For all the questions, the code I proposed is available in Appendix .

Question 1

With the default bonus, **it seems that the algorithm requires approximately 20 episodes to achieve its first success**. Observing the shape of the cumulative average reward (Fig 1), the algorithm does not earn any rewards during the first 20 games, as it is exploring the space. However, **once it secures its first reward, the cumulative average reward begins to increase rapidly for 100 episodes**, until it reaches a plateau. This plateau is close to one, suggesting that most of the time, the agent following the algorithm reaches the top of the hill

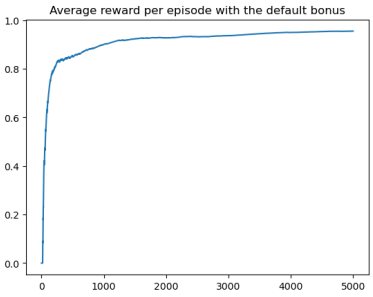

Figure 1: Cumulative average rewards.

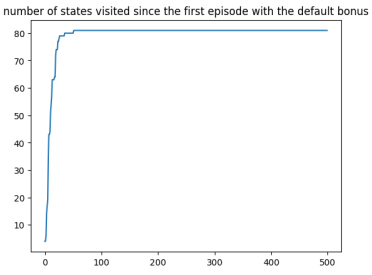

Figure 2: Number of states visited since the beginning.

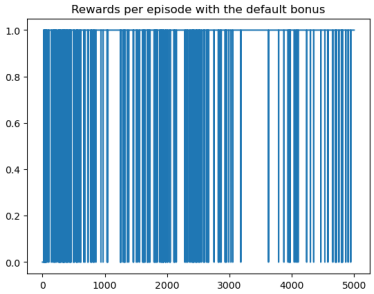

Figure 3: Rewards per episode.

Since UCBVI builds upon the UCBVI algorithm for bandits, comparing plots from a UCBVI algorithm and UCBVI is insightful (plots in Appendix). For UCBVI, we cannot compute regret since we do not have access to the optimal value function for this problem. Therefore, we choose to display average rewards instead. **The primary distinction with the curves typically observed in bandit theory with UCBVI lies in the initial part of the curve, where the agent receives a zero reward**. The fact that the algorithm reaches a plateau quite rapidly is also noteworthy, but this observation must be considered in relation to the horizon of each episode. Indeed, after 200 episodes, the agent has encountered $100,000$ states, averaging 333 per state-action pair. This plateau also needs to be interpreted in light of the high variance in rewards per episode.

Indeed, since UCBVI's primary goal is not to produce optimal policies, we observe variance in the rewards that the algorithm manages to obtain (Fig 3). Even though the agent, acting greedily with respect to the optimistic Q value, begins to solve the game, the bonus—dependent on the number of visits to each state continues to encourage exploration. Consequently, the algorithm persists in exploring states before reaching the goal. Moreover, the algorithm only updates its Q values after the episode concludes, so a poor initialization of the optimistic Q value can result in unsuccessful episodes. This plot also explains why the cumulative average rewards reach a plateau quickly: once the agent achieves its first goal, it begins to frequent states near this location. The number of states visited since the first episode (Fig 2) further illustrates that the algorithm swiftly accesses more than 80 states. However, theoretically, the state space comprises 100 states. This discrepancy suggests that **many states are effectively unreachable**. As discussed in question 2, these include extreme states: states characterized by very high velocity at the board's extremities.

Please note that we choose to halt the algorithm once it reaches the mountain's summit, which explains why the rewards do not exceed 1, in contrast to what might be observed in the original Colab notebook. We opted for this approach as it provides insight about whether of not an agent successfully ascend the hill. Nonetheless, this setting can be altered in our code by simply modifying the *stop_if_done* argument.

Question 2

The goal is to identify a policy that consistently leads to successful task completion while minimizing regret, thereby demonstrating the algorithm's ability to navigate the complexities of the given environment effectively.

The framework for the UCBVI algorithm is centered on regret minimization: unlike the BPI (Best Policy Identification) framework, where the agent's performance is judged solely by the policy it endorses at the conclusion, UCBVI aims to reduce cumulative regret. To still generate viable candidates for the optimal policy, several approaches can be explored:

- **An optimistic policy, acting greedily based on the optimistic Q value estimated at the end of the last episode**. This approach, employed during the UCBVI algorithm's training, maintains an optimistic projection of the expected reward. Nevertheless, in more intricate environments, this method might lead to suboptimal actions if the upper bound is overly optimistic. It persists in exploration even post-training, which could be a double-edged sword.
- **Acting greedily based on the estimated Q value, representing the optimal policy according to the empirically estimated MDP derived from $Rhat$ and $Phat$**. Unlike the optimistic approach, this strategy exclusively leverages prior experiences, which, being sourced from an optimistic exploration, ensures comprehensive exploration of the state space. However, it might prove suboptimal if the state space is insufficiently explored, particularly when encountering uncharted states.
- **A nuanced strategy** could entail acting greedily with respect to the optimistic Q value only if the expected subsequent state has been visited beyond a certain threshold. This nuanced approach adopts an optimistic stance, except in instances of uncertainty about the potential outcome. For example, during testing, if it finds itself in state s with samples from action a_1 but none from action a_2, it might avoid selecting an action that could lead to previously unexplored states.

We attempted to contrast the first two strategies. Given that the state space in our environment is relatively small and most reachable states are explored by the agent, the third strategy appears inapplicable in this context. Due to the similarity in Q value plots, we have decided to include them in Appendix . Instead, we propose showcasing the actions chosen by the agent based on the strategy it adopts (Fig 4 & 5). **For the majority of actions, the outcomes are identical across both strategies**. However, it is noteworthy that for extreme actions (characterized by high velocity at the board's edge), which are less likely to be explored, **the agent adhering to the optimistic policy may select sub-optimal actions as these areas still require exploration**. Conversely, in such states, the agent following the Q value derived from the estimated MDP opts for the optimal action, which is not to move right when the velocity is positive.

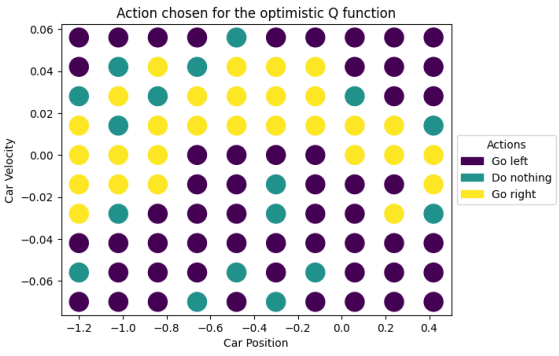

Figure 4: Actions based on the optimistic Q value after 500 episodes of the UCBVI algorithm

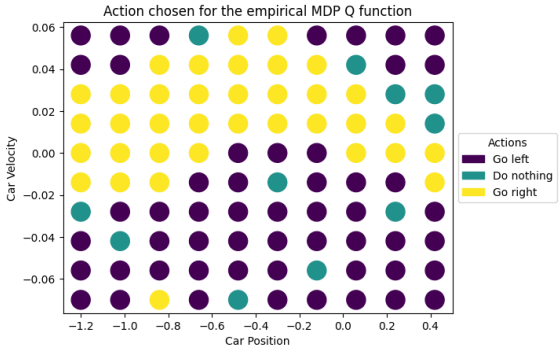

Figure 5: Actions based on the empirical estimated MDP after 500 episodes of the UCBVI algorithm

It is also important to highlight situations where the optimal action is not clear-cut, such as when the car is in the hole and the velocity is low. In these instances, the algorithm faces uncertainty regarding the best course of action. This ambiguity stems from the discretization of the space, leading to scenarios that are inherently equivocal: it is plausible for the actual velocity to be negative, albeit small, yet its corresponding discretized state box is positive.

It is also worth mentioning that **actions taken at the top left and right corners are sub-optimal for both strategies**: when the velocity is exceedingly high, it is more advantageous to move right to ascend the hill. This observation leads to the conjecture that such states are, in fact, unreachable, which could account for the algorithm's inability to access them. Clearly, when reaching a state at the summit of the hill, the velocity of the car is less than what it was at the base. These inaccessible states might influence the algorithm's performance: as they are never visited, their associated bonuses remain high. However, given that the transition probabilities are estimated in real-time, the **algorithm learns that these states are unreachable, as it is a model-based**

algorithm. Consequently, even if the bonus for these states is high, the optimistic Q value for these states remains low.

Question 3

First we recall the definition of the regret of an episodic RL algorithm after T episodes:

$$R_T(\pi) = \sum_{t=1}^{T} (V^\star(s_1^t) - V^{\pi^t}(s_1^t))$$

Let us assume the following regret bound :

$$\mathbb{E}[R_T] \leq \beta_T$$

Let us consider the proof of the bound obtained using Markov's inequality in the context of the UCBVI algorithm. Markov's inequality is given by:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

for a non-negative random variable X and a positive constant a. In our scenario, we define X as the regret $R_T = V^* - V_T^{\hat{\pi}}$, where V^* is the value of the optimal policy, and $V_T^{\hat{\pi}}$ is the value of the policy obtained by UCBVI after T episodes. The expected regret after T episodes is bounded by B_T, hence $\mathbb{E}[R_T] \leq B_T$.
Applying Markov's inequality to bound the probability that the regret per episode exceeds ϵ, we have:

$$\mathbb{P}\left(V^\star - V^{\widehat{\pi}_T} > \epsilon\right) \leq \frac{\mathbb{E}_\tau[\frac{1}{T}\sum_{t=1}^{T}(V^\star - V^{\widehat{\pi}_t})]}{\epsilon} \leq \frac{\mathbb{E}[R_T]}{\epsilon T} \leq \frac{B_T}{\epsilon T}$$

As $\mathbb{E}_\tau[V^{\widehat{\pi}_T}] = \frac{1}{T}\sum_{t=1}^{T} V^{\pi^t}$ where τ is the discrete uniform distribution on $[1, T]$.
The variable to which we applied the Markov inequality is indeed positive, by definition of V^*
To ensure the algorithm outputs an ϵ-optimal policy with probability at least $1 - \delta$, we set:

$$\frac{B_T}{\epsilon T} \leq \delta$$

This gives us the minimum number of episodes T required:

$$T \geq \frac{B_T}{\delta \epsilon}$$

However, this bound still depends on T on the right hand side. This can be fixed noting that the regret is upper bounded by H, but we can do better. The original paper provides an upper bound on the regret, for instance for UCBVI-CH the regret is bounded with probability at least $1 - \delta$, by

$$\text{Regret}(K) \leq 20H^{3/2}L\sqrt{SAK} + 250H^2S^2AL^2,$$

where $L = \ln\left(\frac{5HSAT}{\delta}\right)$, and K the number of episodes.

This upper bound is a bound in probability that can design a bound in expectation, thanks to the calculus of $E[R_T] = \int_0^\infty P(R_T > \epsilon)d\epsilon$. We will not compute this value for this specific algorithm, and use instead another trick, that gives a worst upper bound:

$$\mathbb{E}[R_T] = \mathbb{E}[R_T \mathbf{1}_{R_T < \text{Bound}}] + \mathbb{E}[R_T \mathbf{1}_{R_T > \text{Bound}}] \leq \min(20H^{3/2}L\sqrt{SAK} + 250H^2S^2AL^2, H) + H\delta$$

To get a minimum number of episodes needed by UCBVI to output a ε-optimal policy with probability $1-\delta$, one just need to solve the inequality in T, with the proposed upper bound $T \geq \frac{B_T}{\delta\epsilon}$ In the bound we propose, the constant factors are polynomial instances of the environment which is not a bad bounds, **but the dependency on** δ that appears as $\frac{\ln(\frac{1}{\delta})}{\delta}$ **might be more problematic**.

This upper bound can help to design algorithms that minimize the sample complexity, but those discussions are left to question 7.
This regret bound can be improved. As stated in the original paper, [2], for $T \geq HS^3A$ and $SA \geq H$, this bound translates to a regret bound of $O(H\sqrt{SAT})$. However, in our situation, the first inequality is not achieved.
We also note that despite the sharp regret guarantees, all of the results require estimating and storing the entire transition matrix and thus suffer from unfavorable time and space complexities compared to model-free algorithms.

Question 4

We conducted a comparison of various bonuses for this game. Below is a detailed list of all the bonuses we evaluated:

- **The default bonus**: This is the original bonus that decays according to the square root of the inverse of the number of times a state-action pair has been visited.
- **The first bonus proposed in the original paper**: Introduced in the original paper [2], this bonus is part of the $UCBVI - CH$ algorithm. It is derived from Chernoff-Hoeffding's concentration inequality, providing strong theoretical guarantees in terms of regret.
- **The second bonus proposed in the original paper**: Also introduced in the original paper [2], this bonus is associated with the $UCBVI - BF$ algorithm. It is based on the empirical variance of the estimated next values and utilizes Bernstein's inequality. This approach offers even stronger theoretical guarantees in terms of regret, as it matches the established lower bound $\Omega\sqrt{\text{HSAT}}$ [1] up to logarithmic factors, given $T \geq H^3S^3A$ and $SA \geq H$. However, this first inequality is not satisfied in the settings of our notebook.
- **One binary bonus**: We devised this bonus to incentivize the exploration of previously unvisited states. Given the known reward function, we assign a bonus of 0 if a state has been visited and 100 if it has not. This bonus's visualization is available in Appendix .
- **A re-scaled version of the default bonus**: This bonus logarithmically rescales the original bonus, decaying as $B[s,a] = \sqrt{\frac{\ln(N_{sa}[s,a])}{N_{sa}[s,a]}}$.
- **A bonus closer to a more theoretical one**: We constructed a Q value that facilitates solving the game in an almost optimal manner. We set all values to 0 except for: 1 for the action "Go right" with high velocity; 1 for "Go left" with low velocity; and 1 for "Do nothing" when the velocity is near zero. Using this bonus as a Q value and acting greedily results in quickly solving the game.
- **No bonus at all**: We also tested the algorithm without any bonus to compare its performance against other strategies. The first variant is a zero bonus function, which does not promote exploration whatsoever.
- **A uniform bonus**: This final bonus uniformly encourages exploration across all states, serving as another baseline for comparison.

We finished our experiment trying an ϵ-Greedy strategy, with the default bonus.

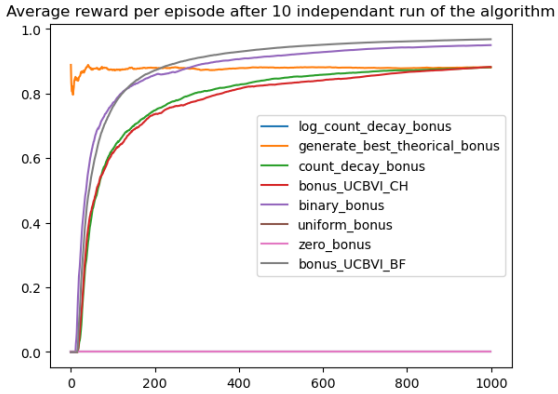

Figure 6: Average cumulative reward

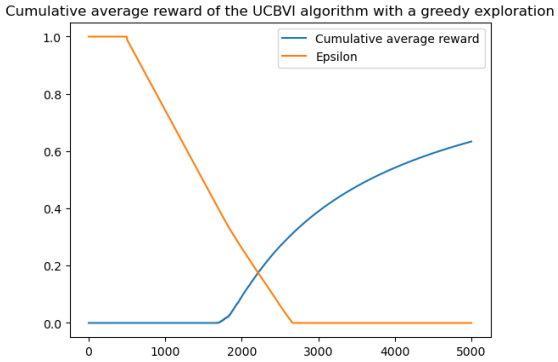

Figure 7: ϵ-Greedy combined with UCBVI with a binary bonus

We can observe that the bonuses which perform the best are **the binary bonus and the second bonus proposed in the original paper** which is the one based on Bernstein inequality (Fig 6). However, for all the bonuses that solve the game, the results are similar. On the other hand, both the **uniform and zero bonus are not sufficient to solve the game**, and never reach the top of the hill. This is not surprising, as these exploration bonuses are considered less effective. **The shapes of all other algorithms are really similar**. The theoretical bonus allows the algorithm to solve the game quickly, but as its bonuses are not decaying, it is challenging for the algorithm to learn, and it may fall into situations where the car is stuck and the best action is to do nothing, depending on the initial state, so it does not encourage the pursuit of other strategies. The ϵ-Greedy version of the algorithm also solves the game, but it takes more time for the algorithm

to learn (Fig 7). We emphasize that **the randomness present in the ϵ-Greedy version is not necessary, as it is supposed to lead to uniform exploration whereas the goal of the UCBVI algorithm is to perform informed exploration**. Therefore, this version is sub-optimal.

Question 5

This task can also be viewed through the lens of a discounted MDP framework. The fixed horizon was necessary for our algorithm model-based UCBVI algorithm as we need it for the back-propagation function. However, for our specific problem of Mountain Car, the goal is to get the car up the hill. Once this task is achieved, the learner receive a reward and starts to learn. It is likely that at the beggining of the learning, the learners needs many steps before reaching the top. If the discount factor is too law, the reward that will be associated to other states will be too law for the learner to efficently learn. Indeed, a low discount factor would reduce the value of the hill's reward too much, making the algorithm act as though it's operating with a more immediate time horizon. That is why **we need to set the discount factor close to 1, at 0.99 for instance**. Oppositely, using a higher discount factor, the algorithm is encouraged to value future rewards more, which is crucial for motivating the car to continue its effort to reach the hilltop.

We also choose to go beyond the question trying to think about other algorithms to which the idea of replacing rewards, by rewards plus a bonus can be applied.

- **Fitted-Q Iteration** is a batch reinforcement learning algorithm that approximates the Q-function, which represents the expected returns of taking an action in a given state, through supervised learning. By modifying the reward signal to include an exploration bonus, the algorithm can be encouraged to explore areas of the state-action space that are less known or that it is uncertain about. The exploration bonus would typically decrease as the algorithm becomes more confident about the value estimates of state-action pairs, reflecting the decreasing benefit of exploring those areas further.
- **Monte Carlo Tree Search** where an exploration bonus is already used in the selection phase (such as UCT algorithm) to balance exploration and exploitation when choosing which node to explore next. The bonus encourages exploration of less-visited nodes.
- **Deep Q-Networks** where an exploration bonus could be added to the reward function to encourage the network to explore uncertain state-action pairs. It is however more difficult to set the bonus function, as the state space is often continuous. For continuous cases, estimating a bonus online with another neural network could be investigated, or using methodologies that integrates an upper confidence bound adding noise to the model like in NoisyNet [7] could be studied.
- **Thompson Sampling**, although not a Q-learning based method, Thompson Sampling in the context of reinforcement learning selects actions based on samples from posterior distributions of the model's parameters, inherently balancing exploration and exploitation. An exploration bonus can adjust the degree of exploration by influencing the sampling process. It is also worth noting that leveraging Bayes-UCB for multi-armed bandits [5], it is possible to build the Bayes-UCBVI algorithm [9] without adding bonus or noise, but leveraging the same principles.

Question 6

We coded a Q-learning with UCB Exploration algorithm, based on a paper describing this idea [3]. We just modified the way to choose the the bonus, as the one proposed in the original paper produced too large bonuses. Instead, for each state-action, we add the bonus $\frac{\ln(N_{sa}[s,a])}{N_{sa}[s,a]}$. Except the bonus, the code for a classic Q-Learning algorithm is used. The code is available in appendix, but the update we perform is :

$$Q(s,a) \leftarrow Q(s,a) + \alpha_t \left[r + bonus + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where the learning rate $\alpha_t = \frac{H+1}{H+N_{sa}(t)}$ is fine tuned online for each state action pairs. The code is available in Appendix.

We compared two different strategies : The QLearning with UCB, and a classic QLearning. The curve we observe (Fig 8 is **similar to the one we already observed for a UCBVI algorithm with ϵ-Greedy exploration** (Fig 7). We can note that the algorithm starts to collect samples thanks to the exploration induced by the parameter ϵ, but also visits new states thanks to the optimistic bound (Fig 9). We also note that the bonus highly encourages to reach the top of the hill. Indeed, with a random initialisation of the Q Table and **without**

any bonus, the Q Learning algorithm is not able to solve the game (Fig 10), and even if it visits 50 states of the board (Fig 11), it is not enough to get a non-zero reward.

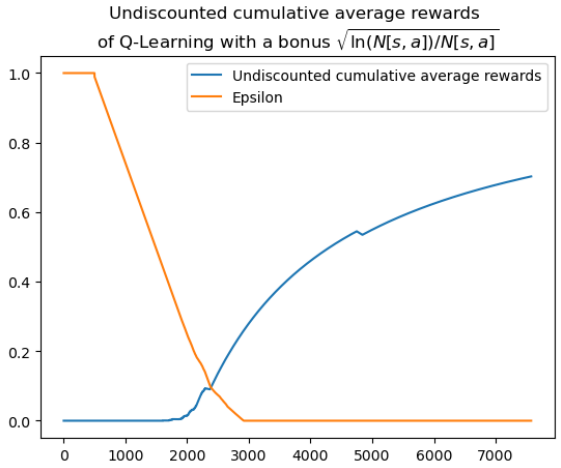

Figure 8: Average cumulative reward of Q-Learning algorithm with bonuses

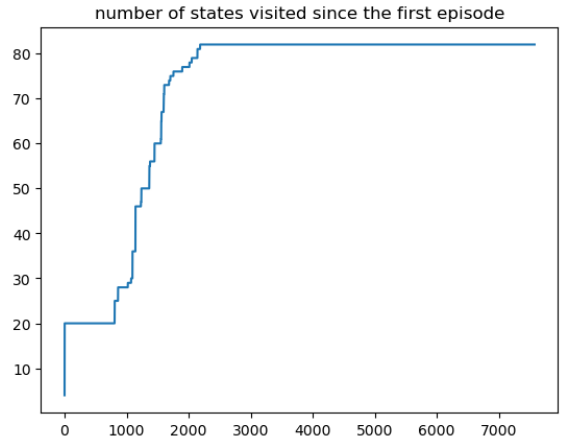

Figure 9: Number of states visited since the first episode

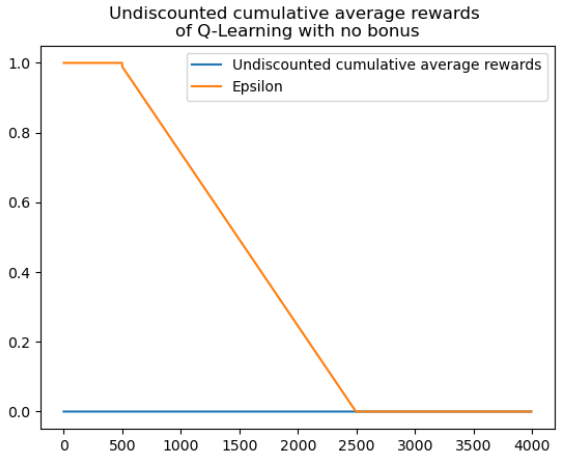

Figure 10: Average cumulative reward of Q-Learning algorithm without bonuses

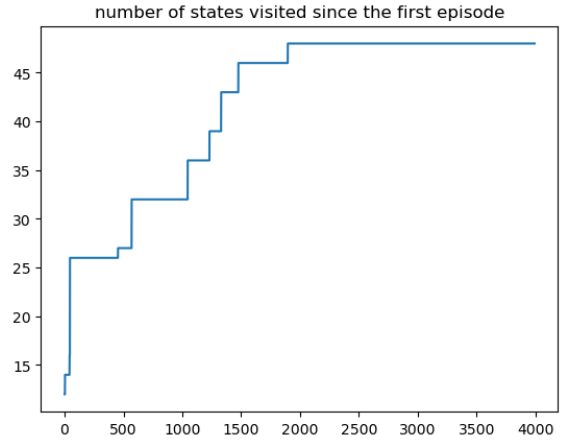

Figure 11: Number of states visited since the first episode

Question 7

The aim of this question is to determine **"How many samples are required to learn a sufficiently good policy?"**. Our objective is for our algorithm to return a policy $\hat{\pi}_\tau$ after a minimal number of episodes τ that satisfies

$$\mathbb{P}\left(\mathbb{E}_{s_1 \sim P_0}[V^\star(s_1) - V^{\hat{\pi}_\tau}(s_1)] \leq \epsilon\right) \geq 1 - \delta. \tag{1}$$

Given our algorithm, we need only to construct the stopping rule that should be as minimal as possible. To achieve this, several approaches can be explored. Depending on the bonus chosen, one approach is to investigate some lower bounds of the value for each state-action pair, allowing the algorithm to stop when all lower bounds are close to the upper bounds (indicating when the bonus is smaller than a certain threshold). This stopping rule could be expressed as

$$\tau = \inf\{t \in \mathbb{N}, \forall (s,a), \text{bonus}[s,a] < \epsilon\}$$

However, considering that some states are unreachable, leading to a constant bonus, another approach could be to concentrate on the variation of the empirical value functions, proposing a stopping rule such as

$$\tau = \inf\{t \in \mathbb{N}, ||Q^{t+1} - Q^t|| \leq \epsilon\}$$

Further theoretical investigation is required for this kind of stopping rule.
One could also use the upper bound we proved in question 3 to build a stopping rule. Indeed

$$\mathbb{P}\left(V^\star - V^{\widehat{\pi}_T} > \epsilon\right) \leq \frac{\min(20H^{3/2}L\sqrt{SAK} + 250H^2S^2AL^2, H) + H\delta}{\epsilon T}$$

Choosing δ and T that ensures the algorithm satisfy the requirement expressed in equation 1, we can get a stopping rule and a policy by choosing uniformly at random among all the policies executed by the agent.

Let us discuss on both sample complexity of the model-free algorithm to that of the model-based algorithm. It has been proven that for specific instances of both algorithms, UCBVI achieves a regret of $\tilde{O}(\sqrt{H^2SAT})$ and Q-Learning with UCB based on a Bernstein upper bound of $\tilde{O}(\sqrt{H^4SAT})$ [4]. Therefore, using the last strategy, one could expect to get closed stopping rule, up to a factor H from both algorithms, **with a better sample complexity for the UCBVI algorithm**. The UCBVI algorithm has no exploration parameter to tune, unlike the optimistic Q-Learning that relies on a carefully chosen strategy for the ϵ decay strategy. However, it is a model-based algorithm that requires many samples in order to approximate the real MDP closely.

It is also worth noting that some modifications of the UCBVI algorithms can help to build better version in terms of sample complexity of this algorithm, such as BPI-UCBVI [8].

Empirically, since the problem we are studying is a special one where all rewards are 0 except for the top of the hill, one could also investigate the robustness of both algorithms, and output a policy once it consider that the algorithm has indeed converged. For instance, the stopping rule could be : "If the last 10 episodes reached the hill, choose a policy at random among those 10 episodes.".

References

[1] Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. *Advances in neural information processing systems*, 21, 2008.

[2] Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. Minimax regret bounds for reinforcement learning. In *International conference on machine learning*, pages 263–272. PMLR, 2017.

[3] Kefan Dong, Yuanhao Wang, Xiaoyu Chen, and Liwei Wang. Q-learning with ucb exploration is sample efficient for infinite-horizon mdp. *arXiv preprint arXiv:1901.09311*, 2019.

[4] Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I Jordan. Is q-learning provably efficient? *Advances in neural information processing systems*, 31, 2018.

[5] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On bayesian upper confidence bounds for bandit problems. In *Artificial intelligence and statistics*, pages 592–600. PMLR, 2012.

[6] Emilie Kaufmann, Pierre Ménard, Omar Darwiche Domingues, Anders Jonsson, Edouard Leurent, and Michal Valko. Adaptive reward-free exploration. In *Algorithmic Learning Theory*, pages 865–891. PMLR, 2021.

[7] Bilal Piot Jacob Menick Ian Osband Alex Graves Vlad Mnih Remi Munos Demis Hassabis Olivier Pietquin Charles Blundell Shane Legg Meire Fortunato, Mohammad Gheshlaghi Azar. Noisy networks for exploration. In *International Conference on Learning Representations*, 2018. arXiv:1706.10295 [cs.LG].

[8] Pierre Ménard, Omar Darwiche Domingues, Anders Jonsson, Emilie Kaufmann, Edouard Leurent, and Michal Valko. Fast active learning for pure exploration in reinforcement learning. In *International Conference on Machine Learning*, pages 7599–7608. PMLR, 2021.

[9] Daniil Tiapkin, Denis Belomestny, Éric Moulines, Alexey Naumov, Sergey Samsonov, Yunhao Tang, Michal Valko, and Pierre Ménard. From dirichlet to rubin: Optimistic exploration in rl without bonuses. In *International Conference on Machine Learning*, pages 21380–21431. PMLR, 2022.

Appendix

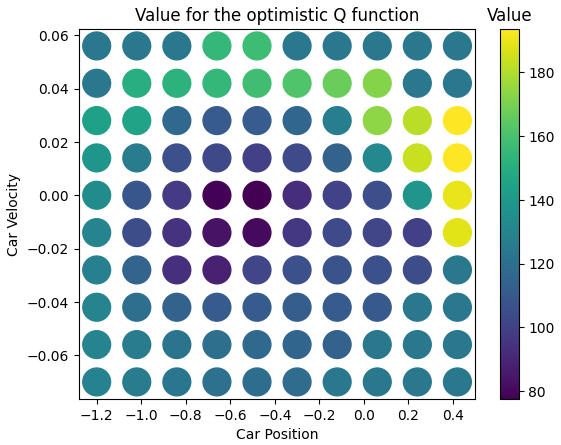

Figure 12: Optimistic Q values after 500 episodes of the UCBVI algorithm

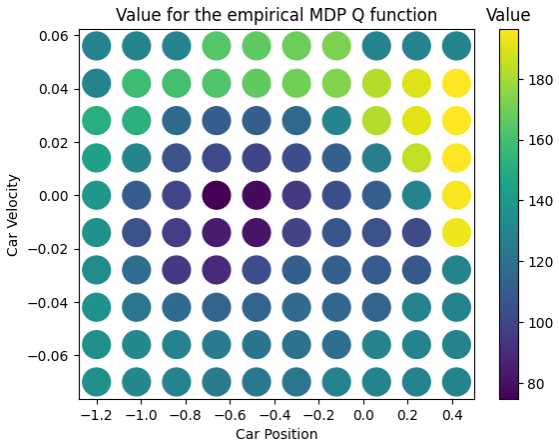

Figure 13: Q values based on the empiracal estimated MDP after 500 episodes of the UCBVI algorithm

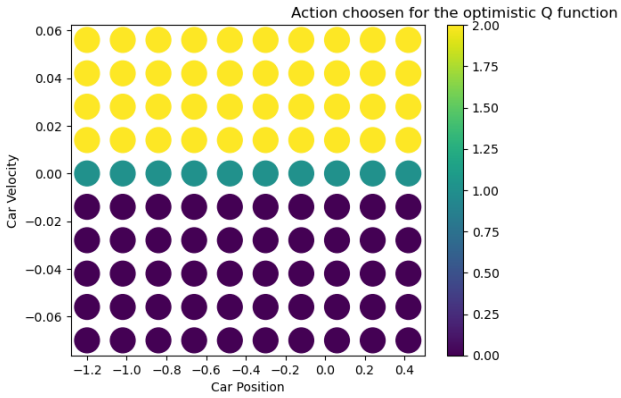

Figure 14: The action chosen based on the bonus that mimic a close-to-optimal policy.

```
def backward_induction(P, R, H, gamma=1.0):
    S, A = P.shape[0], P.shape[1]
    V = np.zeros((H + 1, S))
    Q = np.zeros((H+1,S,A))
    for h in range(H-1, -1, -1):
        for s in range(S):
            # compute the value
            for a in range(A):
              Q[h, s, a] = R[s, a] + gamma * np.sum(P[s, a, :] * V[h+1, :])
            V[h,s] = np.max(Q[h,s,:])
            # ... and clip it (needed later in UCBVI)
            if (V[h, s] > H - h):
                V[h, s] = H - h
    return Q

def UCBVI(env,H, nb_episodes,verbose="off",bonus_function=bonus,gamma=1,
    stop_if_done = True, use_eps = False, eps_val = 0.1):
    S = env.observation_space.n
    A = env.action_space.n
    Phat = np.ones((S,A,S)) / S
    Rhat = np.zeros((S,A))
    N_sa = np.zeros((S,A), dtype=int) # number of visits
```

```
    N_sas = np.zeros((S,A,S), dtype=int) # number of transitions
    S_sa = np.zeros((S,A)) # cumulative rewards
    episode_rewards = np.zeros((nb_episodes,))
    states_visited = np.zeros((nb_episodes,))

    T = nb_episodes * H

    optimistic_Q = np.random.rand(H+1, S, A)

    for k in range(nb_episodes):

        sum_rewards = 0
        state, _ = env.reset()

        ### TO BE COMPLETED: RUN AN EPISODE OF UCV-VI
        optimistic_Q = backward_induction(Phat, Rhat + bonus_function(N_sa,
            S, A, T, H), H, gamma=1.0)
        state, _ = env.reset()
        for h in range(H):
            if use_eps:
                if np.random.rand() < eps_val:
                    action = env.action_space.sample()
                else:
                    action = np.argmax(optimistic_Q[h, state,:])
                eps_val
            else:
                action = np.argmax(optimistic_Q[h, state,:])
            next_state, reward, done, trunc, info = env.step(action)
            N_sa[state, action] += 1
            N_sas[state, action, next_state] += 1
            S_sa[state, action] += reward
            Phat[state, action, :] = N_sas[state, action, :] / N_sa[state,
               action]
            Rhat[state, action] = S_sa[state, action] / N_sa[state, action]
            sum_rewards += reward

            state = next_state
            if stop_if_done and done:
                break

        episode_rewards[k] = sum_rewards
        states_visited[k] = (N_sa.sum(axis=1) > 0).sum()

        if (verbose=="on"):
          # periodically display the rewards collected and visited states
          if k % 50 == 0 or k==1:
            print("rewards in episode {}: {}".format(k, episode_rewards[k])
                , end = ", ")
            print("Number of visited states: ", states_visited[k] )
    optimistic_Q = backward_induction(Phat, Rhat + bonus_function(N_sa, S,
       A, T, H), H, gamma=1.0)
    return episode_rewards, states_visited, N_sa, Rhat, Phat,optimistic_Q


def count_decay_bonus(N, S=None, A=None, T=None, env = ScaleRewardWrapper(
   get_discrete_mountain_car_env()), delta=None, c=None, H=None, nn=None):
    nn = np.maximum(N, 1)
    return np.sqrt(1.0/nn)

def log_count_decay_bonus(N, S=None, A=None, T=None, env =
   ScaleRewardWrapper(get_discrete_mountain_car_env()), delta=None, c=None
```

```
    , H=None, nn=None):
    nn = np.maximum(N, 1)
    return np.sqrt(np.log(nn)/nn)

def bonus_UCBVI_CH(N, S, A, T, H, env = ScaleRewardWrapper(
    get_discrete_mountain_car_env()), delta=1e-3, c=7.0):
    nn = np.maximum(N, 1)
    L = np.log( 5.0 * S * A * T / delta )
    bon = c * H * L * np.sqrt(1.0/nn)
    return bon/np.max(bon)

def binary_bonus(N, S=None, A=None, T=None, env = ScaleRewardWrapper(
    get_discrete_mountain_car_env()), delta=None, c=None, H=None, nn=None):
    nn = np.minimum(N, 1)
    return 1.0 - nn

def normalize_binary_bonus(N, S, A, T, H, env = ScaleRewardWrapper(
    get_discrete_mountain_car_env()), delta=1e-3, c=7.0):
    nn = np.minimum(N, 1)
    number_of_states_visited = (N.sum(axis=1) > 0).sum()
    number_of_states_visited = np.minimum(number_of_states_visited, 1)
    return (1.0 - nn) / number_of_states_visited

def uniform_bonus(N, S=None, A=None, T=None, H=None, env =
    ScaleRewardWrapper(get_discrete_mountain_car_env()), delta=1e-3, c=1.0)
    :
    return c*np.ones_like(N)

def zero_bonus(N, S=None, A=None, T=None, H=None, env = ScaleRewardWrapper(
    get_discrete_mountain_car_env()), delta=1e-3, c=7.0):
    return np.zeros_like(N)

def generate_best_theorical_bonus(N, S=None, A=None, T=None, H=None, env =
    ScaleRewardWrapper(get_discrete_mountain_car_env()), delta=1e-3, c=7.0)
    :
    bonus_opti = np.zeros((S, A))
    for s in range(S):
        p = env.discretized_states[0, s]
        v = env.discretized_states[1, s]
        margin = 0.01
        if v > margin :
            bonus_opti[s,2] = 1000
        elif v > -margin:
            bonus_opti[s,1] = 1000
        else:
            bonus_opti[s,0] = 1000
    return bonus_opti

def bonus_paper_2(N, S=None, A=None, T=None, H=None, N_prim=None, V=None, P
    =None, delta = 1e-2):
    L = np.log(5 * S* A * T / delta)

    bonus_val = np.zeros((S, A))

    nn = np.maximum(N, 1)
    nn_prim = np.maximum(N_prim, 1)

    for s in range(S):
        for a in range(A):
            temp_var = []
            for s_ in range(S):
```

```
                c = V[(h+1,s_)] * P[(s,a,s_)]
                temp_var.append(c)
            var = np.var(temp_var)
            term1 = np.sqrt(8.*L*var/nn[s,a])
            term2 = 14.*H*L/(3.*nn[s,a])
            sum_y = 0
            for y in range(S):
                sum_y += P[s,a,y] * min( 100**2*H**3*S**2*A*L**2 / nn_prim[
                    h, s, a] )
            term3 = np.sqrt( (8*sum_y) /nn[s,a] )
            term3 = term2 * 0.0
            bonus_val[s,a] += term1 + term2 + term3

    return bonus_val / np.max(bonus_val)

def Q_Learning(env, H=500, delta = 1e-3, gamma = 0.99, verbose = False,
    eps_min = 0.01, eps_decay = 2_000, delay_decay = 500, use_bonus = True)
    :
    S = env.observation_space.n
    A = env.action_space.n
    N_sa = np.zeros((S,A))
    Vinit = np.zeros((env.observation_space.n))
    episode_rewards = []
    h = 0
    ep = 0
    Qql = np.zeros((env.observation_space.n,env.action_space.n))
    Qql[:,2] += 1.
    count = np.zeros((env.observation_space.n,env.action_space.n)) # to
        track update frequencies
    max_steps = int(2e6)
    eps_val = 1
    delay_decay *= H
    eps_decay *= H
    alpha = 0.01
    error = np.zeros((max_steps))
    tab_eps = []
    x,_ = env.reset()
    ep_r = 0
    states_visited_tab = []
    for t in tqdm(range(max_steps)):
        h += 1
        if t > delay_decay:
            eps_val = ( 1 - min((t-delay_decay)/eps_decay,1) ) * (1-eps_min
                )
            # print(t, eps_val)
        a = epsilon_greedy(env,Qql,x,eps_val)
        y,r,d,_,_ = env.step(a)

        N_sa[x,a] += 1
        i_k = np.log(S*A*(N_sa[x,a] + 1)*(N_sa[x,a]+2)/delta)
        bonus = 4*1.41/(1-gamma) * np.sqrt(H*i_k / N_sa[x,a])
        if use_bonus:
            bonus = np.sqrt(np.log(N_sa[x,a])/N_sa[x,a])
        else:
            bonus = 0.
        alpha = (H+1)/(H+N_sa[x,a])
        # print(bonus)
        Qql[x][a] = Qql[x][a] + alpha * (r + bonus +gamma*np.max(Qql[y][:])
            -Qql[x][a])
        count[x][a] += 1
        error[t] = np.max(np.abs(Qql))
        ep_r += r
```

```
    if d==True or h>H:
        ep += 1
        tab_eps.append(eps_val)
        states_visited = (N_sa.sum(axis=1) > 0).sum()
        states_visited_tab.append(states_visited)
        episode_rewards.append(ep_r)
        ep_r = 0
        x,_ = env.reset()
        h = 0
        if verbose:
            # periodically display the rewards collected and visited
               states
            if ep % 200 == 0 or ep==1:
                print("rewards in episode {}: {}".format(ep,
                   episode_rewards[-1]), end = ", ")

                print("Number of visited states: ", states_visited )
                print("eps = ", eps_val)
    else:
        x=y

return episode_rewards, tab_eps, Qql, states_visited_tab

def UCBVI_prim(env,H, nb_episodes,verbose="off",bonus_function=
   bonus_paper_2,gamma=1, stop_if_done = False):
 # Code to run the second bonus, we also return the value V in the
    backward_induction_prim
S = env.observation_space.n
A = env.action_space.n
Phat = np.ones((S,A,S)) / S
Rhat = np.zeros((S,A))
N_sa = np.zeros((S,A), dtype=int) # number of visits

N_sas = np.zeros((S,A,S), dtype=int) # number of transitions
N_prim = np.zeros((H,S,A,S), dtype=int)
S_sa = np.zeros((S,A)) # cumulative rewards
episode_rewards = np.zeros((nb_episodes,))
states_visited = np.zeros((nb_episodes,))

T = nb_episodes * H

optimistic_Q, V = backward_induction_prim(Phat, Rhat + bonus_function(
   N_sa, S, A, T, H, N_prim, np.ones((H+1, S)), Phat), H, gamma=1.0)

for k in range(nb_episodes):

    sum_rewards = 0
    state, _ = env.reset()

    ### TO BE COMPLETED: RUN AN EPISODE OF UCV-VI
    state, _ = env.reset()
    for h in range(H):
        action = np.argmax(optimistic_Q[h, state,:])
        next_state, reward, done, trunc, info = env.step(action)
        N_sa[state, action] += 1
        N_sas[state, action, next_state] += 1
        S_sa[state, action] += reward
        Phat[state, action, :] = N_sas[state, action, :] / N_sa[state,
           action]
        Rhat[state, action] = S_sa[state, action] / N_sa[state, action]
        sum_rewards += reward
```

```
            state = next_state
            if stop_if_done and done:
                break

        episode_rewards[k] = sum_rewards
        states_visited[k] = (N_sa.sum(axis=1) > 0).sum()

        if (verbose=="on"):
          # periodically display the rewards collected and visited states
          if k % 50 == 0 or k==1:
            print("rewards in episode {}: {}".format(k, episode_rewards[k])
                , end = ", ")
            print("Number of visited states: ", states_visited[k] )
            # print(V[0, :])

        optimistic_Q, V = backward_induction_prim(Phat, Rhat +
            bonus_function(N_sa, S, A, T, H, N_prim, V, Phat), H, gamma
            =1.0)

    return episode_rewards, states_visited, N_sa, Rhat, Phat,optimistic_Q
```