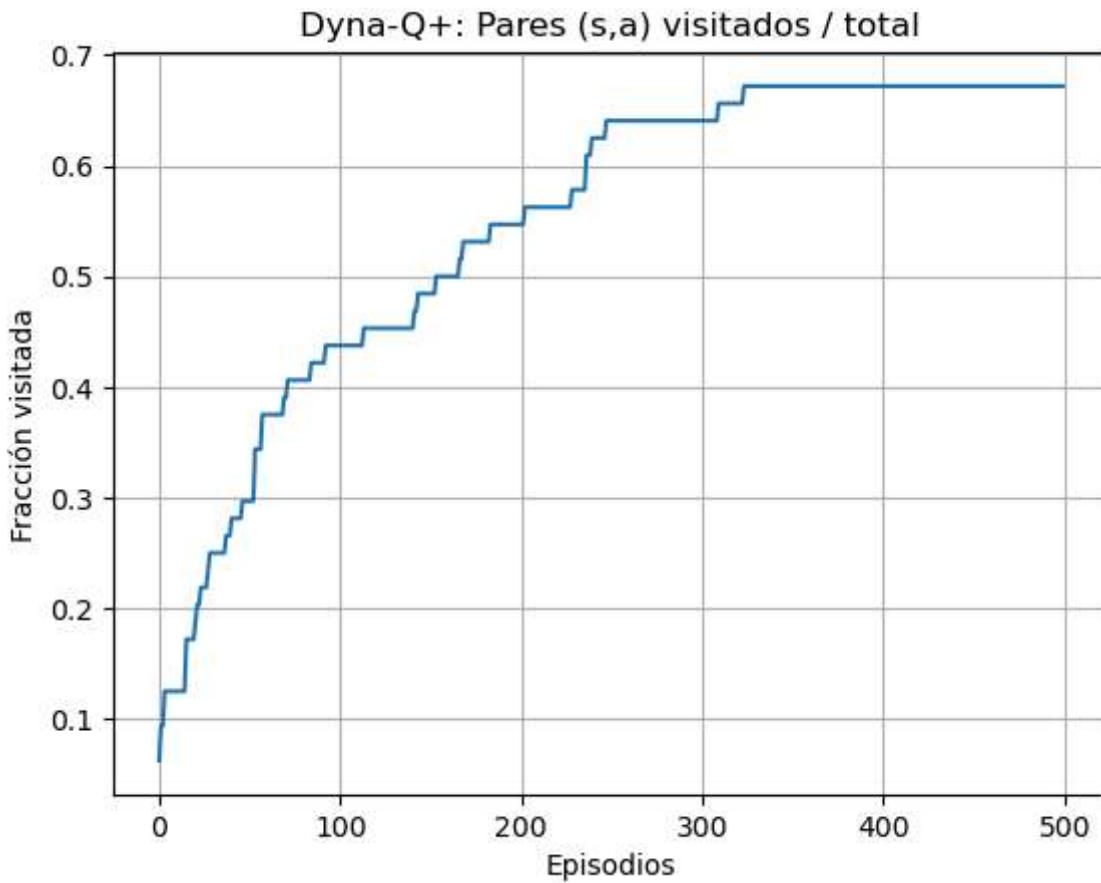
MCTS: Pasos hasta la meta (suavizado)



Dyna-Q+: Pasos hasta la meta (suavizado)



```
In [12]: plot_series(res_dyna["visited_ratio"], "Dyna-Q+: Pares (s,a) visitados / total", "E
```



## Análisis

### a) Comparación entre MCTS y Dyna-Q+.

En términos de tasa de éxito y recompensa promedio, el desempeño de MCTS se mantiene cercano a cero durante la mayor parte de los 300 episodios, presentando únicamente picos aislados en torno al 0.10–0.15. La recompensa promedio sigue el mismo patrón esporádico. En contraste, Dyna-Q+ muestra un arranque más tardío, aproximadamente a partir del episodio 280, pero logra alcanzar picos cercanos al 0.30 tanto en la tasa de éxito como en la recompensa promedio, reflejando una mejora sostenida tras la fase inicial.

Respecto a los pasos necesarios para llegar a la meta, MCTS se mantiene en un rango estable de 14.3 a 15.3 pasos, lo cual evidencia trayectorias relativamente cortas cuando el algoritmo logra alcanzar la meta. Dyna-Q+, en cambio, requiere entre 26.5 y 29.6 pasos, con una tendencia a reducirlos hacia el final, aunque todavía superiores a los de MCTS. Finalmente, la cobertura de exploración en Dyna-Q+ evidencia un crecimiento significativo: la fracción de pares ( $s,a$ ) visitados aumenta de aproximadamente 0.05 al inicio hasta cerca de 0.68 al final, lo que muestra la efectividad de la bonificación de exploración y la planificación para expandir la cobertura del espacio de estados.

### b) Fortalezas y debilidades en el entorno FrozenLake-v1.

El algoritmo MCTS presenta como principal fortaleza la utilización del modelo verdadero del entorno para planificar trayectorias. Cuando logra alcanzar la meta, los planes son más eficientes en número de pasos, lo que refleja una planificación adecuada. Sin embargo, entre sus debilidades se encuentra la ausencia de un mecanismo de aprendizaje entre episodios, la baja frecuencia de recompensas en el entorno y el número limitado de simulaciones por movimiento (150), factores que disminuyen la probabilidad de que las simulaciones exploren trayectorias exitosas. En un entorno resbaloso, la ramificación del árbol se amplía y las estimaciones se mantienen cercanas a cero en la mayoría de los casos.

Por su parte, Dyna-Q+ combina Q-learning con la planificación a partir de un modelo aprendido y una bonificación de exploración, lo que aumenta la eficiencia en el uso de las muestras tras los episodios iniciales. Esto se refleja en una mayor tasa de éxito y recompensa final. No obstante, el modelo utilizado registra únicamente el último resultado observado para cada par  $(s,a)$ , en lugar de la distribución completa de transiciones, lo que puede introducir sesgos. Asimismo, las trayectorias exitosas suelen ser más largas debido a la estocasticidad inherente del entorno y a la exploración continua incentivada por el algoritmo.

### c) Impacto de la estocasticidad.

En MCTS, la estocasticidad dispersa la probabilidad de que un rollout alcance el objetivo dentro de los 50 pasos máximos, lo que, sumado a la naturaleza binaria de la recompensa, genera estimaciones ruidosas y con poca información útil para guiar la búsqueda. En cambio, Dyna-Q+ mitiga el efecto de la estocasticidad al promediar las transiciones a través del aprendizaje por diferencias temporales y reforzar la planificación mediante actualizaciones repetidas. Además, la bonificación de exploración ( $\kappa \cdot \sqrt{\Delta t}$ ) impulsa la reexploración de pares de estado-acción olvidados, lo que le confiere mayor robustez frente a la aleatoriedad del entorno, aunque a costa de trayectorias más largas.

## Respuestas a las preguntas

**1. Estrategias de exploración** En Dyna-Q+, la bonificación de exploración incentiva reexplorar pares estado-acción poco visitados, ampliando la cobertura del espacio y favoreciendo la convergencia. En MCTS, la exploración se limita al árbol mediante UCT y no se acumula entre episodios. En FrozenLake-v1, Dyna-Q+ logra una convergencia más clara, mientras que MCTS no muestra un patrón definido con el número de simulaciones utilizado.

**2. Rendimiento del algoritmo** Dyna-Q+ supera a MCTS en tasa de éxito y recompensa promedio, ya que combina aprendizaje real con planificación repetida, lo que lo hace más eficiente en un entorno estocástico con recompensas escasas. MCTS, al no retener información entre episodios, rara vez logra acumular recompensas significativas.

**3. Impacto de las transiciones estocásticas** En MCTS, la aleatoriedad expande el árbol y reduce la probabilidad de alcanzar la meta, generando valores inestables. Dyna-Q+, en cambio, promedia las transiciones mediante Q-learning y planificación, resultando más robusto frente a la estocasticidad.

**4. Sensibilidad de los parámetros (Dyna-Q+)** Un mayor número de pasos de planificación acelera la propagación de valores pero puede inducir sobreajuste si el modelo es sesgado. La bonificación de exploración  $\kappa$  debe equilibrarse: valores bajos llevan a estancamiento y valores altos a trayectorias largas. En una versión determinista del entorno se recomienda reducir tanto  $\kappa$  como los pasos de planificación.

Link de GitHub: <https://github.com/angelcast2002/LAB-06-RL>