

# Monte Carlo Tree Search & Priority Sweeping for Q-Learning

**Vineeth Dorna**

Department of Computer Science  
University of Massachusetts, Amherst  
vdorna@umass.edu

**Yaswanth Babu Vunnam**

Department of Computer Science  
University of Massachusetts, Amherst  
yashwanthbab@umass.edu

## 1 INTRODUCTION

In this project, we've incorporated two widely recognized algorithmic approaches: a) Monte Carlo Tree Search, and b) Priority Sweeping for Q-learning. Monte Carlo Tree Search (MCTS) stands out as a rollout algorithm that has demonstrated significant success in decision-time planning scenarios. Notably, MCTS played a pivotal role in elevating computer Go performance from a modest amateur level in 2005 to a noteworthy grandmaster level by 2015.

On the other hand, Q-learning, falling under the umbrella of model-free Reinforcement Learning (RL) algorithms, distinguishes itself by dispensing with the need for a predefined model. Instead, it endeavors to comprehend the value associated with actions in specific states through learning.

In further sections we will provide comprehensive insights into the algorithm's intricacies, shedding light on the various components that contribute to their functionality and effectiveness. Furthermore, our exploration extends to practical assessments. We subjected both algorithms to rigorous testing, evaluating their performance in the CS-687 Grid World, Cliff Walking, Cartpole and Mountain Car environments. In addition, we evaluate the performance of both Monte Carlo Tree Search (MCTS) and Priority Sweeping algorithms in contrast to the Policy Iteration algorithm on Grid World. These evaluations were conducted utilizing the Gym library from OpenAI, serving as a practical validation of the algorithms' adaptability and efficacy in real-world scenarios.

## 2 ENVIRONMENTS

### 2.1 CARTPOLE

The CartPole environment, a classic benchmark in reinforcement learning, is part of the OpenAI Gym toolkit, designed to facilitate the development and evaluation of reinforcement learning algorithms. The CartPole scenario involves a simple yet illustrative physics simulation, where a pole is attached to a cart via a joint. The objective is to balance the pole on the cart by applying left or right forces. The state of the system is defined by the cart's position, velocity, pole angle, and angular velocity. An episode terminates if the pole tilts beyond a certain angle or if the cart moves outside predefined bounds. The primary challenge is to develop a policy that allows the agent to maintain the pole in an upright position for an extended period, thus demonstrating the algorithm's ability to learn and generalize control strategies in a dynamic and partially observable environment. Figure 1 shows how the cart pole environment looks.

### 2.2 GRID WORLD

The environment is defined within a  $5 \times 5$  grid world, where each state ( $s = (r, c)$ ) denotes the current coordinates of the agent. The variable  $r$ , representing the current row, takes values within the range  $[0, 4]$ , while  $c$ , denoting the current column, ranges from  $[0, 4]$  as well. Referencing Figure 4 provides a visual representation of the grid layout. In this illustration, the topmost row is designated as row zero, and the leftmost column is identified as column zero. For instance, State1 corresponds to  $s = (0, 0)$ , and State16 corresponds to  $s = (3, 1)$ .

Actions in this environment are restricted to four possibilities: AttemptUp (AU), AttemptDown (AD), AttemptLeft (AL), and AttemptRight (AR). The dynamics of the system are characterized by

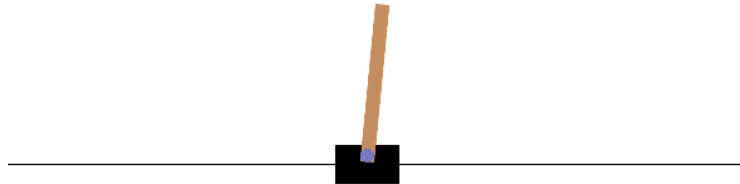


Figure 1: Cart Pole

a stochastic Markov Decision Process (MDP) with distinctive probabilities for different outcomes. Specifically, there is an 80% probability of the agent moving in the intended direction, a 5% chance of the agent veering to the right, another 5% chance of veering to the left, and a 10% probability of the agent temporarily halting. The grid world is enclosed by walls, and if the agent collides with a wall, it remains in its current state. Additionally, two Obstacle states exist at (2, 2) and (3, 2), where the agent remains stationary upon collision and is prohibited from entering Obstacle states. Notably, a Water state is situated at (4, 2), and a Goal state is positioned at (4, 4). These distinctive dynamics contribute to the complexity and variability of the agent's interactions within the grid world.

The reward structure in this scenario is consistently set to 0, except during the transition to the Goal state, where the reward becomes 10, or during the transition to the Water state, where the reward is -10.



<b>Start</b> State 1	State 2	State 3	State 4	State 5
State 6		State 8	State 9	State 10
State 11	State 12	Obstacle	State 13	State 14
State 15	State 16	Obstacle	State 17	State 18
State 19	State 20		State 22	<b>End</b> State 23

Figure 2: 687 Grid World

### 2.3 CLIFF WALKING

The Cliff Walking environment, structured as a  $4 \times 12$  matrix, serves as a compelling Markov Decision Process (MDP) within the realm of reinforcement learning. This environment is defined with specific coordinates, commencing with the agent's starting position at  $[3, 0]$ , located at the bottom-left corner. The goal, positioned at  $[3, 11]$  on the bottom-right, beckons the agent to navigate through the grid. However, a challenging element is introduced in the form of a cliff spanning  $[3, 1..10]$  at the bottom center. Stepping onto the cliff incurs consequences, necessitating the agent to return to the initial starting point. The MDP terminates upon the agent successfully reaching the designated goal.

The action space in this Cliff Walking environment is discrete and deterministic, offering the agent four possible actions: Up, Down, Left, and Right. Navigating through the grid involves the agent selecting one of these actions at each time step. The dynamics of the environment pose a challenge, as the agent must learn to balance exploration and exploitation to successfully traverse the grid and avoid the hazardous cliff.

For the Reward system, each time step incurs a penalty of  $-1$ , encouraging the agent to navigate the grid efficiently. Stepping onto the cliff, a more severe penalty of  $-100$  is imposed, emphasizing the need for cautious exploration. The interplay of rewards and penalties influences the agent's learning process, steering it towards acquiring a policy that maximizes cumulative rewards while avoiding hazardous terrain.

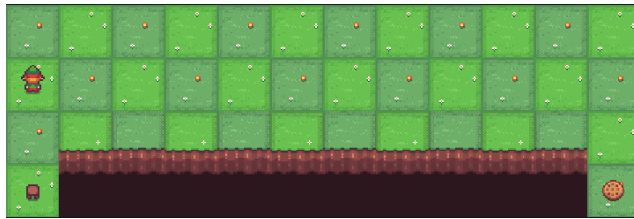


Figure 3: Cliff Walking

## 3 MONTE CARLO TREE SEARCH

### 3.1 ALGORITHM

The Monte Carlo Tree Search (MCTS) algorithm proposed in Coulom (2006) operates as a family of rollout strategies, strategically planning decisions in real-time rather than relying on exhaustive searches. In contrast to exhaustive approaches, MCTS adeptly balances exploration and exploitation of the initial set of actions while simulating trajectories. The algorithm systematically estimates the value of each action from the present state, ultimately selecting the optimal one. Subsequently, it updates its state and iteratively repeats the algorithm anew. This dynamic process, while deciding the action to take at the current step, enables MCTS to progressively refine its decision-making by iteratively exploring and exploiting the wealth of information accumulated during simulated trajectories.

The MCTS algorithm involves four pivotal steps:

**Selection:** Commencing from the root node, the algorithm strategically chooses one of its children, corresponding to an actionable move from the current node. This decision-making process carefully balances the exploration and exploitation tradeoff using the Upper Confidence Bound Algorithm (UCB). It recursively navigates through the children until it reaches a leaf node where it has no children.

**Expand:** Upon reaching a leaf node, the algorithm expands its horizons by introducing new nodes, each representing a potential action, as children of the leaf node.

**Rollout:** With the newly added children, the algorithm employs a rollout policy to sample trajectories, providing estimates for the rollout return. This step involves simulating potential outcomes based on the chosen policy.

**Back Track:** Once the rollout return is estimated, its value serves to update the action values for all nodes within the current trajectory, backpropagating from the leaf node to the root.

These four steps iteratively repeated, the number of iterations guided by a predefined budget, often reflective of the available time for decision-making at each step. Introducing a novel parameter, depth, adds another layer of control, representing the maximum number of steps an agent can query the model for results from the current state. In practical terms, this parameter mirrors the limitations of the environment simulator, indicating how many steps it can provide as outcomes from the current state to the agent. This combination of steps, budget, and depth facilitates a dynamic and adaptive decision-making process within the algorithm.

### 3.2 PSEUDO CODE

---

#### Algorithm 1 Monte Carlo Tree Search

---

```

1: Given : State  $s$ ,  $budget$ ,  $depth$ , environment  $env$ , rollout policy  $RolloutPolicy$ 
2: Return : Action  $a$ 
3:  $i \leftarrow 0$ 
4: for  $i$  in  $budget$  do
5:    $tree \leftarrow MonteCarloTree(RootNode = s)$ 
6:    $leafNode \leftarrow tree.selection()$ 
7:    $expNode \leftarrow leafNode.expand()$ 
8:    $RolloutReturn \leftarrow Tree.rollout(RolloutPolicy, expNode)$ 
9:    $Tree.backTrack(leafNode, RolloutReturn)$ 
10: end for
11:  $bestAction \leftarrow Tree.rootNode.bestAction()$ 
12: return  $bestAction$ 

```

---

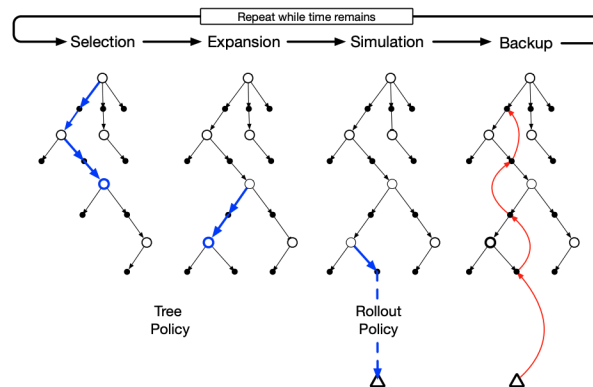


Figure 4: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations Selection, Expansion (though possibly skipped on some iterations), Simulation, and Backup, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

### 3.3 EXPERIMENTS

#### 3.3.1 BUDGET FOR MCTS

The budget here refers to the time or the number of trials we allocate to decide on action at each step. What makes Monte Carlo efficient is that it can stop whenever it reaches the budget limit, providing us with the best action based on what it has simulated so far. Additionally, we can enhance its efficiency by running simulations simultaneously on multiple computer cores, speeding up the decision-making process. The larger the budget, the more accurate the estimates of action values become. In our experiment, we explore different budgets in two scenarios—CartPole and 687Grid World—to observe how the rewards change.

Examining Figures 5 and 6 reveals a noteworthy trend: as the budgets increase, the agent demonstrates an improved capacity to converge towards the optimal policy. However, challenges emerge in environments characterized by sparse rewarding states. In such scenarios, the Monte Carlo trajectories face difficulty reaching these elusive states during simulation, posing a distinctive challenge to the optimization process. From Figure 5 we can see that we are able to take an optimal action at each step in Cart pole leading to a return of 500. For stochastic environments like 687-GridWorld in Figure 6, we can see that we are able to achieve close to the optimal reward on increasing the budget

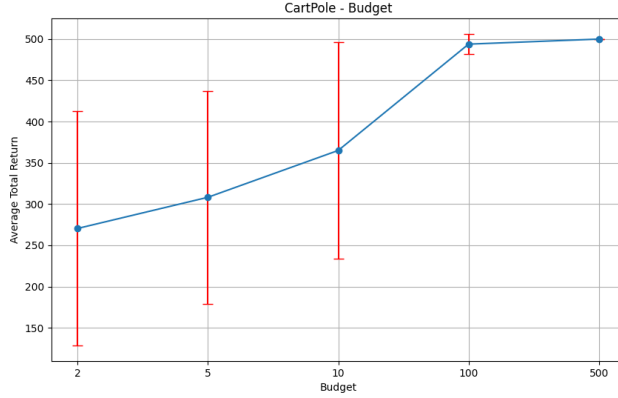


Figure 5: Experimenting with a deterministic environment like cart pole, in this plot, we ran an episode where action at each step is determined by the MCTS algorithm. The Y-axis indicates the average return from 5 differently seeded episodes utilizing MCTS and X-axis indicates the budget constraint of the MCTS. We set the exploration-exploitation strategy as UCB with  $c = 100$ ,  $\gamma = 1$  and  $depth = 50$

#### 3.3.2 DEPTH FOR MCTS

The depth is a parameter resembling a horizon that sets the limit on how far the agent can explore the environment. This implies that starting from the current state, the agent can simulate an episode with a maximum of *depth* steps. The agent can also initiate another episode with the same depth constraint in the given budget. If the rewarding state doesn't appear within this depth limit, the agent may struggle to improve, lacking feedback in its simulated interactions. Essentially, this depth serves as the agent's foresight, determining how far ahead it can look while engaging with the environment. In real-world scenarios, envision a player in a game like AlphaGo; the depth aligns with the player's ability to anticipate and strategize for the future, akin to memory or skill in foreseeing upcoming moves.

Our experimentation aimed to assess the impact of varying depth values on an MCTS agent's ability to make optimal decisions, scrutinizing both CartPole and 687Grid World environments. As depicted in Figures 7 and 8, elevating the depth parameter consistently results in the agent making more optimal decisions. Notably, in CartPole, the improvement is pronounced due to the environment providing valuable feedback at each step. In contrast, the 687Grid World scenario presents a

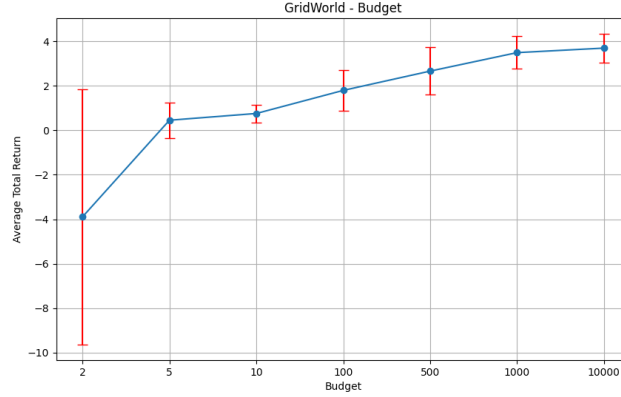


Figure 6: Experimenting with a stochastic environment like 687 GridWorld, in this plot, we ran an episode where action at each step is determined by the MCTS algorithm. The Y-axis indicates the average return from 5 differently seeded episodes utilizing MCTS and X-axis indicates the budget constraint of the MCTS. We set the exploration-exploitation strategy as UCB with  $c = 100$ ,  $\gamma = 0.9$  and  $depth = 200$

distinct challenge, as trajectory feedback is available only when the trajectory incorporates a terminal state. Despite this limitation, increasing depth still proves effective in guiding the agent toward optimal actions in both environments.

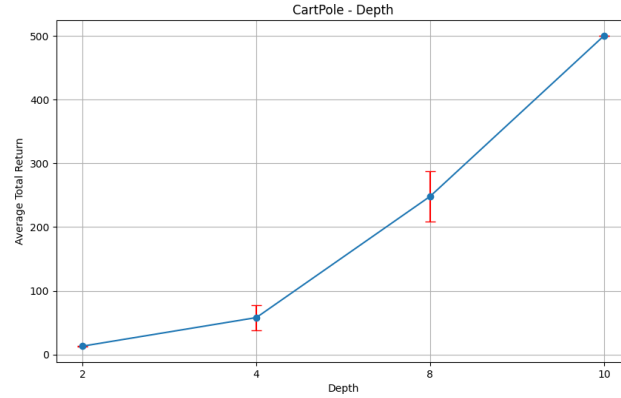


Figure 7: Experimenting with a deterministic environment like cartpole, in this plot we ran an episode where action at each step is determined by the MCTS algorithm. The Y-axis indicates the average return from 5 differently seeded episodes utilizing MCTS and X-axis indicates the depth constraint of the environment in MCTS. We set the exploration-exploitation strategy as UCB with  $c = 100$ ,  $\gamma = 1$  and  $budget = 100$

### 3.4 COMPARISON WITH VALUE ITERATION

In this experiment, we endeavor to draw a comparison between the Monte Carlo Tree Search (MCTS) algorithm and the Policy Iteration algorithm. Despite belonging to different algorithmic families, we conduct this comparison within the context of a gridworld environment. In the gridworld, we initiate the assessment from each state, employing the MCTS algorithm to determine an action. Subsequently, we evaluate and contrast this determined action with the optimal action derived from the Policy Iteration algorithm. This comparative study aims to elucidate the performance

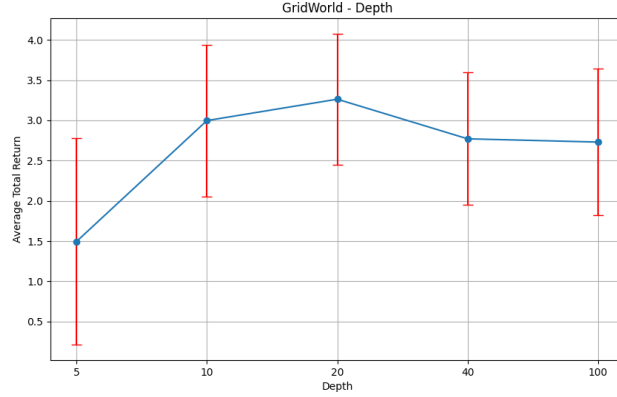


Figure 8: Experimenting with stochastic environment like 687 GridWorld, in this plot we ran an episode where action at each step is determined by MCTS algorithm. The Y-axis indicates the average return from 5 differently seeded episodes utilizing MCTS and X-axis indicates the depth constraint of the environment in MCTS. We set the exploration-exploitation strategy as UCB with  $c = 100$ ,  $\gamma = 0.9$  and  $budget = 1000$

characteristics of these distinct algorithms when applied to the gridworld environment. The final optimal policy actions from each state are elucidated in Figure 9.



Figure 9: The policy on the left is of Policy Iteration and the policy on the right is of MCTS where environment is initialized at each step for a fair comparison. The  $\gamma$  is set to 1.0

### 3.4.1 MOUNTAIN CAR

Furthermore, we applied the Monte Carlo Tree Search Algorithm to the Mountain Car Problem. Nevertheless, we acknowledge that MCTS encounters challenges in environments like Mountain Car, characterized by sparse rewards obtainable only upon reaching specific states, notably the flag at the top of the hill. Due to MCTS's dependence on simulations to estimate action and state values, the exploration-exploitation trade-off becomes notably intricate in scenarios with sparse rewards. The scarcity of rewards poses a hurdle to the algorithm's capacity to effectively explore and uncover the optimal trajectory leading to the rewarding state.

## 4 PRIORITY SWEEPING FOR Q-LEARNING

### 4.1 ALGORITHM

Reinforcement learning aims to enable agents to learn optimal behaviors through interaction with an environment. Dyna-Q is a model-based reinforcement learning algorithm that leverages a model of the environment, allowing the agent to simulate experiences and update its Q-values efficiently. Prioritized Sweeping as proposed in Moore & Atkeson (1992), when implemented with Dyna-Q, enhances the learning efficiency further by prioritizing the updates made to the Q-values, optimizing the exploration of the state-action space. In the next subsections, we describe the parts of the algorithm in more detail.

#### 4.1.1 EPISODIC INTERACTION

The algorithm begins by interacting with the environment in an episodic manner. Actions are selected based on the current Q-values, and the Q-values are updated using the observed rewards and next states, following the principles of Q-learning. Simultaneously, the agent updates its internal model of the environment with the newly observed transitions. This step constitutes the fundamental Q-learning update process.

#### 4.1.2 PLANNING

##### **Prioritization with TD Error:**

In the planning phase, the algorithm introduces a priority queue to manage state-action pairs based on the magnitude of the Temporal Difference (TD) error. The TD error quantifies the disparity between the predicted Q-value and the updated Q-value using the observed reward and the model's prediction for the next state. By prioritizing state-action pairs with higher TD errors, the algorithm focuses on experiences where the current Q-values significantly deviate from the model's predictions.

##### **Sample and Replay:**

State-action pairs are sampled from the priority queue for replay. The prioritization ensures that experiences with substantial TD errors take precedence in the replay process. This step enhances the algorithm's ability to address and rectify discrepancies between the agent's predictions and the true environment dynamics.

##### **Model-Based Q-Value Update:**

For each sampled state-action pair, the algorithm queries its internal model for predictions of the next state and reward. These predictions are then utilized to update the Q-values for the sampled state-action pairs. This model-based update complements the episodic interaction, enabling the agent to generalize its knowledge beyond direct experiences.

##### **Conditional Predecessor Update:**

As a distinctive feature of Prioritized Sweeping, the algorithm conditionally adds the possible predecessors of the sampled state-action pair into the priority queue. This step prioritizes states that are most affected by the current update, allowing the agent to focus its attention on regions of the state-action space that are likely to yield significant improvements.



## 4.2 PSEUDO CODE

---

### Algorithm 2 Priority Sweeping Algorithm

---

```

Initialize  $Q(S, A)$ ,  $Model$  and  $PQ$  to empty
2: while State is not terminal do
     $S \leftarrow$  Current State
4:    $A \leftarrow Policy(S, Q)$ 
    Take action  $A$ , observe reward  $R$ , and next state  $S'$ 
6:    $Model(S, A) := R, S'$ 
     $\delta \leftarrow |R + \gamma * \max_a Q(S', a) - Q(S, A)|$ 
8:   if  $\delta < threshold$  then Insert  $S, A$  into  $PQ$  with priority  $\delta$ 
       for  $n$  times do
10:    if  $PQ$  is empty, Break
         $S, A \leftarrow pop(PQ)$ 
12:     $R, S' \leftarrow Model(S, A)$ 
         $Q(S, A) \leftarrow Q(S, A) + \alpha * (R + \gamma * \max_a Q(S', a) - Q(S, A))$ 
14:    for All predecessors  $\bar{S}, \bar{A}$  of  $S$  do
         $R \leftarrow$  from  $Model(\bar{S}, \bar{A})$ 
16:         $\delta \leftarrow |R + \gamma * \max_a Q(\bar{S}', a) - Q(\bar{S}, \bar{A})|$ 
        if  $\delta < threshold$  then Insert  $\bar{S}, \bar{A}$  into  $PQ$  with priority  $\delta$ 
18:    end for
       end for
20: end while

```

---

## 4.3 EXPERIMENTS

In our experimentation, we tested various values for the learning rate and exploration parameters. The optimal results were obtained by utilizing the values that demonstrated the most favorable outcomes. We illustrate how the convergence to the optimal policy varies for both Markov Decision Processes (MDPs) in relation to the algorithm's hyperparameters.

### 4.3.1 NUMBER OF PLANNING STEPS

The number of planning steps is a crucial factor influencing the algorithm's convergence to the optimal policy. Increasing the number of steps results in a slower algorithmic pace per episode, yet it accelerates convergence over a reduced number of episodes. As illustrated in Figure 10, the algorithm displayed a rapid settling pattern in the case of the Grid world when the planning steps were increased. Similar was the case with Cliff Walking MDP as illustrated in Figure 11, when the planning steps are increased, the algorithm approaches optimal policy faster.

### 4.3.2 THRESHOLD THETA

In the context of priority sweeping, the update of Q-values is limited to states with high priorities. The determination of these priorities relies on a minimum threshold, dictating the significance of a state. This threshold significantly impacts the algorithm's convergence. A higher  $\theta$  (threshold) enhances the algorithm's speed by updating only the most critical states. However, it's essential to be cautious, as excessively increasing theta may hinder convergence, causing states to not undergo sufficient Q-value changes above the threshold. Conversely, reducing theta too much may slow convergence, as it involves updating states with marginal Q-value variations. It's important to note that theta's influence is more pronounced in larger state spaces.

In the grid world scenario depicted in Figure 10, it is evident that raising the threshold (theta) leads to the algorithm converging towards a sub-optimal policy. The heightened threshold resulted in the Q-values of states failing to surpass the specified threshold, thereby circumventing the iterations necessary for Q-value updates. Setting the threshold too high prevented crucial states from undergoing the updating step, consequently preventing the algorithm from converging to the optimal policy. A similar situation was observed in the CliffWalking environment, as illustrated in Figure 11

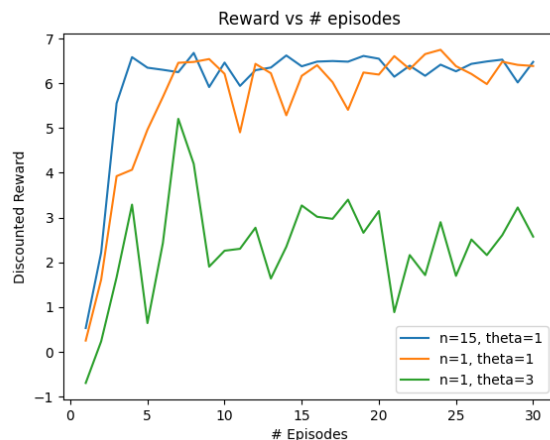


Figure 10: The above image shows the comparison between different hyperparameter configurations of the priority sweeping algorithm for **Gridworld**. The  $\alpha = 0.6, \gamma = 0.95, \epsilon = 0.15$ , being the same for all the three curves. The *horizontal axis* represents the *number of episodes* and the *vertical axis* represents the *discounted return*.

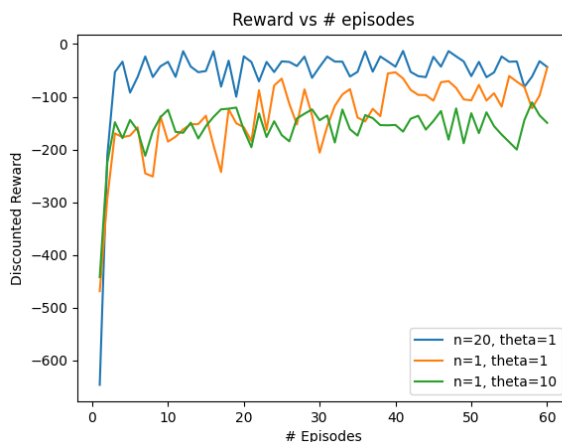


Figure 11: The above image shows the comparison between different hyperparameter configurations of the priority sweeping algorithm for **Cliff Walking**. The  $\alpha = 0.6, \gamma = 0.95, \epsilon = 0.15$ , being the same for all the three curves. The *horizontal axis* represents the *number of episodes* and the *vertical axis* represents the *discounted return*.

#### 4.4 COMPARISON WITH VALUE ITERATION

In this study, we aim to contrast the Priority Sweeping algorithm with the Policy Iteration algorithm, despite their distinct algorithmic approaches. The comparison is conducted within the confines of a gridworld environment. We obtain the optimal policy from the Priority Sweeping Algorithm after convergence from the Q-values. Subsequently, we examine and compare this chosen action policy with the optimal policy determined through the Policy Iteration algorithm. The primary objective of this comparative analysis is to shed light on the performance characteristics of these diverse algorithms when applied to the grid world context. The optimal policy actions for each state are outlined in Figure 12.

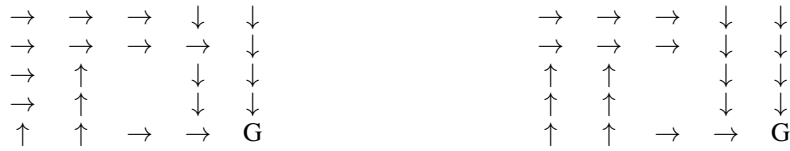


Figure 12: The policy on the left is of Priority Sweeping and the policy on the right is of Policy iteration where environment is initialized at each step for a fair comparison. The  $\gamma$  is set to 1.0

## REFERENCES

- Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers (eds.), *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, volume 4630 of *Lecture Notes in Computer Science*, pp. 72–83. Springer, 2006. doi: 10.1007/978-3-540-75538-8\_7. URL [https://doi.org/10.1007/978-3-540-75538-8\\_7](https://doi.org/10.1007/978-3-540-75538-8_7).
- Andrew Moore and Christopher Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. *Advances in neural information processing systems*, 5, 1992.