Comparative Analysis of Hierarchical Reinforcement Learning and RL for an MDP environment

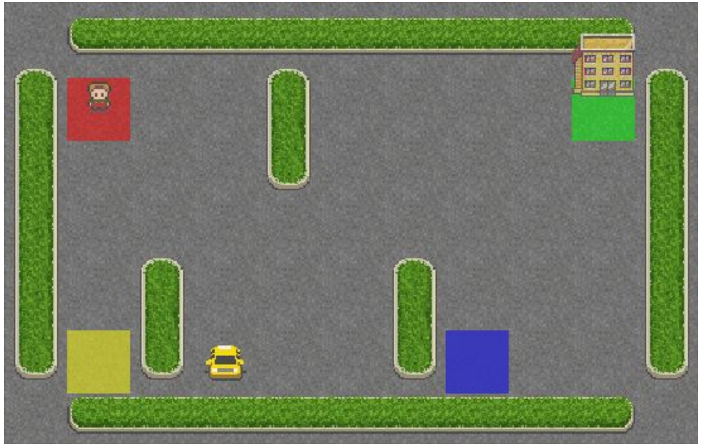
Edoardo Zompanti 1985499

Marco Antonio Iossa 2001300

Project for Reinforcement Learning course

The Environment

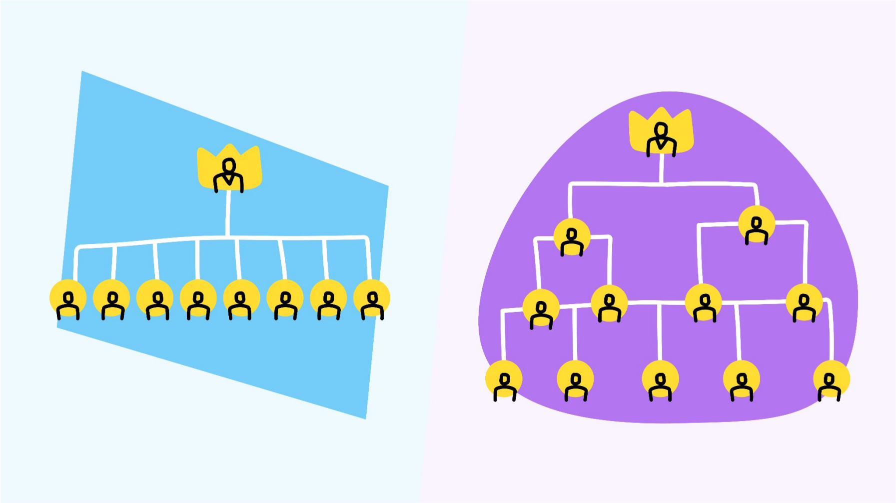
- Gymnasium Taxi-V3, 5x5 grid with 4 locations
- Observation Space:
  - 0: Red destination/passenger location
  - 1: Green destination/passenger location
  - 2: Yellow destination/passenger location
  - 3: Blue destination/passenger location
  - 4: In taxi passenger location
- Action Space:
  - 0: Move south (down)
  - 1: Move north (up)
  - 2: Move east (right)
  - 3: Move west (left)
  - 4: Pickup passenger
  - 5: Drop off passenger
- **Objective: take the passenger in one of the 4 location and bring him to the destination**
- Rewards:
  - -1 for step unless another reward is given
  - -10 executing pickup or drop off illegally
  - +20 if it center the objective



Objectives of the Project

- Explanation and visualization of performances of different algorithms for the already described environment
- Focus on the differences between algorithms of flat RL and Hierarchical Reinforcement Learning
- For this kinds of environment is the HRL really needed?
- The other algorithms are functional for this environment?

FLAT vs HIERARCHICAL



Implemented Algorithms

1. Flat Reinforcement Learning

- Q-Learning
- SARSA
- Monte Carlo (Every-Visit)

2. Hierarchical Reinforcement Learning

- Hierarchical RL

Comparative analysis methodology

1. Learning Performance (Training Phase)

- Metric: Average Reward per Episode.
- Goal: Analyze how quickly the agent learns the optimal policy (Convergence Speed).

2. Efficiency (Optimality)

- Metric: Average Steps per Episode.
- Goal: Verify if the agent finds the shortest path.

3. Robustness (Testing Phase)

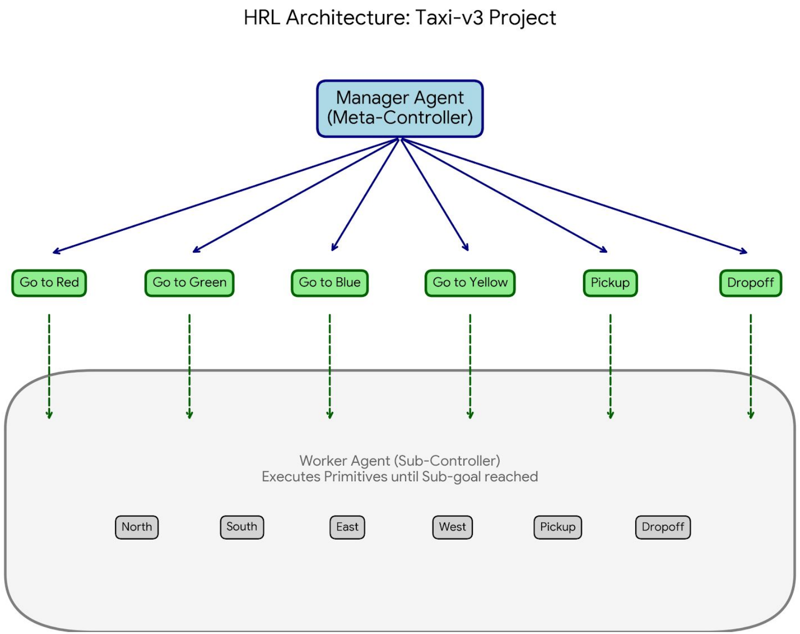
- Metric: Success Rate (over 100 test episodes).
- Method: We run the trained agents with Exploration ($\varepsilon=0$) to evaluate their final performance.

Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning decomposes a complex problem into a hierarchy of smaller sub-tasks.

- A High-Level Agent selects a specific sub-goal or macro-action .

- A Low-Level Agent executes the sequence of primitive actions required to achieve that goal.



Hierarchical Reinforcement Learning

Manager Frequency & Graph View

- Worker: Operates on the raw grid at every discrete step (t=1).
- Manager: Operates asynchronously on a high-level State-Option Graph.
- The Graph: Nodes represent critical sub-goals, and edges represent temporally extended Options. The Manager jumps between nodes, ignoring the path in between.

From MDP to SMDP

- Since edges in this graph have variable durations (k), the process is a Semi-Markov Decision Process (SMDP).
- SMDP Update Rule: We discount reward based on the actual time taken to traverse the edge (steps):

$$Target = R_{sum} + \gamma^{steps} \max Q(s', \omega')$$

```python
def check_option_termination(self, state, option_idx):
    if option_idx <= 3:
        if taxirow == target_loc[0] and taxicol == target_loc[1]:
            return True, 10.0

def execute_option(self, option_idx, state, ...):
    steps = 0
    while not option_terminated:
        next_state, reward, ... = self.env.step(action)
        steps += 1
    return state, cumulative_reward, steps

next_state, reward, steps = agent.execute_option(option_idx, state, ...)

target = reward + (agent.gamma ** steps) * agent.Q_meta[next_state, best]
agent.Q_meta[state, opt] += agent.alpha * (target - agent.Q_meta[state, opt])
```

Graph Node Definition (Sub-goals)

Worker Action Loop (Edge Traversal & Duration k)

Manager SMDP Learning (Discounting by Duration k)

Flat Reinforcement Learning

Idea: the agent learns a policy directly over **state** → **action** mappings (no hierarchy / no sub-goals).

- **Q-Learning:** an *off-policy* method that estimates Q(s,a) and improves the policy by selecting argmax_aQ(s,a).
- **SARSA:** an *on-policy* method that updates Q(s,a) using the actions actually taken (more consistent with exploration).
- **Monte Carlo Control:** learns from **complete episodes**, estimating returns G (no bootstrapping), useful as a simple reference baseline.

Q-Learning

Functioning:

- The algorithm explores the environment while updating the Q-values, gradually converging to the optimal action-value function.

- **Off-policy**: Updates Q(s,a) using the action that maximizes future rewards, not necessarily the action actually taken in the current state.

Algorithm:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

SARSA (State-Action-Reward-State-Action)

Functioning:

- **On-policy**: The Q-values are updated based on the action actually taken, making it sensitive to the current policy's exploration and exploitation.

- SARSA learns from real interactions with the environment, rather than assuming the optimal policy.

Algorithm:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Monte Carlo Control

Functioning:

- Monte Carlo Control only updates its Q-values after completing an episode, making it a batch learning method.
- It does not rely on bootstrapping (updating based on the current estimate), but instead uses actual returns to make updates.
- The algorithm is **non-bootstrapping**, meaning it waits until the end of the episode to update its estimates, ensuring unbiased estimates of the value function.

Algorithm:

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s, a) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ arbitrary
    $Returns(s, a) \leftarrow$ empty list

Repeat forever:
    (a) Generate an episode using exploring starts and π
    (b) For each pair s, a appearing in the episode:
        $R \leftarrow$ return following the first occurrence of s, a
        Append R to $Returns(s, a)$
        $Q(s, a) \leftarrow \text{average}(Returns(s, a))$
    (c) For each s in the episode:
        $\pi(s) \leftarrow \arg\max_a Q(s, a)$

Algorithms & Parameters Justification

1. HRL

- Episodes: 5,000. Sufficient due to fast convergence (< 800 eps).
- Reward Strategy: Reward Shaping. Manager gets internal bonuses (+10 Sub-goal, +50 Pickup).

2. Q-Learning

- Episodes: 5,000. Standard baseline budget. Slower than HRL but explores well.
- Reward Strategy: Sparse (Env Reward only +20/-1). No internal guidance.

3. SARSA

- Episodes: 10,000. Doubled budget. Needs more time due to safe On-Policy learning.
- Reward Strategy: Sparse. Same as Q-L but curve remains lower/noisier (suboptimal policy).

4. Monte Carlo

- Episodes: 10,000. Doubled budget. High variance (updates only at episode end).
- Reward Strategy: Sparse. Very slow/unstable as it lacks step-by-step feedback.

HRL Training

```
[Episode 0] Opening graphics window...
Episode 0 completed. Reward: -569, Steps: 200
Ep 0 | Avg Reward: -569.00 | Steps: 200 | Eps: 1.00
```

```
[Episode 100] Opening graphics window...
Episode 100 completed. Reward: -695, Steps: 200
Ep 100 | Avg Reward: -645.18 | Steps: 200 | Eps: 0.97
```

```
[Episode 500] Opening graphics window...
Episode 500 completed. Reward: -63, Steps: 30
Ep 500 | Avg Reward: -291.43 | Steps: 30 | Eps: 0.83
```

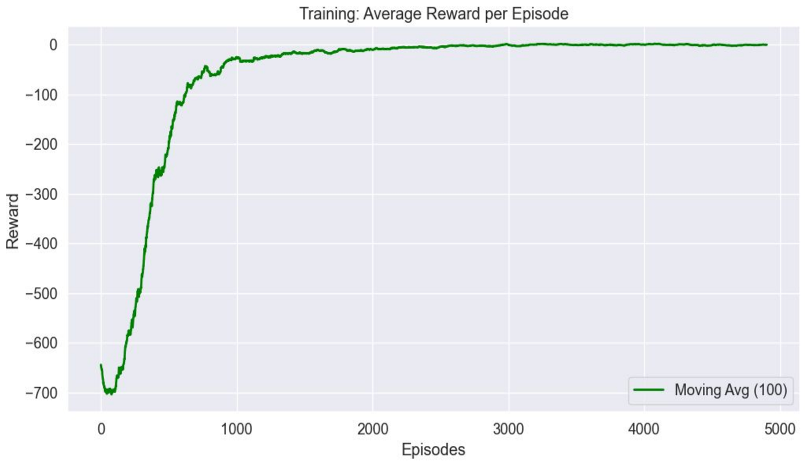
```
[Episode 3000] Opening graphics window...
Episode 3000 completed. Reward: -8, Steps: 20
Ep 3000 | Avg Reward: -0.06 | Steps: 20 | Eps: 0.01
```

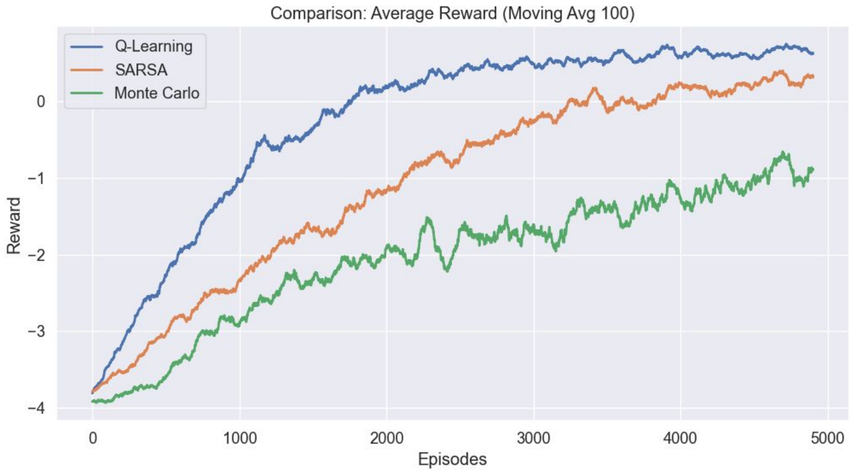
```
[Episode 5000] Opening graphics window...
Episode 5000 completed. Reward: 4, Steps: 17
Ep 5000 | Avg Reward: 0.13 | Steps: 17 | Eps: 0.01
Model saved.
```

Q-Learning, SARSA, Monte Carlo Training and test

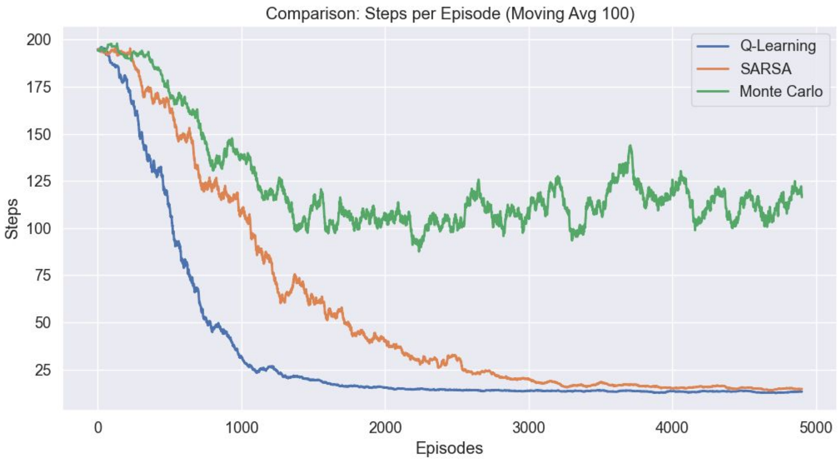
```
--- Training ALL Algorithms ---

--- Q-Learning Started ---
Q-Learning Ended. Avg Reward (last 100 eps): 0.62

--- Testing Q-Learning for 100 episodes (with Epsilon=0.1) ---
Success Rate: 100/100 (100.0%)
 -> Saved test plots for Q-Learning

--- SARSA Started ---
SARSA Ended. Avg Reward (last 100 eps): 0.68

--- Testing SARSA for 100 episodes (with Epsilon=0.1) ---
Success Rate: 100/100 (100.0%)
 -> Saved test plots for SARSA

--- Monte Carlo Started ---
MC Ended. Avg Reward (last 100 eps): -0.38

--- Testing Monte_Carlo for 100 episodes (with Epsilon=0.1) ---
Success Rate: 69/100 (69.0%)
 -> Saved test plots for Monte_Carlo
```

Training Performance (Reward vs Episodes)





Training performances - Steps per episode

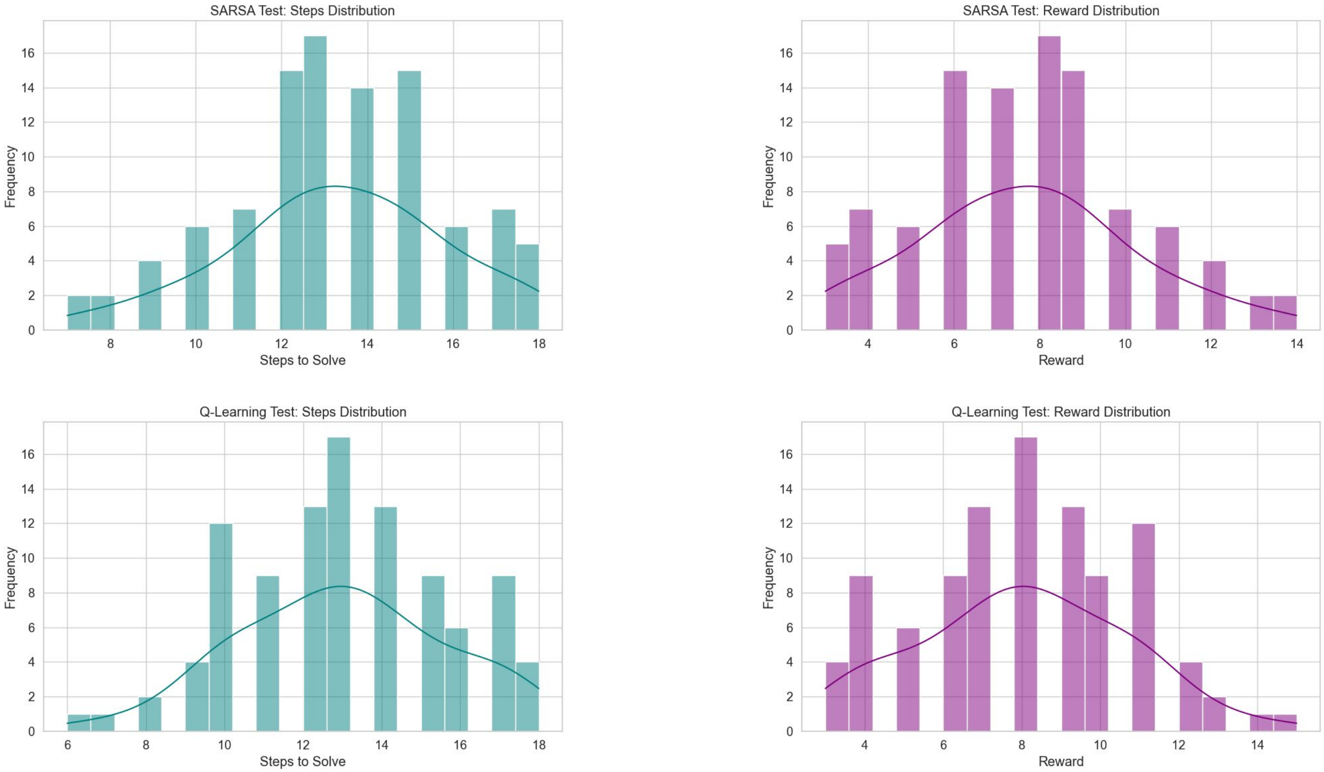


Training results

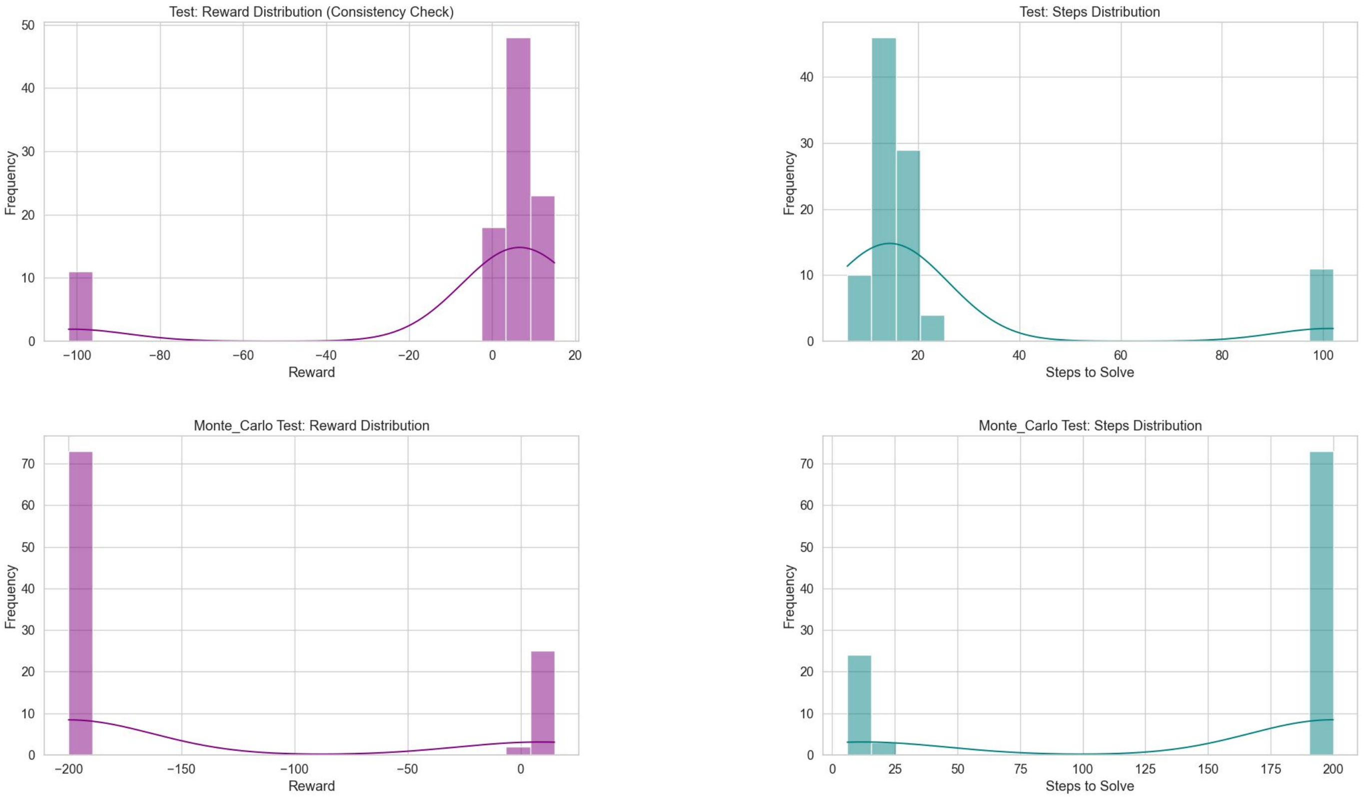
- **HRL:** Achieved optimal convergence in < 800 episodes. It wins due to Temporal Abstraction (learning 3 logical decisions instead of 20 steps).
- **Q-Learning:** Reached the goal but required ~5,000 episodes. As an Off-Policy algorithm, it finds the best path but struggles to explore the massive 500-state grid without hierarchy.
- **SARSA:** Slower and suboptimal. Being On-Policy, it learns "safe" longer paths to avoid risks during exploration, delaying convergence.
- **Monte Carlo:** Failed to stabilize (High Variance). It only updates at the end of the episode, lacking the step-by-step feedback needed for long navigation tasks.

Final Verdict: HRL proves superior by transforming a complex grid problem into a simple decision graph, learning 5x faster than the baselines.

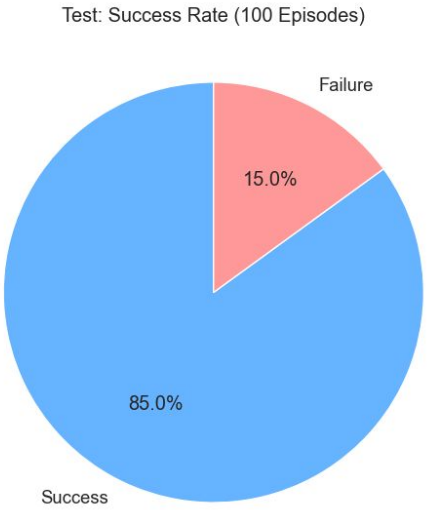
Testing Phase - Step Distribution

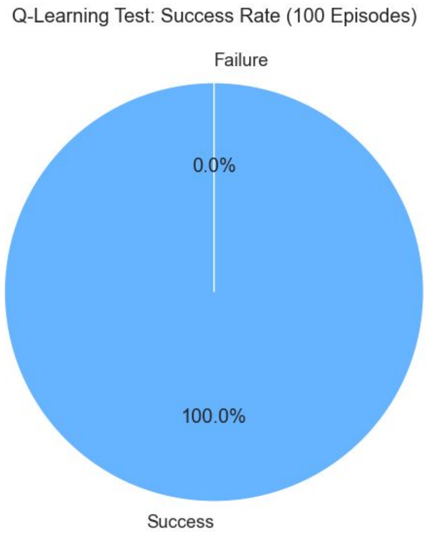


Testing Phase - Step Distribution

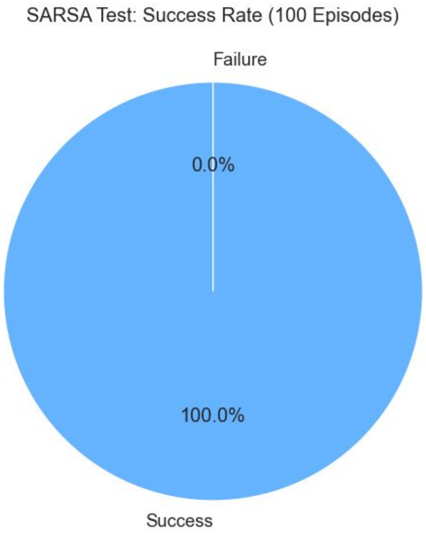


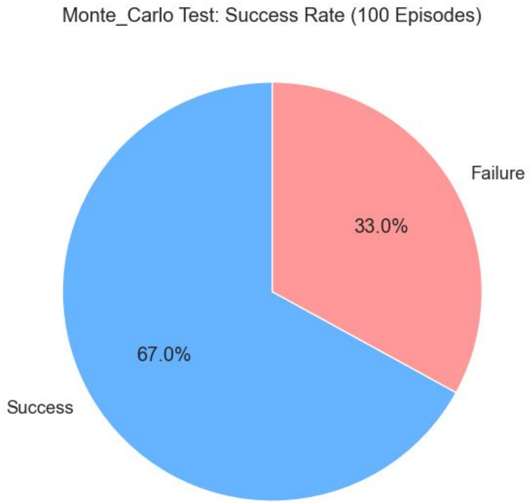
Testing Phase - success pie





Testing Phase - success pie





Testing Results Analysis

1. **Q-Learning & SARSA (100% Success):** These agents showed superior robustness. Because they re-evaluate the best move at every single step, they instantly correct any random errors caused by noise, ensuring they always reach the destination.
2. **HRL (85% Success):** Performance dropped due to delayed correction. The Manager commits to a long-term sub-goal; if noise causes the Worker to slip, the error propagates for many steps before the Manager regains control, leading to timeouts.
3. **Monte Carlo (67% Success):** The agent failed completely. Since it only updates at the end of an episode, it never learned a stable policy to navigate the long, complex path of the Taxi grid, resulting in constant timeouts.

Why start at epsilon=0.1? We initialized all agents (Flat and HRL) with 100% exploration that decays over time. This ensures the agents don't just "get lucky" at the start but actively explore the entire grid before settling on a strategy. The test results above reflect their performance after this full learning process, using a residual noise of 0.1 to test stability.

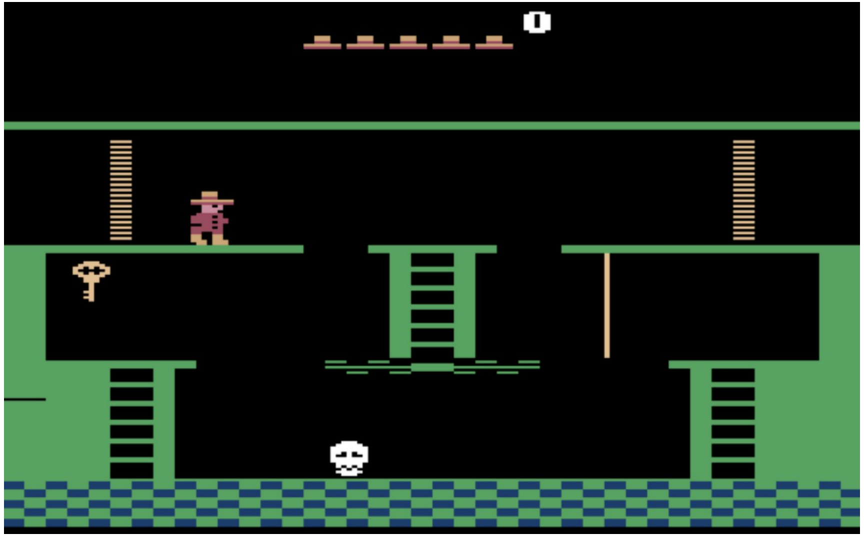
Conclusions

Is HRL "Necessary" for Taxi-v3?

- No. The state space is small enough for standard Q-Learning to solve it perfectly
- HRL is necessary if we care about sample efficiency. It reduced training time by 80%.

When HRL is Superior

- **Sparse Rewards:** Environments where the goal is very far away, and random exploration rarely hits
- **Long Horizons:** Tasks requiring thousands of steps



Conclusions

Are the other algorithms functional?

- **Q-Learning / SARSA: Yes, they are functional.** They are robust and eventually reach 100% success rates
- **Monte Carlo: No, it is not functional.** It failed to learn a stable policy because the feedback loop (waiting for the end of the episode) is too slow for a navigation task with -1 penalties at every step.
- Monte Carlo is ideal for environments with **short episodes** and a clear "Win/Loss" result at the end, where intermediate steps matter less.