UNIVERSITY OF PISA

SYMBOLIC AND EVOLUTIONARY ARTIFICIAL INTELLIGENCE

PROJECT F3

Didactic Reiforcement Learning

Bruchi Caterina
Ruggieri Andrea

A.Y. 2023-2024

`GitHub Folder`

Didactic RL

Andrea Ruggieri, Caterina Bruchi

Contents

1 Introduction

Paper's goal is provide a complete overview of Monte Carlo methods for Policy Optimization as an alternative solution to Temporal Difference ones for problems in which complete knowledge of the environment is not required. For this aim, didactic approach was exploit for Monte Carlo methods which only require experience—sample sequences of states, actions and rewards from actual or simulated interaction with an environment- without complete probability distribution of all possible transitions, required for Dynamic Programming instead, avoiding issue in case of complex environment. Experience is gathered at the end of every episode, where episode states for the set of state-action-reward the agent pass through from starting state to terminal state and only once the latter is reached then Value function can be updated.
Main Monte Carlo features can be resumed into:

- **Model-free**: remove the assumption that mathematical model of the environment is well-known
- **Sample-based**: do not consider all the states and all actions together at the same time
- **Off-line learning**: does not learn while the agent is behaving
- **Unbiased learning**: expected return is computed using the average of returns
- **High variance**: return may change waiting for the end of the episode, TD one-step learning reduce instead the possibility of having long term variability

1.1 Tabular Methods

This Monte Carlo overview starts with the simplest method available in reinforcement learning which is **tabular** one, since using small learning spaces like the ones for this didactic purpose these methods are quite effective thanks to the table for state-action pairs visited storing, updating entry values obtaining an accurate estimation of expected return.

1.2 Tree Search

Then as a more complex application of Montecarlo methods we will analyze 2 different implementations of Monte Carlo Tree Search which is a recent and strikingly successful example of decision time planning. Core idea behind Monte Carlo Tree Search method is build up a **search tree** using simulation of samples in the search space, updating set of visited states following four step:

1. **Selection**: Select root node at first iteration, a not-fully expanded children at next iterations
2. **Expansion**: Unless a termination condition is reached, select next children nodes choosing not tested yet actions
3. **Playout (or Rollout)**: Choose one node and start a simulation until termination state is reached
4. **Backpropagation**: Update N(s,a) as number of visited state-action pairs and Q(s,a) using reward r

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. Starting from this point, MCTS is also enhanced by the addition of a means for accumulating value estimations obtained from simulations in order to direct next simulations toward more highly-rewarding ones.
Historically the first proposed idea was to implement a full tree which from a state simulated paths up to the terminal states, but later for complex games this kind of exploration was not suitable anymore so more solution using value function and policy approximation caught on.

2 Frozen-Lake

Environment chosen the aforementioned algorithms is Frozen-Lake since discrete action and state space was needed: main objective is the agent reaches the goal without falling into one of the lakes. The state space is described by four actions, every one mapped to a number:

- Move Left: 0
- Move Down: 1
- Move Right: 2
- Move Up: 3

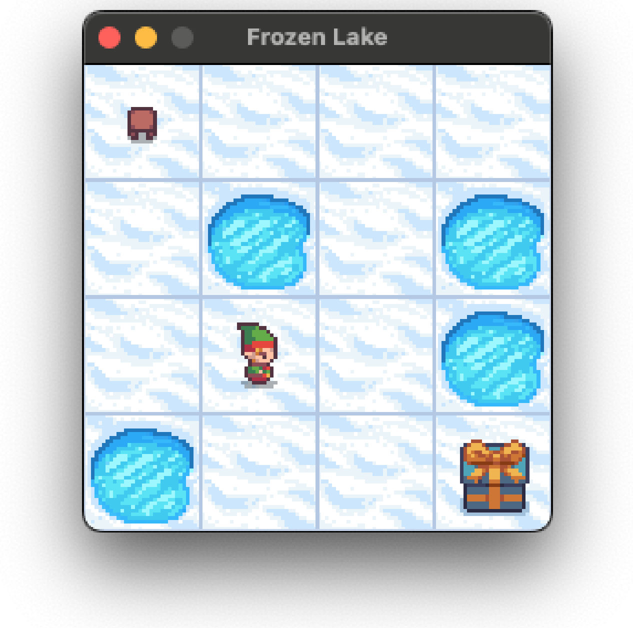


Figure 1: Frozen Lake image rendering

Below there is a visual representation of the states number distribution on the real map for a better understanding of the whole comments we are going to make on the experiments.

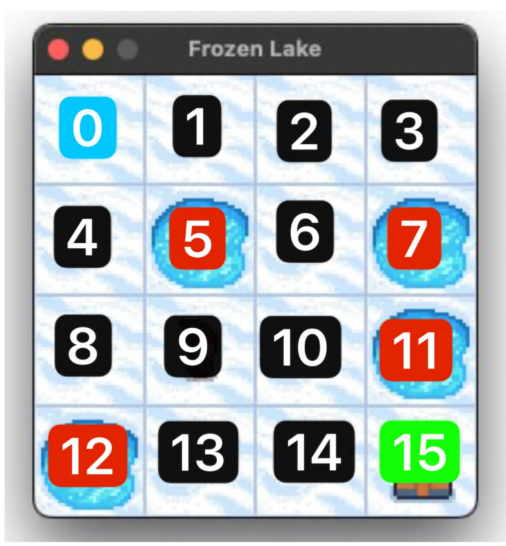


Figure 2: Frozen Lake image rendering

Apart from the ice-cells, there are also:

- One Starting state (0)
- Five terminal states
  - Four lakes (5-7-11-12)
  - One gift (15)

State space is a 4×4 matrix where every cell is mapped with an integer equal to current_row $\times$ n_rows + current_col, where both row and col start at 0:

- Starting Point cell: whenever the agent falls or reach the goal, every episode start from this cell
- Lake-Hole cells are [5,7, 11, 12]: if the agent reach one of the stages, episode ends but without giving a negative reward
- Goal cell is number 15: if the agent reach this cell the episode ends with a reward equal to one
- Any other cells are the ones the agent can cross during the episode without taking any reward

There is also the `is_slippery` feature which if true enforce the player to slip and replay action just made.

3 Tabular implementation

The aim of this kind of algorithm is the realization of a Q-Table, which is a 16×4 matrix with same number of columns as possible actions and same number of rows as possible states, where every entry represents possible reward achieved from every cell. In Monte Carlo implementation Q-Table is updated only once episode is terminated so by the end of the training phase the higher matrix values stands for the optimal path from starting point to the goal. For this purpose is important to underline that as main feature of Monte Carlo approaches the table is always updated at the end of the episode, but this can be done in two different ways:

- **First visit:** entry table is update only at the first time state-action pair is visited in the episode
- **Every visit:** every time a state-action pair is crossed during an episode, related entry in the table is updated computing the average reward

As all policy optimization algorithms, also this one can be implemented using a random policy or an ϵ-greedy one, since the first makes next action always random and the latter looks for a trade off between Exploration and Exploitation:

- **Exploration** with probability ϵ: probability the agent selects a never-before explored state-action pair
- **Exploitation** with probability $1-\epsilon$: probability the agent selects the action that currently seems best according to previous knowledge and maximizing the return as objective

Obviously different performances are achieved based on previous choice, but thanks to GLIE property which is automatically satisfied by Monte Carlo methods both converges to the optimal policy even if in different number of episodes. For didactic proposal both version with ϵ-greedy policy were implemented achieving very similar result. A tabular method can be successfully implemented in a Monte Carlo fashion way under the following assumption:

- Discrete state and action space: since a table is required, would be unfeasible test the algorithm in other conditions
- A Q-Table is needed for state-action related reward storing
- Simple game: tabular solution would be unfeasible for complex multiplayer games as Chess or Go, for which a Tree Search implementation would be suggested instead
- Known environment: regarding didactic proposal of the work, environment is considered well-known for sake of simplicity

Key features:

- Every episode starts in cell 0 which is the upper left-most cell and only one starting point in chosen environment
- Episode's steps are considered from the last to the first one since the presence of discount factor $\gamma \in [0, 1]$ which helps giving more relevance to rewards achieved at the end of the episode. Reverse study of steps' rewards is a mandatory key point of all Monte Carlo implementations
- Decay function is needed for a not stationary value for ϵ because an a-priori prediction of how much exploration and how much exploration is needed would be unmeaning: if less exploration is needed, the agent should be able to understand it, update ϵ value and follow exploitation. Two different speed of ϵ decay will be exploited to appreciate differences in policy learning

Policy optimization algorithm can be realized ON policy or OFF policy, their main difference is that in the ON policy version the episode are collected using the same policy which is been optimized, instead for the OFF policy the experience is collected using an exploratory policy μ. Both on policy and off policy version for target policy π improvement were implemented.

3.1 Algorithm structure

For the update rule, both algorithms exploit an optimization which performs an incremental mean online using following structure, instead that saving all the returns and then averaging them at the end:

$$\mu_k = \frac{1}{k}\sum_{i=1}^{k} x_i = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \tag{1}$$

For better understanding here are provided pseudo code for the used methods.

3.1.1 ON-policy method

Algorithm 1 On Policy Generate Episode

```
procedure GENERATE_EPISODE(ϵ, Q, env, max_env_steps)
    state ← ENV.RESET
    total_reward ← 0
    done ← False
    trajectory ← []
    for each step from 1 to max_env_steps do
        action ← CHOOSE_ACTION(Q, state, ϵ, env)
        new_state, reward, done, info ← ENV.STEP(action)
        Append [state, action, reward] to trajectory
        state ← new_state
        total_reward ← total_reward + reward
        if done then
            Break the loop
        end if
    end for
    Return trajectory, total_reward
end procedure
```

Algorithm 2 On-Policy Monte Carlo

```
procedure MONTE CARLO
    Q ← INITIALIZE_Q_TABLE(env)
    visits_counter ← zero matrix of size (number of states, number of actions)
    rewards ← []                                        ▷ One entry for each episode
    for each episode do
        ϵ ← DECAY_FUNCTION(episode, total_train_episodes, min_epsilon)
        trajectory, total_reward ← GENERATE_EPISODE(ϵ, Q, env, max_env_steps)
        G ← 0
        for t ← length of trajectory down to 1 do
            state, action, reward ← trajectory[t]
            G ← γ × G + reward
            if Every visit then
                if state, action is the first visit in the episode then
                    visits_counter[state, action] ← visits_counter[state, action] + 1
                    Q[state, action] ← Q[state, action] + (G − Q[state, action]) / visits_counter[state, action]
                end if
            else
                Q[state, action] ← G
            end if
        end for
    end for
    return Q, rewards
end procedure
```

3.1.2 OFF-Policy Method

Algorithm 3 On Policy Generate Episode

procedure GENERATE_EPISODE(ϵ, Q, env, max_env_steps)
  $state \leftarrow$ ENV.RESET
  $total\_reward \leftarrow 0$
  $done \leftarrow False$
  $trajectory \leftarrow []$
  for each step from 1 to max_env_steps **do**
    $action \leftarrow$ CHOOSE_ACTION(Q, state, ϵ, env)
    $new\_state, reward, done, info \leftarrow$ ENV.STEP(action)
    Append $[state, action, reward]$ to trajectory
    $state \leftarrow new\_state$
    $total\_reward \leftarrow total\_reward + reward$
    if done **then**
      Break the loop
    end if
  end for
  Return trajectory, total_reward
end procedure

Algorithm 4 Off-Policy Monte Carlo

Input: Policy π (target policy), Behavior policy μ
Initialize $Q(s, a)$ and $\pi(s)$ arbitrarily for all $s \in S, a \in A(s)$
Initialize $N(s, a) \leftarrow 0$ and $R(s, a) \leftarrow 0$ for all $s \in S, a \in A(s)$
for each episode **do**
  Generate an episode using behavior policy μ
  Initialize $G_\mu \leftarrow 0$
  for each step t in the episode, from last to first **do**
    $G_\mu \leftarrow \gamma G_\mu + R_t$
    for each (s_t, a_t) in the episode **do**
      $W(s_t, a_t) \leftarrow \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{W(s_t,a_t)}{N(s_t,a_t)}(G_\mu - Q(s_t, a_t))$
      $N(s_t, a_t) \leftarrow N(s_t, a_t) + W(s_t, a_t)$
    end for
  end for
end for

- **Target Policy (π)**: Policy to optimize and improve, dictates the actions we want to take in each state.
- **Behavior Policy (μ)**: Policy used to generate episodes, might be different from the target policy and is used to collect data.
- **Q(s, a)**: Action-Value function representing the expected return when taking action a in state s and following the policy π thereafter.
- **N(s, a)**: Number of times action a has been taken in state s.
- **W(s, a)**: Ratio of the probabilities $\frac{\pi(a|s)}{\mu(a|s)}$ that reflects how often actions are taken under the target policy compared to the behavior policy.
- **G_μ**: The return (total reward) following the behavior policy.
- **G_π**: Return (total reward) following the target policy, updated using the importance sampling ratio.

Importance sampling is a technique used to correct the bias introduced when the behavior policy (μ) differs from the target policy (π).

$$G_t^{\pi} = \prod_{i=t}^{T} \frac{\pi(a_i|s_i)}{\mu(a_i|s_i)} G_t^{\mu} \tag{2}$$

The ratio $\frac{\pi(a|s)}{\mu(a|s)}$ is used to adjust the returns obtained from episodes generated under the behavior policy to estimate the expected return under the target policy. This adjustment helps in learning about the target policy even when episodes are generated using a different policy. The numerator represents the probability of the action to be ϵ-greedily chosen in that particular state, and is computed like this:

$$\pi_k(a|s) = \begin{cases} \frac{\epsilon}{m} + (1-\epsilon) & \text{if } a = \arg\max_{a \in A} Q_{\pi}(s,a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \tag{3}$$

Instead the denominator is the probability of the action to be chosen in that particular state following the exploratory policy which as we know in this case is a random one and thus:

$$\pi_k(a|s) = \frac{1}{Npossibleactions}$$

In fact this technique makes the variance pretty significant, aspects which joined with the structural variance of MC approaches tends to bring the overall variance of the Q table to very very high levels, and this is why the method might need more episodes to find a winning path.

What are the advantages of using Off policy then?

Mainly being able to reuse experience for learning different policies with allegedly different goals without having to collect different experiences for all the policy we want to realize, which for costly environment is a great advantage.

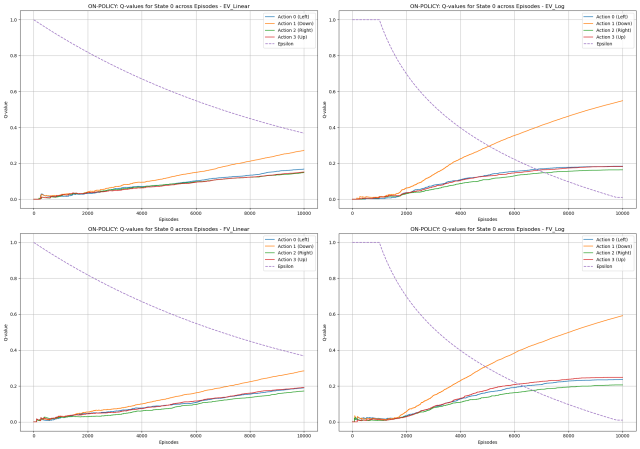
4 Experiments

Following section compare experiments made over tabular implementation in a Monte Carlo fashion way for both On Policy and Off Policy optimization

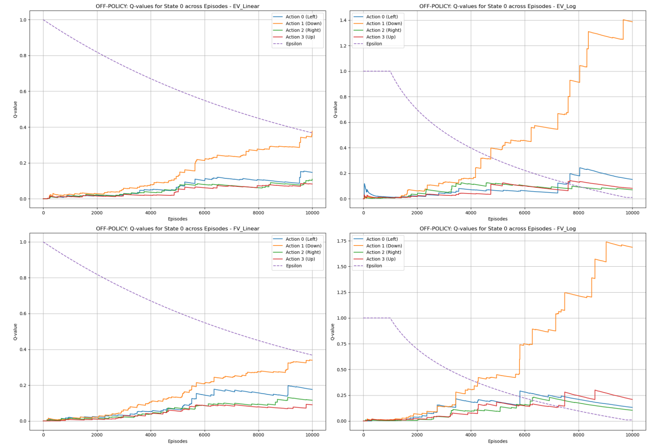
To better appreciate the differences between them, focus is not only on the number of episodes necessary for them to determine an effective Q table but also the variance of the matrix Q values in some interesting cells like the ones we know belongs to the 3 safe paths available in this environment.

4.1 Q table variance analysis

Following plots were obtained using both on and off policy version of the algorithm for most important crossed states of the Q-Table, giving a complete overview of how the agent learns over the time. Since ϵ-greedy approach was exploited, different functions for ϵ decay were used to study how learning changes with a faster one, the logarithmic function, and a lower one which decreases proportionally to the number of episodes. Number of episodes was fixed to a very high number as 10000 to show up as much as possible the way by a state-action pair can converge to higher values.



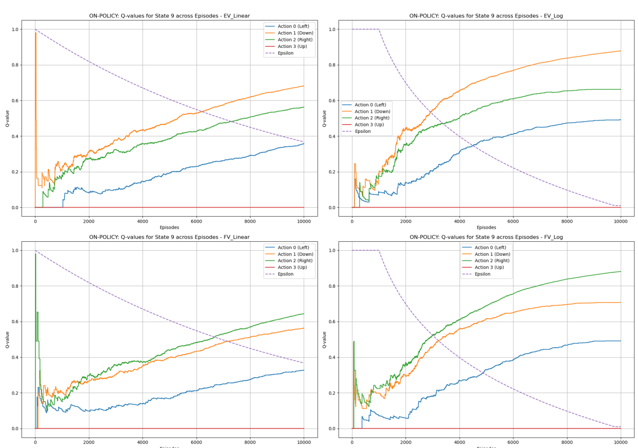
(a) On-Policy plots



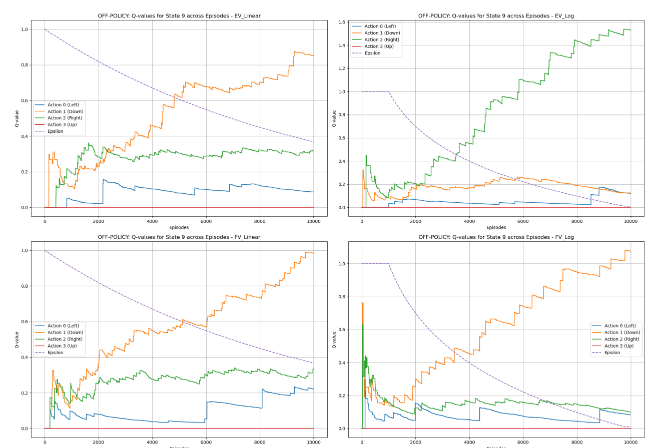
(b) Off-Policy plot

Figure 3: State 0 comparison

As expected, faster decay function of brings higher values since if ϵ decreases then probability to follow exploitation increases therefore the Q-table entries learned are reinforced and agent don't choose randomly anymore. Talking about the trends for On and Off policies we can notice how the latter has a more "saw like" trend, which reflects the noisiness and the variance of the importance sampling as already explored in the theory, while instead the On policy version has a more monotonic trend. We can also notice how the on-policy values not explode as much, this is because the update formula will never return a gain bigger than one, and that being joined with the discount will maintain the overall values controlled. Instead for the off-policy the asymptotic value will head to 4 since decreasing the ϵ the weight coefficient of the importance sampling will end up multiplying the gain (which in the same way as the on policy will never get bigger than one) for a coefficient of 4 given by the exploratory random policy.



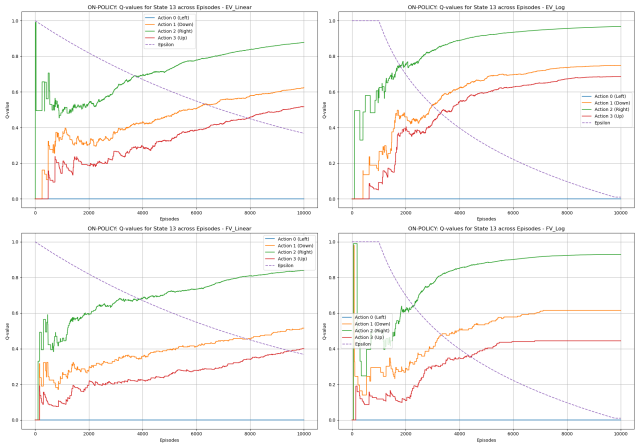
(a) On-Policy plots



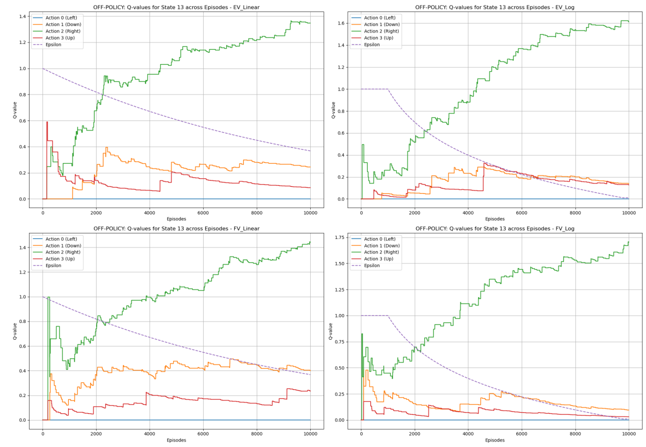
(b) Off-Policy plot

Figure 4: State 9 comparison

As show in figure 3, nor action from state 9 is *more correct* than another one as the agent can both go down or right for reach a two-step goal cell and the choice is mostly random. Up action is the only one with zero constant value as brings to lake cell, Left action has lower values related but not zero since doesn't end episode without reward neither brings closer to the goal. General differences explained before are present in this state too as fast ϵ decay brings step-wise behaviour and the Off-Policy implementation leaves out less-suitable actions before On-Policy action.



(a) On-Policy plots



(b) Off-Policy plot

Figure 5: State 13 comparison

Figure 4 states that agent learns to go on the right in all simulations and as the previous example the only action doesn't raise from zero is Left, since brings into a lake, while the others seems to be still suitable, in particular for the On-Policy implementation. This problem has been faced in mostly all states: while Off-Policy doesn't increase others action's value when finds out a most profitable choice, On-Policy implementation still continue to choose other paths after the optimum one finding due to the fact that the episode collection is in one case performed completely at random by the exploratory policy (so trough the course of the episodes the agent doesn't choose the paths accordingly to how learned so far, which might mean continuing to fall into the same lake multiple times), while in the on-policy one the episodes created in the latter phases will be created using an almost trained policy, which learns what paths to avoid and what to explore creating more useful episodes.

Another interesting observation which can be made for all the above concerns the confidence of the agent decision and the amount of time necessary to achieve it. It appears the Off policy takes a longer time to understand the better action but once it does the

4.2 Obtained Final policies

To better understand the efficiency of these methods here we represented the optimal policy learned at the end of the training process, and we can observe how the algorithms correctly learn at least a winning path trough the goal, the chosen path is more often the "lower" one.

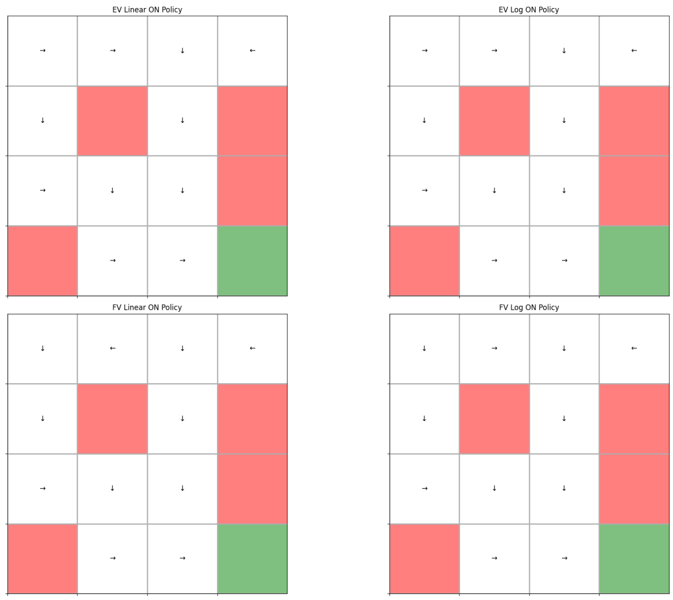


Figure 6: An example of learned policies for the ON policy algorithm

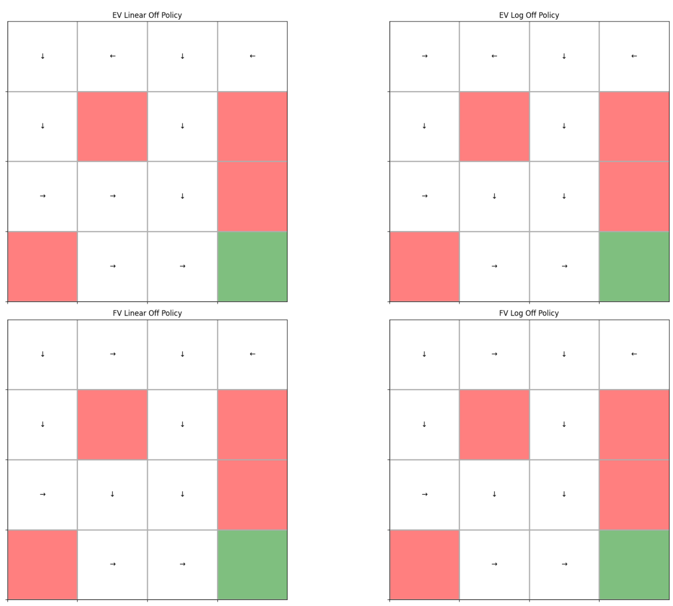


Figure 7: An example of learned policies for the OFF policy algorithm

4.3 Convergence Analysis

Since we have some difference in the variance levels between the on and off policy implementation and above we observed them trough some interesting cells we also wanted to observe more in details the times for which they converged

Results in table 1 were obtained considering:

- Number of experiments: 10
- Maximum number of episodes per experiment: 500
- Number of episode in a row with reward obtained before considering experiment successful: 5

As expected results are mostly comparable and no particular implementations seems to be much better than the others due to simplicity of the problem, given discrete space of both actions and states allow

	Decay function	Average ϵ value	Average episodes needed	Convergence reached
On policy first visit	Exp	0.68	112	7
Off policy first visit	Exp	0.27	286	9
On policy first visit	Linear	0.68	127	8
Off policy first visit	Linear	0.46	372	9
On policy every visit	Exp	0.68	105	8
Off policy every visit	Exp	0.28	282	7
On policy every visit	Linear	0.79	113	8
Off policy every visit	Linear	0.54	306	10

Table 1: Convergence study

We can observe how the convergence takes a longer time in terms of episodes in the off policy case, which is coherent with the variance of the Q table observed in the plot above; and also we can observe how in the linear ϵ decay the convergence is slower compared to the logarithmic decay in both on and off policy optimization.

5 Monte Carlo Tree Search

The idea behind MCTS as mentioned before is an incremental search tree building using simulation of more than one action sequence (called rollouts) from the current state, carry out until a terminal state or a predefined number of steps is reached. The results of these simulations are then backpropagated up the tree, updating the records of the nodes visited at some stage in the play including.

MCTS has been efficiently implemented in numerous domains, including turn based board games (e.g., Go, chess, and shogi), card video games (e.g., poker), and video games[2]. It has done splendid overall performance in lots of challenging recreation-gambling scenarios, frequently surpassing human understanding. MCTS has also been prolonged and tailored to deal with different trouble domains, which include making plans, scheduling, and optimization.

Monte Carlo Tree Search is mainly used for for such purposes even with unknown or imperfect data, as it relies on statistical sampling as opposed to whole know-how of the game state thanks to scalability and efficient parallelization.

5.1 Algorithm Structure

Building block of MCTS search tree are **nodes** which represents a state reched using an action, and the tree is built as outcome of stages described below.

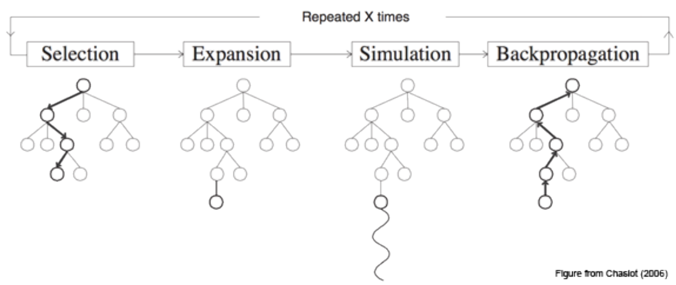


Figure 8: Caption

Algorithm 5 MCTS(env, episodes)

tree ← InitializeTree()
state ← env.reset()
root_node ← CreateNode(state)
tree.add_node(root_node)
for $i = 1$ to episodes **do**
    node ← SelectionPhase(tree, root_node)
    reward ← SimulationPhase(env, node)
    BackpropagationPhase(tree, node, reward)
end for
return ExtractBestPolicy(tree, root_node)

We are now going to analyse more in detail the phases of the algorithm:

1. **Selection**:
   In this step, the MCTS algorithm traverses the current tree from the root node using a specific strategy. During tree traversal, a node is selected based on some parameters that return the maximum value.

Algorithm 6 SelectionPhase(tree, node)

```
while not node.terminal do
    if IsExpandable(node) then
        return Expand(tree, node)
    else
        node ← BestChildByQ(tree, node)
    end if
end while
return node
```

Here is a drawing for better understanding:

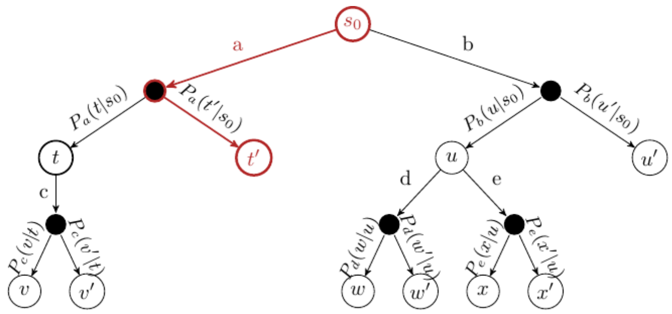


Figure 9: Selection Phase

Selection Metrics
After a predefined number of algorithm runs, simplest way to use roll-out results is the **Pure Monte Carlo Game Search**, which applies same number of roll-out stages after each legal move of current player and than chooses epsilon-greedily the action which may lead to highest number of victories. The problem of this method is that the tree size often increases with time as more play-outs are assigned to the moves that have frequently resulted in the current player's victory according to previous ones. By the way, the action can be also chosen in a *more optimistic way* using an evaluation function for node with the highest estimated value finding. Another possible implementation is called **UCT** (Upper Confidence Bound for Tree search), which implements exploration-exploitation trade-off logic using expected reward estimation.
What is the purpose of such trade-off?
Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. The greedy actions are those that look best at present, but some of the other actions may actually be better. Greedy action selection forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates.

The Pure Monte Carlo approach approach for selecting actions based on the average reward is given by:

$$\bar{Q}_i(t) = \frac{S_i(t)}{n_i(t)}$$

where:

- $\bar{Q}_i(t)$ is the average reward for action i up to time t,
- $S_i(t)$ is the total accumulated reward from action i up to time t,
- $n_i(t)$ is the number of times action i has been chosen up to time t.

The Upper Confidence Bound (UCB) formula is given by:

$$\text{UCB}_i(t) = \bar{Q}_i(t) + \sqrt{\frac{2\ln(t)}{n_i(t)}}$$

the added term is a sort of measure of the of the uncertainty or variance in the estimate of the actions's value where:

- $\bar{Q}_i(t)$ is the average reward of action i up to time t,
- t is the current time step,
- $n_i(t)$ is the number of times action i has been chosen up to time t.

2. **Expansion**:
   In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.

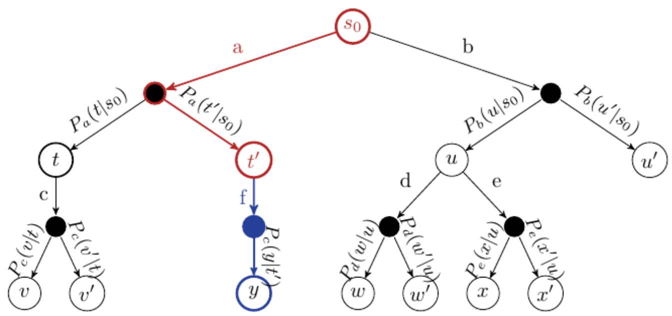


Figure 10: Expansion Phase

Algorithm 7 ExpansionPhase(tree, node)

```
action ← node.untried_action()
new_state, reward, done ← env.step(action)
new_node ← CreateNode(new_state, action, reward, done)
tree.add_node(new_node, node)
return new_node
```

3. **Simulation**:
   In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.

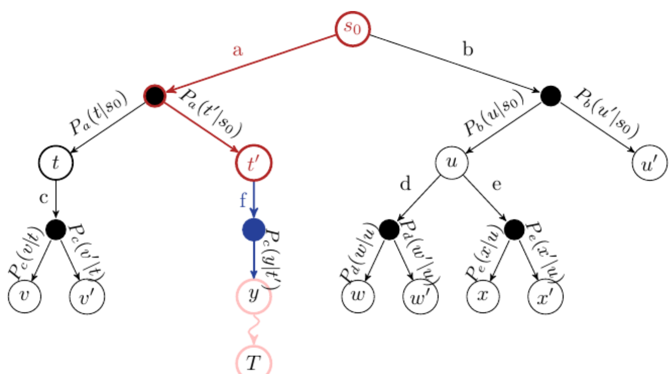


Figure 11: Simulation Phase

4. **Backpropagation**:
   After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it backpropagates from the new node to the root node. During the process, number of simulation stored in each node is incremented and also if new node's simulation results in a win, then this number is incremented too.

Algorithm 8 SimulationPhase(env, node)

if node.terminal **then**
  return node.reward
end if
while True **do**
  action ← RandomAction(env)
  new_state, reward, done ← env.step(action)
  if done **then**
    return reward
  end if
end while

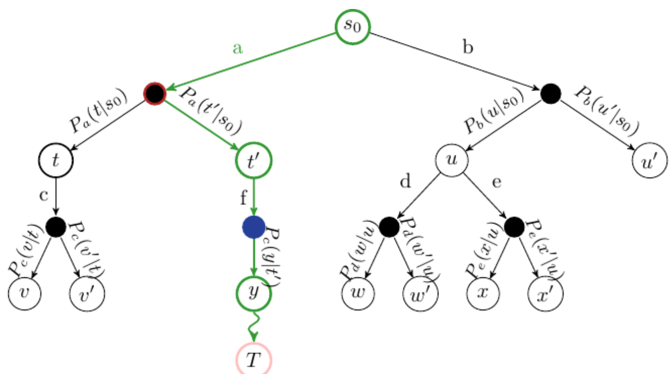


Figure 12: Backpropagation

Algorithm 9 Backpropagation Algorithm

Input: state-action pair (s, a), Q-function Q, rewards G
Output: none

repeat
  $N(s, a) \leftarrow N(s, a) + 1$
  $G \leftarrow r + \gamma G$
  $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}[G - Q(s, a)]$
  $s \leftarrow$ parent of s
  $a \leftarrow$ parent action of s
until $s = s_0$

5.2 Discussion

5.2.1 Advantages of Monte Carlo Tree Search

- **Heuristic Nature:** Monte Carlo Tree Search is a heuristic algorithm which can operate effectively without any domain-specific knowledge apart from the rules and end conditions. It can discover its own efficient moves and learn from them by playing random playouts.
- **State Preservation:** The MCTS can be saved in any intermediate state, which can be utilized in future use cases whenever required. This allows for simpler computation at runtime and is useful for many real applications.

5.2.2 Issues in Monte Carlo Tree Search

- **Exploration-Exploitation Trade-off:** MCTS faces the challenge of balancing exploration and exploitation during the search. It needs to explore different branches of the search tree to gather information about their potential rewards, also exploiting promising actions based on existing knowledge. Achieving the right trade-off is crucial for the algorithm's effectiveness and performance

- **Computation and Memory Requirements:** MCTS can be computationally intensive, especially in games with long horizons or complex dynamics. The algorithm's performance depends on the available computational resources, and in resource-constrained environments, it may not be feasible to run MCTS with a sufficient number of simulations. Additionally, MCTS requires memory to store and update the search tree, which can become a limitation in memory-constrained scenarios.
- **Reliability and speed Issues:**In some scenarios, a single branch or path might lead to a loss against the opposition when implemented for turn-based games. This is mainly due to the high number of combinations, and each node might not be visited enough times to understand its result or outcome in a long run. The MCTS algorithm needs a large number of iterations to effectively decide the most efficient path, which can result in speed issues.

5.3 Experiments

Starting from Naive version, many versions of the algorithm were developed to adapt to different purposes. UCB variant was already compared to the greedy approach, but these are not the only two ways to face tree size problem which makes computation heavier[1]. Overall, a simpler and full tree version were analyzed in both with and without UCT variant:

- Naive MCTS ϵ-greedy
- Naive MCTS UCT

and a more efficient one inspired by a very well-known implementation of MCTS which relies on some model knowledge, in particular on the value function V

- Bootstrapped MCTS
- Bootstrapped MCTS with UCT

For this latter model we will try to also insert some variability in the environment behaviour (which so far behaved deterministically) implementing a custom slippery function to try and obtain more robust results.

For the didactic purpose we have tested the algorithm on a very simple environment which is obviously suitable even to much simper kind of algorithms like the tabular one explained above.

5.3.1 Naive MCTS Implementation

We experimented it for different executions time and analyzed the found path and the tree characteristics, the implementation follows the one explained above.

ϵ-greedy MCTS Implementation
In this case the game tree search will be unique, this is why this version is also called **Full tree MCTS**, of course is evident how this mode of research is quite heavy to carry out for complex games, in the table we tested the algorithm with different number of episodes to understand the rapid growth of the tree.

Steps	Tree Size	Max Depth	Average Depth	Goal Reached
800	450	12	5.46	Yes
1000	641	11	5.69	Yes
5000	1907	14	6.73	Yes
10000	3362	14	7.30	Yes

Table 2: Results of ϵ-greedy MCTS Experiments

We can observe how even in this very simple environment with only 16 states and 4 actions the tree becomes bigger very quickly, in fact we perform almost a full exploration of the action space which of course is not at all the best solution.

Algorithm 10 Monte Carlo Tree Search (MCTS) with UCB1

```
function SELECTION(v)
    while v is completely expanded and not terminal do
        Choose v ← arg max_{v'} ( w(v')/n(v') + c · sqrt( ln n(v) / n(v') ) )
    end while
    return v
end function
```

$$v \leftarrow \arg\max_{v'} \left(\frac{w(v')}{n(v')} + c \cdot \sqrt{\frac{\ln n(v)}{n(v')}} \right)$$

UCT MCTS Implementation

We highlight only the differences with the original implementation which are to be found in the selection phase

This implementation utilizes the exploration exploitation trade-off we talked about before which overall should lead to a better selection of the most promising candidates, we tested it not only with different number of episodes but also with different values of the exploration constant

- For SMALL values of C the model will tend more to **Exploitation** favor actions that have performed well in the past
- For BIG values of C instead it will favour **Exploration** so will explore more, trying out less-frequently chosen actions to gather more information about their potential rewards, even if this means occasionally choosing sub-optimal actions in the short term.

We can observe how in the results below the tree size (in terms of leaves) tends to increase at the increasing of the parameter C value and quickly approaches very high

Exploration Constant	Steps	Tree Size	Max Depth	Avg Depth	Goal Reached
0.5	5000	1428	9	5.1092	Yes
0.707	5000	1745	10	5.2991	Yes
1.4	5000	2849	9	5.5893	Yes
2.0	5000	2975	7	5.5597	Yes
0.5	10000	1009	10	4.8246	Yes
0.707	10000	1378	11	5.1872	Yes
1.4	10000	2741	11	5.7256	Yes
2.0	10000	4374	10	6.0457	Yes

Table 3: MCTS UCT Results varying Episodes and C

COMPARISON

Steps	Tree Size (Table 1)	Tree Size (Table 2)
5000	1907	1428 (C=0.5)
5000	1907	1745 (C=0.707)
5000	1907	2849 (C=1.4)
5000	1907	2975 (C=2.0)

Table 4: Tree Size Comparison at 5000 Steps

Steps	Tree Size (Table 1)	Tree Size (Table 2)
10000	3362	1009 (C=0.5)
10000	3362	1378 (C=0.707)
10000	3362	2741 (C=1.4)
10000	3362	4374 (C=2.0)

Table 5: Tree Size Comparison at 10000 Steps

We can appreciate how for the same number of episodes analyzed a bigger C will lead the tree size to get close to the pure implementation since we get close to an exploratory approach, while using lower coefficient we get almost half the size of the tree still reaching the destination.

Constrained tree comparison
Except for comparing the own characteristics of the tree built of the 2 above implementation which of course will be different due to their algorithmic differences we wanted to compare their performances when their trees are bound to some common constrains like the number of rollout levels allowed and the total depth of the tree (so the maximum number of expansions).

The full tables are available at the end of the paper here we show a aggregation for better understanding of the observation made

Steps	Avg Max Sim Depth	Avg Max Expansions	Goal Reached
800	6.00	6.00	2 / 9
1000	6.00	6.00	2 / 9
3000	6.00	6.00	1 / 9

Table 6: Aggregated results for ϵ-greedy MCTS

Steps	Avg Max Sim Depth	Avg Max Expansions	Goal Reached
800	6.00	6.00	9 / 9
1000	6.00	6.00	9 / 9
3000	6.00	6.00	9 / 9

Table 7: Aggregated results for table 8

The first observation we can make is that the UCT tends to find a successful path in all the analyzed situation despite the constraints, obtaining better average rewards, instead the Pure version needs more exploration and thus the constraints limits it too much. watching again the performances of the ϵ-greedy version we notice how the max depth was more then 10 levels and so with the limitations it cannot work properly. Instead the UCT version can find the optimal policies in all the cases since its trees are more balanced.

5.4 Bootstrapped MCTS

Another approach was then tried, since as observed above the full tree implementation (both with and without optimism) was not scalable even for a small problem like the one we are considering. We can easily imagine how unmanageable the situation could become for a complicated game like chess or Go where the famous AlphaGO was tested, in fact it was this specific algorithm which uses 2 DNN, one for the policy and one for the value approximations that introduced the idea to use value network to evaluate the board positions without completing the full tree to evaluate the states.

This implementation tests a version inspired by this specific algorithm:
main features are maximum level for the tree, which is fixed to two, and the simulation phase that uses V-Table for state's evaluation.

5.4.1 Non-Slippery Environment

This version is very handy because allows to use a partial tree only, reducing consistently the computational weight since instead of simulating at each level until the terminating state we can stop after a short number of depth and evaluate the goodness of the state using a value function, which could be computed beforehand and then used to speed the runtime. In fact is possible to observe that with a much simpler tree we can find a winning path realtime, this assuming we have a previously computed V table available. In a sense it can be seen as speeding up the utilization phase making the training more complicated.

Algorithm 11 Bootstrapped MCTS

```
Initialize environment, value table (V), and set the target state
Set initial state and create an empty path list
Set maximum iterations and iteration count
while target state is not reached and iteration count ¡ maximum iterations do
    Create a new Monte Carlo Tree Search (MCTS) instance with the current state
    Reset the MCTS tree with the initial state
    for each expansion in the MCTS tree (up to max_expansions) do
        Select a node based on ϵ-greedy policy
        if node can be expanded then
            Expand the node by taking an action
        end if
        Simulate from the expanded node and obtain the reward
        Backpropagate the reward through the tree
    end for
    Choose the best action based on the tree's evaluation
    Execute the chosen action in the environment
    Update the current state based on the environment response
    Append the new state to path
    if current state is the target state then
        Break the loop
    else if fell into the lake) then
        Reset environment and path
        Reset iteration count
    else
        Increment iteration count
    end if
end while
```

Both UCT and ϵ-greedy behaviour were observed in selection phase, due to the small nature of the environment and the small size of the tree we could not observe many difference but in more complex games with a lot possible action this will bring the same advantages as in the full tree case.

We will now show you how the 2 versions of the algorithms build the tree, both for better understanding of the

5.4.2 Naive MCTS Tree

Here is a breakdown of all the phases performed by the algorithm for a single iteration

Here is an example of an iteration of the Naive MCTS algorithm with all the for 4 phases highlighted:

Root node:
0: (action=None, visits=0, ratio=0.0000) ▷ Starting node

After Selection:
0: (action=None, visits=0, ratio=0.0000) ▷ Example of Selection

After Expansion:
0: (action=None, visits=0, ratio=0.0000)
↪ 1: (action=2, visits=0, ratio=0.0000) ▷ Example of Expansion

After simulation (reward = 0.9506):
0: (action=None, visits=0, ratio=0.0000)
↪ 1: (action=2, visits=0, ratio=0.0000) ▷ Example of Simulation

After Backpropagation:
0: (action=None, visits=1, ratio=0.9606)
↪ 0: (action=1, visits=1, ratio=0.9606) ▷ Example of Backpropagation

After Selection:
0: (action=None, visits=1, ratio=0.9606)
↪ 1: (action=2, visits=1, ratio=0.9606)
▷ Example of Selection

After Expansion:
0: (action=None, visits=1, ratio=0.9606)
↪ 1: (action=2, visits=1, ratio=0.9606)
↪ 0: (action=0, visits=0, ratio=0.0000)

After simulation (reward = 0.9509):
0: (action=None, visits=0, ratio=0.9606)
↪ 1: (action=2, visits=1, ratio=0.9606)
↪ 0: (action=0, visits=0, ratio=0.0000)

After Backpropagation:
0: (action=None, visits=2, ratio=0.9558)
↪ 1: (action=2, visits=1, ratio=0.9606)
↪ 0: (action=0, visits=1, ratio=0.9510)

For a comparison here are shown the tree for the first and then some more central states, where we could think of observing some differences.

Final tree for state 0:
0: (action=None, visits=8, ratio=0.7193)
  ↪ 1: (action=2, visits=2, ratio=0.4803)
    ↪ 5: (action=1, visits=1, ratio=0.000)
  ↪ 0: (action=0, visits=2, ratio=0.9558)
    ↪ 4: (action=1, visits=1, ratio=0.9606)
  ↪ 4: (action=1, visits=3, ratio=0.6436)
    ↪ 8: (action=1, visits=1, ratio=0.9703)
    ↪ 5: (action=2, visits=1, ratio=0.000)
  ↪ 0: (action=3, visits=1, ratio=0.9510)

Final tree for state 9:
9: (action=None, visits=8, ratio=0.7364)
  ↪ 13: (action=1, visits=2, ratio=0.4950)
    ↪ 12: (action=0, visits=1, ratio=0.000)
  ↪ 8: (action=0, visits=2, ratio=0.9654)
    ↪ 4: (action=3, visits=1, ratio=0.9606)
  ↪ 10: (action=2, visits=3, ratio=0.99)
    ↪ 6: (action=3, visits=1, ratio=0.9801)
    ↪ 14: (action=1, visits=1, ratio=1.00)
  ↪ 5: (action=3, visits=2, ratio=0.000)

Final tree for state 10:
10: (action=None, visits=8, ratio=0.7425)
  ↪ 9: (action=0, visits=1, ratio=0.9801)
  ↪ 14: (action=1, visits=3, ratio=0.9967)
    ↪ 10: (action=3, visits=1, ratio=0.99)
    ↪ 14: (action=1, visits=1, ratio=1.00)
  ↪ 6: (action=3, visits=3, ratio=0.6567)
    ↪ 10: (action=1, visits=1, ratio=0.99)
    ↪ 5: (action=0, visits=1, ratio=0.0.00)
  ↪ 11: (action=2, visits=1, ratio=0.000)

5.4.3 UCT version

This instead are the tree built by the version using UCT

Final tree for state 0:
0: (action=None, visits=8, ratio=0.9570)
  ↪ 4: (action=1, visits=2, ratio=0.9654)
    ↪ 8: (action=1, visits=1, ratio=0.9703)
  ↪ 0: (action=0, visits=2, ratio=0.9510)
    ↪ 0: (action=3, visits=1, ratio=0.9510)
  ↪ 0: (action=3 visits=2, ratio=0.9510)
    ↪ 0: (action=0, visits=1, ratio=0.9510)
  ↪ 1: (action=2, visits=2, ratio=0.9606)
    ↪ 1: (action=3, visits=1, ratio=0.9606)

Final tree for state 9:
9: (action=None, visits=8, ratio=0.6163)
  ↪ 13: (action=1, visits=2, ratio=0.4950)
    ↪ 12: (action=0, visits=1, ratio=0.000)
  ↪ 8: (action=0, visits=2, ratio=0.4851)
    ↪ 12: (action=1, visits=1, ratio=0.000)
  ↪ 5: (action=3, visits=2, ratio=0.000)
  ↪ 10: (action=2, visits=3, ratio=0.990)
    ↪ 14: (action=0, visits=1, ratio=1.00)
    ↪ 6: (action=3, visits=1, ratio=1.9801)

Final tree for state 10:
10: (action=None, visits=8, ratio=0.7401)
↪ 14: (action=1, visits=3, ratio=0.9967)
↪ 10: (action=3, visits=1, ratio=0.990)
↪ 14: (action=1, visits=1, ratio=1.00)
↪ 11: (action=2, visits=1, ratio=0.000)
↪ 9: (action=0, visits=2, ratio=0.9752)
↪ 8: (action=0, visits=1, ratio=0.9703)
↪ 6: (action=3, visits=2, ratio=0.490)
↪ 5: (action=0, visits=1, ratio=0.000)

5.5 Slippery Environment

We also implemented a version with a custom slippery phenomenon (different from the behaviour already available in gym) where when simulating there is a certain probability (we experimented with a 50% probability) of making the same action step twice.

The only difference with the version above will be in the simulation step: of course the agent can slip only the it

Algorithm 12 Simulation Phase

1: **Input:** Node *node*
2: Set environment state to $node.state$
3: If $node.state$ is not terminal
4: If simulation is slippery, take another step in the same direction step
5: Return the value of the new state or current state from the value table V

did not already reach a final state with the first step.

We tried this approach to incorporate the concept or risk in our analysis, so far the methods focused in finding the shortest possible winning path, now instead the slipping property will bring the agent to focus also on the danger of one path respect to the other. To do this we also modified the V table setting the value of the lakes as -1.

$$V = \begin{bmatrix} 0.95099005 & 0.96059601 & 0.97029794 & 0.96059601 \\ 0.96059601 & -1. & 0.9801 & -1. \\ 0.970299 & 0.9801 & 0.99 & -1. \\ -1. & 0.99 & 1. & - \end{bmatrix}$$

These are some run that we performed and for which we will report the slips which lead to lakes to correlate such events with the learned path, for a better understanding of the movements in the maps we will also put a simplified illustration of the path to the destination:

Example run 1

- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 13, action 0, slipped to state 12
- From state 10, action 2, slipped to state 11
- From state 9, action 3, slipped to state 5
- From state 9, action 3, slipped to state 5
- From state 13, action 0, slipped to state 12

- From state 9, action 3, slipped to state 5

The built path towards the destination was the following:

$$[0, 4, 0, 4, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 6, 10, 9, 13, 13, 14, 15]$$

Here is available a simplified drawing of the final path to better visualize the movements

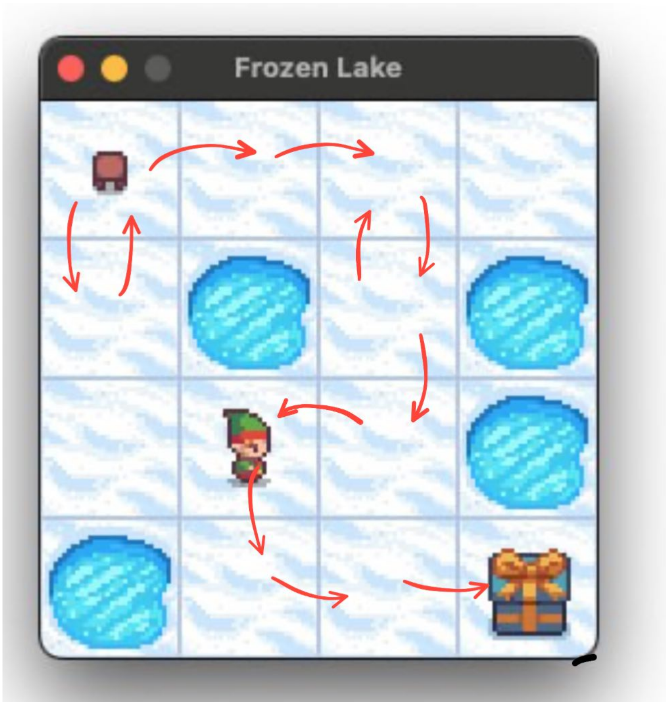


Figure 13: Description of the image

Example run 2

- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 8, action 1, slipped to state 12
- From state 13, action 0, slipped to state 12
- From state 13, action 0, slipped to state 12
- From state 9, action 3, slipped to state 5
- From state 13, action 0, slipped to state 12
- From state 14, action 2, slipped to state 15

The path built towards the destination was the following

$$[0, 0, 4, 4, 0, 1, 2, 2, 2, 6, 2, 6, 10, 9, 10, 9, 8, 9, 10, 14, 15]$$

Here is available a simplified drawing of the final path to better visualize the movements

Figure 14: Description of the image

In all the above examples the agent slips a lot of times into the lake of state 12 and then learns to undergo the upper safe path and this happens for the majority of the simulations.

In reality is also possible to observe how the agent is rather confused at the beginning, this might be due to the used table, which has very high value of V also for the initials state and which may actually mislead the agent into thinking they are better states to go back to.

6 Tables

Steps	Max Sim Depth	Max Expansions	Avg Reward	Goal Reached
800	5	5	0.0113	Yes
800	5	6	0.0097	Yes
800	5	7	0.0085	Yes
800	6	5	0.0094	Yes
800	6	6	0.0111	Yes
800	6	7	0.0134	Yes
800	7	5	0.0059	Yes
800	7	6	0.0092	Yes
800	7	7	0.0105	Yes
1000	5	5	0.0041	Yes
1000	5	6	0.0087	Yes
1000	5	7	0.0089	Yes
1000	6	5	0.0061	Yes
1000	6	6	0.0074	Yes
1000	6	7	0.0075	Yes
1000	7	5	0.0042	Yes
1000	7	6	0.0086	Yes
1000	7	7	0.0087	Yes
3000	5	5	0.0113	Yes
3000	5	6	0.0115	Yes
3000	5	7	0.0134	Yes
3000	6	5	0.0079	Yes
3000	6	6	0.0088	Yes
3000	6	7	0.0156	Yes
3000	7	5	0.0103	Yes
3000	7	6	0.0387	Yes
3000	7	7	0.0107	Yes

Table 8: Simulation results for UCT version

Steps	**Max Sim Depth**	**Max Expansions**	**Avg Reward**	**Goal Reached**
800	5	5	0.0113	No
800	5	6	0.0038	No
800	5	7	0.0013	No
800	6	5	0.0088	No
800	6	6	0.0013	No
800	6	7	0.0050	No
800	7	5	0.0075	Yes
800	7	6	0.0113	No
800	7	7	0.0100	No
1000	5	5	0.0070	No
1000	5	6	0.0040	No
1000	5	7	0.0090	No
1000	6	5	0.0120	No
1000	6	6	0.0030	No
1000	6	7	0.0040	No
1000	7	5	0.0050	Yes
1000	7	6	0.0090	No
1000	7	7	0.0080	No
3000	5	5	0.0060	No
3000	5	6	0.0053	No
3000	5	7	0.0063	No
3000	6	5	0.0080	No
3000	6	6	0.0040	No
3000	6	7	0.0073	No
3000	7	5	0.0140	Yes
3000	7	6	0.0067	No
3000	7	7	0.0070	No

Table 9: Simulation results for Pure version

References

[1] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo Planning". In: *Computer and Automation Research Institute of the Hungarian Academy of Sciences* (2006).

[2] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489. DOI: `10.1038/nature16961`.