Final Project on Network Friendly Recommendations

Christos Konstantas

July 5, 2023

Contents

1 Problem Description

In this problem, we focus on providing network-friendly recommendations to a user who interacts with the environment by selecting one of the agent's recommendations or by searching for a specific content $i \in \{1, \ldots, k\}$. We analyze the problem using a Markov Decision Process (MDP) framework characterized by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$.

For the Network Friendly Recommendations problem, the state space $\mathcal{S} = \{0, \ldots, k\}$ represents the entire catalog of k contents and k is the terminal state. To provide a batch of N recommendations when needed, we define the action space as:

$$\mathcal{A} = \{(i_1, ..., i_N) \mid i_j \in \{1, 2, 3, ..., k\}, i_j \neq i_m \text{ for } j \neq m, N \in \mathbb{Z}^+\} \tag{1}$$

Each action in $\mathcal{A}$ consists of a tuple of N distinct elements selected from the set $\{1, 2, 3, ..., k\}$, ensuring uniqueness within each batch $\tilde{a}$. The size of the action space $|\mathcal{A}|$ is given by the binomial coefficient $\binom{k-1}{N}$, representing the number of ways to choose N distinct elements from a set of $k-1$ remaining elements.

To determine the relevance of recommended content, we introduce a threshold constant $u_{\min}$ and a relevance matrix U. If an element u_{ij} in U exceeds $u_{\min}$, the recommended content is considered relevant. The action corresponds to a batch of N recommendations sampled from $\mathcal{A}$. We define the

transition probabilities as:

$$P_{ss'}^{\tilde{a}} = \begin{cases} 0 & \text{if } s' = s \\ q & \text{if } s' = k \\ (1-q)\left(\frac{a}{N} + \frac{1-a}{k-1}\right) & \text{if relevant}(\tilde{a}) \text{ and } s' \in \tilde{a} \\ (1-q)\left(\frac{1-a}{k-1}\right) & \text{if relevant}(\tilde{a}) \text{ and } s' \notin \tilde{a} \\ (1-q)\left(\frac{1}{k-1}\right) & \text{otherwise} \end{cases} \tag{2}$$

These transition probabilities represent the likelihood of transitioning from state s to state s' given the action, i.e.the batch selected. In equation (2), the constants a and q define the probabilities associated with user behavior. The probability q determines the likelihood of the user ending the session, while the probability $1-q$ represents the likelihood of the user continuing the session. Also a represents the probability of selecting a relevant content, N is the batch size, k is the total number of contents, and $\tilde{a} \in \mathcal{A}(s)$ is a set with N non-repeated elements that is a subset of $\mathcal{A}$, and it can describe the set of N recommended contents (may be relevant or not, or maybe give some cost to the system etc) for state s.

In order to avoid recommending the same content that the user watches while simultaneously ensure that we will recommend only contents that are cached we must create and modify carefully the reward function $\mathcal{R}$.

$$R^{\tilde{a}}(s,s') = \begin{cases} 10^7 & \text{if } s \leq k-1 \text{ and } s' \in \tilde{a} \text{ and } s' = s \\ 1 & \text{if } s \leq k-1 \text{ and } s' \in \tilde{a} \text{ and } s' = k \text{ and } s' \neq s \\ \boxed{-1} & \text{if } s \leq k-1 \text{ and } s' \in \tilde{a} \text{ and } s' \neq s \text{ and cached\_matrix}(s,s') = 0 \\ 1 & \text{if } s \leq k-1 \text{ and } s' \in \tilde{a} \text{ and } s' \neq s \text{ and cached\_matrix}(s,s') \neq 0 \\ -1 & \text{if } s' \notin \tilde{a} \text{ and } s' \neq s \\ \text{(undefined)} & \text{if } s > k-1 \text{ terminal state} \end{cases} \tag{3}$$

The cached_matrix above is a kxk matrix in which every row has $C = n \cdot k$ cached contents, where n $\in (0,1)$ and if user watches $s = i$ and the next content she watches is $s' = j$ and is cached then cached_matrix$(s,s') = 0$ else (s' non-cached) cached_matrix$(s,s') = 1$. Our recommendation system must exploit the boxed value of -1 in (3) by applying the Bellman equations iteratively. The Bellman equations are used to update and optimize the state value function V_π and/or the state-action value function Q_π under a certain policy π. By examining the reward function, we can observe that we are dealing with a minimization problem. Therefore, the algorithms employed will aim to minimize the state value function V_π and/or the state-action value function Q_π under a given policy π.

In summary, the modified reward function (3) takes into account the constraints of avoiding repeated content and recommending only cached content. This enables the recommendation system to make informed decisions based on the minimization objectives and leverage the Bellman equations to optimize the value functions for effective recommendations.

Notice also that $\mathcal{P}$ is a 3-D matrix with dimensions $(k+1)$x$|\mathcal{A}|$x$(k+1)$ and its values are taken by (2) and $\mathcal{R}$ is a 2-D matrix with dimensions $(k+1)$x$(k+1)$ and takes values from (3). In the Python code, the reward function is utilized as a function, and we generate a value from it whenever needed. Notably, we incorporate a significant reward of -1 in two specific cases. The first case pertains to the target content (indicated by the boxed value) to discourage recommending content that the user is already watching while recommending contents with minimum cost and the second case serves in order to avoid considering user random choices of contents that are not in the batch as bad choices. Additionally, we assign a reward of 1 if the user transitions to a terminal state from a recommendation or if a recommended content that is different from what the user already watches is non-cached. It is

crucial to highlight that the transition probabilities within $\mathcal{P}$ always sum up to 1 when considering the probabilities for each possible next state s'. If the probability a is set to a higher value, it signifies that the user has a preference for selecting recommendations rather than searching among the total of k contents arbitrarily.

By incorporating these considerations, we aim to optimize the recommendation system's performance by aligning it with user preferences and maximizing the effectiveness of content recommendations. To show how this is done we'll start by explaining the Bellman Equations and then define some algorithms that solve the problem in the following sections.

2 Markov Decision Processes and Bellman Equations

The figure below illustrates the agent-environment interaction where $S_t \in \mathcal{S}$ is the state, $R_t \in \mathcal{R}$ is the reward and $A_t \in \mathcal{A}(s)$ is the action taken, at time t. Note that at one step later $(t+1)$ agent receives reward R_{t+1} and finds herself at state S_{t+1}:

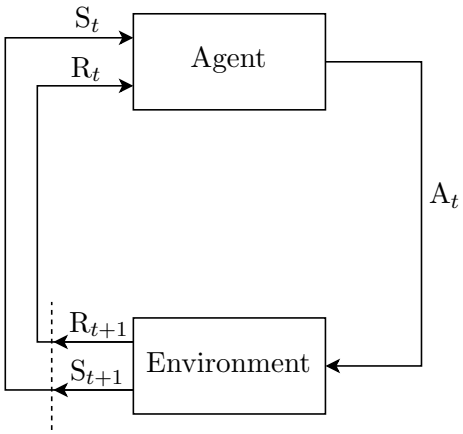


Figure 1: Agent-Environment interaction in a Markov Decision Process (MDP)

In the previous section we explained analytically how the environment is constructed and which is the reward function that the agent uses to interact with the environment. The transition probability matrix $\mathcal{P}$ is part of the environment. At each time step, the agent implements a mapping from states to actions or from states to probabilities of selecting each possible action. In the first case we refer to the definition of a deterministic policy $\pi_t(s)$ and in the second case we refer to a randomized policy $\pi_t(\tilde{a}|s)$, given we are at time step t.

With the intention of demonstrating the Bellman Equations, and keeping in mind that our goal is to find the optimal policy (deterministic or not) we state that the optimal policy is one that maximizes the expected discounted return. In other words the discounted return can be written as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} \cdot R_k \tag{4}$$

where $G_t = R_{t+1} + \gamma \cdot G_{t+1}$ which gives a recursive formula and the goal is to find the optimal policy that maximizes the expected discounted return and $\gamma \in [0, 1]$ is the discount factor. Moreover, it is important to define the state-value function $u_\pi(s)$ that represents the expected cumulative rewards starting from a particular state and following a given policy, and the state-action value function $q_\pi(s)$ that represents the expected cumulative rewards starting from a particular state, taking a specific

action, and then following a given policy as:

$$u_\pi(s) = E_\pi \left[G_t | S_t = s \right] \tag{5}$$

$$q_\pi(s, \tilde{a}) = E_\pi \left[G_t | S_t = s, A_t = \tilde{a} \right] \tag{6}$$

where $\tilde{a}$ is the action that the agent takes and is different from a that is the probability of the user to select one of the recommendations. As long as we find the optimal policy π^*, where $u_*(s) = \max_\pi \{u_\pi(s)\}$ and $q_*(s, \tilde{a}) = \max_\pi \{q_\pi(s, \tilde{a})\}$. These two functions are interconnected and are used in different stages of the policy optimization process. We'll see that the state-action value function is used to guide the policy improvement process by selecting the best actions, while the state value function is used to evaluate the quality of different policies and identify the optimal one.

2.1 Bellman Equations

The Bellman Equations are treated as a universal concept in the world of sequential decision making. They are part of the dynamic programming concept that refers to algorithms used to find optimal policies which have complete knowledge of the environment as an MDP and has complete knowledge of the transition probabilities in $\mathcal{P}$. The Bellman equations are written as:

$$u_\pi(s) = E_\pi \left[G_t | S_t = s \right] = \sum_{\tilde{a}} \pi(\tilde{a}|s) \cdot \underbrace{E_\pi \left[G_t | s, \tilde{a} \right]}_{q_\pi(s, \tilde{a})} \tag{7}$$

$$q_\pi(s, \tilde{a}) = E_\pi \left[G_t | S_t = s, A_t = \tilde{a} \right] = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} P_{ss'}^{\tilde{a}} \cdot \left[r + \gamma u_\pi(s') \right] \tag{8}$$

Equation (7) is pretty straightforward as we only use (5). To prove equation (8) we write:

$$\begin{aligned} q_\pi(s, \tilde{a}) &= E_\pi \left[G_t | S_t = s, A_t = \tilde{a} \right] \\ &= E_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = \tilde{a} \right] \\ &= E_\pi \left[R_{t+1} + \gamma \underbrace{u_\pi(S_{t+1})}_{E_\pi[(7)]=(7)} | S_t = s, A_t = \tilde{a} \right] \\ &= \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} P_{ss'}^{\tilde{a}} \cdot \left[r + \gamma u_\pi(s') \right] \end{aligned}$$

In addition, the Bellman optimality equations are:

$$u_{\pi^*}(s) = \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s, \tilde{a}) \tag{9}$$

$$q_{\pi^*}(s, \tilde{a}) = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} P_{ss'}^{\tilde{a}} \cdot \left[r + \gamma u_{\pi^*}(s') \right] \tag{10}$$

Also, for simplifying the notation, we state that $S_{t+1} = s'$. Furthermore, for (7), (8) $\pi(s) = \tilde{a}$ and for (9), (10) $\pi^*(s) = \tilde{a}$ given that we aim to find a deterministic policy. This means that in (7) $\pi(\tilde{a}|s) = 1$.

3 Model Based Algorithms (Benchmarks)

In this section we are going to show the benchmarks, i.e. the oracle algorithms that we are going to use for this problem. The first one is Value Iteration and the second one is Policy Iteration. These algorithms will give us the minimum possible cost in every user session that is structured by the transition probability matrix $\mathcal{P}$. Also, we state that the task is to solve a minimization problem, thus we use the min operator instead of the max operator considering equations (9) and (10).

3.1 Value Iteration

Value iteration, a fundamental algorithm in reinforcement learning, plays a pivotal role in determining optimal value functions and policies for an MDP. By iteratively updating state values based on the Bellman optimality equation, it aims to minimize the expected cumulative rewards (cumulative costs in our case).

This iterative approach thoroughly explores all available actions $\tilde{a}$, transition probabilities $P^{\tilde{a}}_{ss'}$, and discounted future rewards $R^{\tilde{a}}(s, s')$, converging towards optimal values. Subsequently, the derived policy selects actions that maximize the expected rewards. With its ability to iteratively refine decision-making strategies, value iteration stands as a powerful and indispensable tool in the realm of reinforcement learning. As we can see in Algorithm 1, there is the implemented Value Iteration pseudocode that serves as an oracle when we'll see model free algorithms.

Algorithm 1 // Value Iteration

Require: $\mathcal{S}$: list of states
Require: $\mathcal{A}$: list of actions
Require: γ: discount factor (default: $1 - q$)
Require: ϵ: convergence threshold (default: $1e - 10$)

function VALUE-ITERATION($\mathcal{S}$, $\mathcal{A}$, γ, ϵ)
  $t \leftarrow 0$
  $u \leftarrow$ array of zeros of size $k + 1$
  value_evolution $\leftarrow$ array of zeros of size $k + 1$
  value_evolution $\leftarrow$ stack vertically(value_evolution, u)
  while True do
    $q \leftarrow$ array of zeros of size $(k + 1)$ x $|\mathcal{A}|$
    for s **in** 0 to k **do**
      for $\tilde{a}$ **in** 0 to $|\mathcal{A}| - 1$ **do**
        $q(s, \tilde{a}) = \sum_{s' \in \mathcal{S}} P^{\tilde{a}}_{ss'} \cdot \left[R^{\tilde{a}}(s, s') + \gamma u(s')\right]$
      end for
    end for
    if $\|u - \min_{\tilde{a}}\{q(s, \tilde{a})\}\|_\infty < \epsilon$ **then** ▷ l_∞-norm
      break
    end if
    $u \leftarrow \min\{q(s, \tilde{a})\}$ ▷ Bellman optimality equation
    value_evolution $\leftarrow$ stack vertically(value_evolution, u) ▷ To plot
    $t \mathrel{+}= 1$
  end while
  $\pi(s) = \arg\min_{\tilde{a}}\{q(s, \tilde{a})\}, \quad \forall s \in \mathcal{S}$ ▷ Return the optimal policy (9)
  return u, π, q, value_evolution
end function

3.2 Policy Iteration

Policy iteration, a fundamental algorithm in reinforcement learning, offers a systematic approach to solving MDPs and discovering optimal policies. The process begins with an initial policy, which is iteratively refined through two key steps: policy evaluation and policy improvement.

During policy evaluation, the value function is computed for the current policy by estimating the expected cumulative rewards/costs from each state s, constant action $\tilde{a} = \pi(s)$ and each next state s'. This evaluation step involves solving the Bellman equations iteratively until convergence is achieved. By iteratively updating the value function, policy evaluation provides an accurate estimate of the quality of the current policy.

In the subsequent policy improvement step, the current policy is updated by greedily selecting actions that maximize the expected long-term rewards based on the newly computed value function. This step ensures continual improvement of the policy, as it chooses actions that lead to higher cumulative rewards.

The process of policy evaluation and improvement is iterated until convergence to the optimal policy is attained. Policy iteration guarantees convergence due to the monotonic improvement in policy quality at each iteration.

This iterative approach enables agents to navigate complex environments by dynamically adjusting their decision-making strategies. By iteratively evaluating and refining policies based on the estimated values of states, policy iteration enables the agent to progressively converge towards an optimal policy π^*. As we can see in Algorithm 2, there is the implemented Value Iteration pseudocode that serves as an oracle in our solution.

Algorithm 2 // Policy Iteration

Require: $\mathcal{S}$: list of states
Require: $\mathcal{A}$: list of actions
Require: γ: discount factor (default: $1 - q$)
Require: ϵ: convergence threshold (default: $1e - 10$)

function POLICY-ITERATION($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, γ, ϵ)
  $t \leftarrow 0$
  $\pi(s) \leftarrow$ random policy
  while True do
    $\pi_{\text{old}}(s) = \pi(s)$
    $u =$ **Policy Evaluation**$(\mathcal{S}, \mathcal{A}, \mathcal{P}, \pi, \gamma, \epsilon)$
    $\pi, q =$ **Policy Improvement**$(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \epsilon)$
    $t \mathrel{+}= 1$
    if $\pi_{\text{old}}(s) == \pi(s), \forall s \in \mathcal{S}$ **then**
      break
    end if
  end while
  return u, π, q
end function

Now before showing the algorithm for Policy Evaluation we should state that (7) can also be written as:

$$u_{t+1}(s) = \sum_{\tilde{a} \in \mathcal{A}(s)} \pi(\tilde{a}|s) \sum_{s'} P_{ss'}^{\tilde{a}} \left[R^{\tilde{a}}(s, s') + \gamma u_t(s') \right] \tag{11}$$

Now (11) will be used in the policy evaluation function and we are going to consider that $\tilde{a} = \pi(s)$. In the Policy Evaluation algorithm, we initialize the value function $u_t(s)$ with zeros for all states.

Then, we iterate until the maximum change in the value function is below the convergence threshold ϵ. Inside the loop, we update the value function using the Bellman equation (11). We also track the maximum change in the value function. Once the convergence condition is met, we return the final value function u_t.

Note that we pass the policy π as an input to the Policy Evaluation function to compute the value function based on that policy. Also $\pi(\tilde{a}|s) = 1$ because our extracted policy is deterministic. This means that the policy assigns a probability of 1 to the chosen action $\tilde{a}$ and 0 to all other actions in the rest of the set $\mathcal{A} - \{\tilde{a}\}$.

Algorithm 3 // Policy Evaluation

function POLICY-EVALUATION($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, π, γ, ϵ)
  $t \leftarrow 0$
  $u_t \leftarrow$ array of zeros of size $k+1$
  while True do
    $u_{t+1} \leftarrow$ array of zeros of size $k+1$
    for s **in** 0 to k **do**
      if $s == k$ **then**
        continue ▷ Terminal State
      else
        $u_{t+1}(s) = \sum_{s' \in \mathcal{S}} P_{ss'}^{\pi(s)} \left[R^{\pi(s)}(s, s') + \gamma u_t(s') \right]$
      end if
    end for
    if $\|u_{t+1} - u_t\|_\infty < \epsilon$ **then**
      break
    end if
    $t \mathrel{+}= 1$
  end while
  return u_t
end function

Algorithm 4 // Policy Improvement

function POLICY-IMPROVEMENT($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, γ, ϵ)
  $q \leftarrow$ array of zeros of size $\underbrace{|\mathcal{S}|}_{k+1} \times |\mathcal{A}|$
  for s **in** $\mathcal{S}$ **do**
    if $s \neq |\mathcal{S}| - 1$ **then**
      for $\tilde{a}$ **in** $\mathcal{A}$ **do**
        $q(s, \tilde{a}) = \sum_{s' \in \mathcal{S}} P_{ss'}^{\tilde{a}} \cdot \left[R^{\tilde{a}}(s, s') + \gamma u(s') \right]$ ▷ Estimate state-action value function (8)
      end for
    end if
  end for
  new_$\pi(s) \leftarrow \arg\min_{\tilde{a}} \{q(s, \tilde{a})\}, \quad \forall s \in \mathcal{S}$ ▷ Return the optimal policy (9)
  return new_π, q
end function

Example 3.2.1. *Consider the Network Friendly Recommendations problem. Apply Value Iteration and Policy Iteration algorithms to explore the impact of different parameters on the results. Specifically, vary the values of a and q, and observe the outcomes of the algorithms. Additionally, repeat the experiment for various numbers of contents k.*

Let's say that:

- $|\mathcal{S}| = 10$, $\mathcal{S} = \{0, \dots, k\}$ where if $i \in \mathcal{S}$ and $i = 10$ we define that this is the terminal state.
- $u_{\min} = 0.5$
- $q = 0.2$ and $a = 0.9$ which means that the user is maybe a little addicted because of q and she's not willing to stop the session where in addition a states that she is also willing to click one random recommendation from the batch/action the agent recommends.
- $N = 2$ total items inside a batch/action.
- $C = 0.2k$ cached contents thus the 20% of every row of the cached_matrix (see Figure 2).

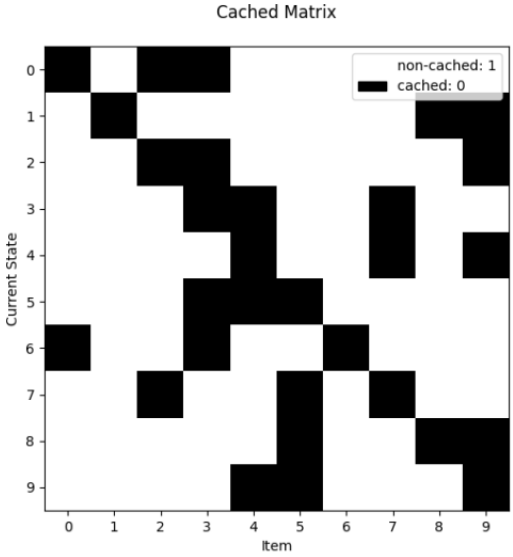
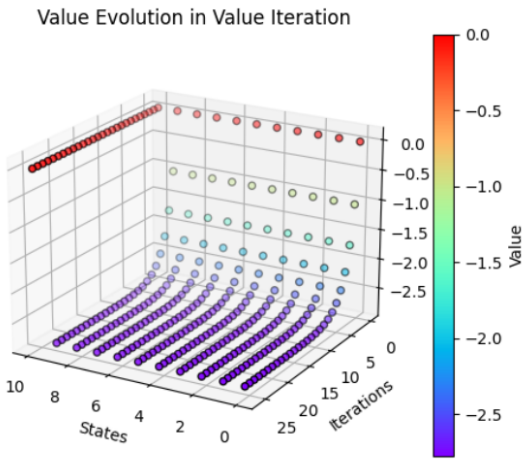


Figure 2: Cached-Matrix.

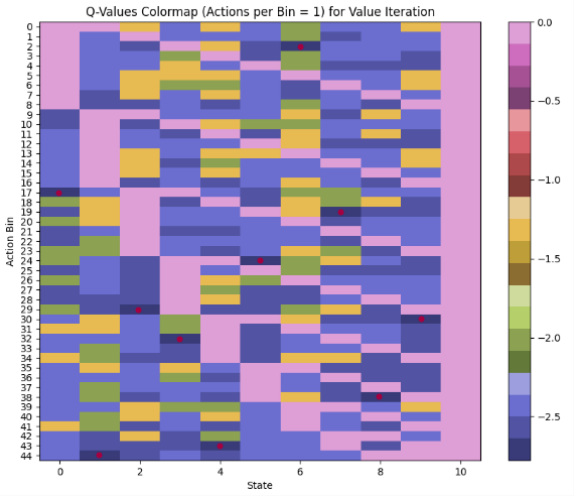
The y-axis of Figure 2 is the current state of the user and the x-axis is the item that she chooses. Then the colormap is the depiction whether the corresponding item is cached.

Afterwards, problem is to learn $u_{\pi^*}(s)$ using the Bellman optimality equation (9) using the preceeding equation (8). Here, in order to see how the algorithm performs, we wish to observe the state-value function's (5) evolution (value_evolution) of the Value Iteration Algorithm 1. This gives the plot in Figure 3a. As we can see, the value function is monotonically decreasing faster at the first four iterations and starts converging to the optimal thereafter. Notice that the sequence $\{u_j(s)\}_{j\in\{0,\dots,\text{total\_iterations-1}\}}$ is the value evolution sequence with $s \in \mathcal{S}$ which gives $k = 10$ total sequences depicted in Figure 3a and if $j = 10$ value evolution is constant and always equal to zero. This happens because we never want to recommend a terminal state to the user.

Observing Figure 3b we set the optimal/minimum $q_{\pi^*}(s, a)$ values as red dots in the plot (•). At the y-axis we have the action id where we have $|\mathcal{A}| = \binom{k}{N} = \binom{10}{2} = 45$ possible actions totally in action space $\mathcal{A}$ for every state $s \in \mathcal{S}$. Observe also the diagonal of this colormap that sets the q-values



(a) Value Evolution $\{u_j(s)\}_{j\in\{0,\ldots,\text{total\_iterations-1}\}}$ during Value Iteration



(b) State-Action value function $q_{\pi*}(s,a)$ after Value iteration

Figure 3: State-value evolution (a) and state-action values (b) applying Value Iteration Algorithm 1

to zero and as the state id increases we have a more sparse representation of the zero q-values. This is because the state-action value function q is always zero if the agent recommends an action/batch that contains the same state/content as the one that the user watches. To see that, the action space is:

$$\mathcal{A} = \left\{ \begin{array}{l} [0,1],[0,2],[0,3],[0,4],[0,5],[0,6],[0,7],[0,8],[0,9], \\ [1,2],[1,3],[1,4],[1,5],[1,6],[1,7],[1,8],[1,9],[2,3], \\ [2,4],[2,5],[2,6],[2,7],[2,8],[2,9],[3,4],[3,5],[3,6], \\ [3,7],[3,8],[3,9],[4,5],[4,6],[4,7],[4,8],[4,9],[5,6], \\ [5,7],[5,8],[5,9],[6,7],[6,8],[6,9],[7,8],[7,9],[8,9] \end{array} \right\}$$

Also we have the action ids that belong to the set $\mathcal{A}_{\text{id}} = \{0,1,2,\ldots,44\}$. Now observe Figure 2 and Figure 3b simultaneously. What we see is that action id 17 corresponds to the batch $[2,3]$ where for state $s=0$, contents 2 and 3 are cached contents. Now for state $s=1$ value iteration recommends action id 44 which corresponds to the batch $[8,9]$ and we can see that both contents 8 and 9 are cached. In general, value iteration algorithm finds and recommends the best combination of contents, thus it gives us the optimal policy. In addition Policy Iteration algorithm gives us the exact same results, shown in Figure 4 :

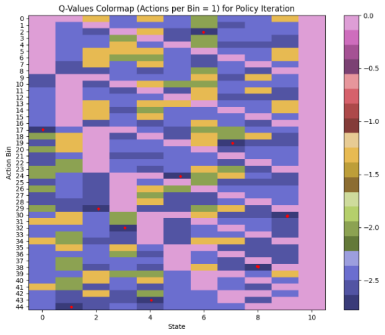


Figure 4: State-Action value function $q_{\pi*}(s,a)$ after applying Policy Iteration Algorithm 2

4 Model Free Algorithms (Alternatives)

In reinforcement learning, model-free learning emerges as a paradigm where agents learn without explicit environment models. Diverging from model-based methods like policy iteration and value iteration, which rely on intricate model construction, model-free approaches like Q-learning and SARSA embody the essence of direct $\underbrace{\text{experiential learning}}_{\text{learning by trial and error}}$ for optimal policy acquisition.

4.1 Q-learning

Q-learning aims to estimate the optimal action-value function, denoted as q_{π^*} (10). However, since Q-learning is a model-free algorithm, we lack access to the transition probabilities of the environment. Consequently, it is well-suited for unknown environments rather than model-based environments like gridworld. Nonetheless, to ensure convergence to the optimal policy π^*, it is crucial to provide the Q-learning algorithm with an adequate number of episodes and an appropriate learning rate, whether adaptive or not. This ensures that our final policy, denoted as π_Q, does indeed converge to the optimal policy. Q-learning (Algorithm 5) is depicted in Figure 5 as a flowchart . The Q-learning update rule

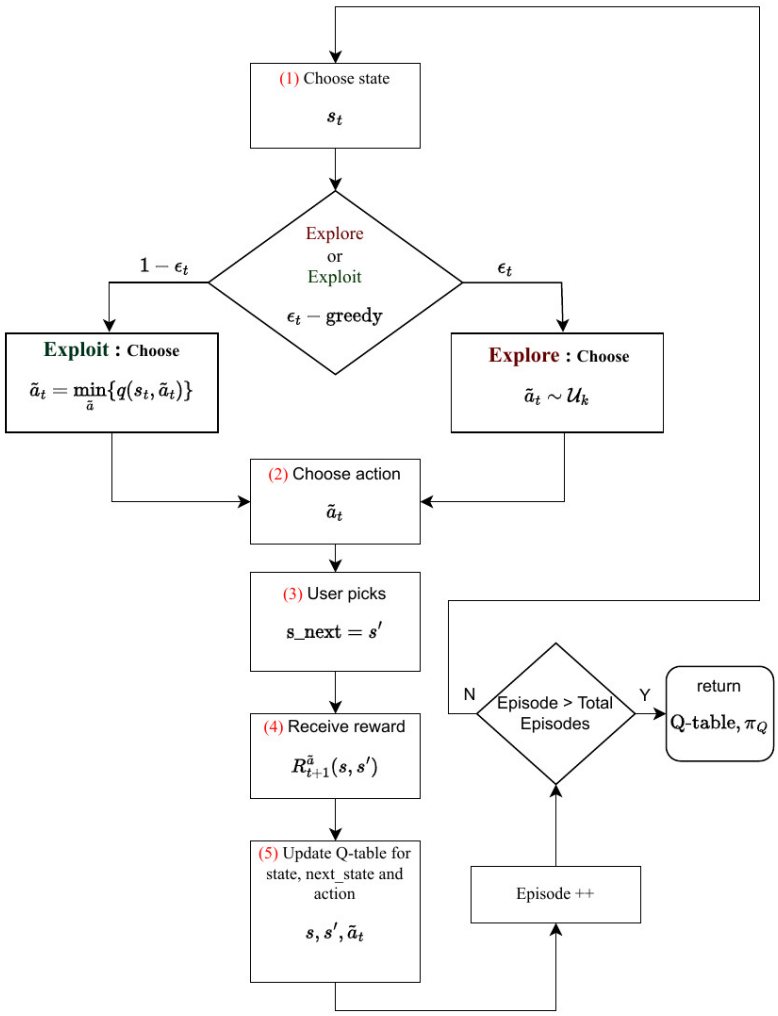


Figure 5: Q-Learning flowchart

allows the agent to improve its estimates of the expected rewards for different actions in different states. It does this by taking into account the observed rewards and the estimated future rewards. By repeatedly updating these estimates over multiple interactions with the environment, the agent learns to associate lower rewards (we have a minimization problem (3)) with better actions, leading to the discovery of an optimal policy π^* given a sufficient amount of episodes. The exploration-exploitation trade-off ensures a balance between trying out new actions and exploiting the learned knowledge. With enough episodes, Q-learning can converge to a reliable estimate of the optimal policy in a Markov Decision Process (MDP) setting.

Algorithm 5 // Q-learning

Require: $\mathcal{S}$: list of states
Require: $\mathcal{A}$: list of actions
Require: γ: discount factor (default: $1 - q$)
Require: num_episodes
Require: learning_rate_schedule = **Learning-rate-schedule**$(\mathcal{S}, \mathcal{A}, \text{num\_episodes}, \text{init\_learning\_rate}, \gamma)$

function Q-LEARNING($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, γ, num_episodes)
  Q $\leftarrow$ array of zeros of size $|\mathcal{S}| \times |\mathcal{A}|$
  num_states $\leftarrow |\mathcal{S}|$
  num_actions $\leftarrow |\mathcal{A}|$
  cost_per_round $\leftarrow$ array of zeros of size num_episodes $\times$ 1
  for episode **in** num_episodes **do**
    $\lambda_{\text{episode}} \leftarrow$ learning_rate_schedule[episode]
    $t\_s \leftarrow$ array of ones of size $|\mathcal{S}| \times 1$
    $s \leftarrow$ randomly choose a state from $\mathcal{S}$ ▷ Init state s
    while $s \neq |\mathcal{S}| - 1$ **do**
      $\epsilon \leftarrow t\_s[s]^{-1/3}$
      $\mathcal{A}(s) \leftarrow$ available actions in current state s
      $\mathcal{A}(s)_{\text{id}} \leftarrow$ filtered indices of available actions that map to $\mathcal{A}$
      $p \sim \mathcal{U}_1$ ▷ Random number between 0 and 1
      if $p < \epsilon$ **then**
        $\tilde{a} \leftarrow$ randomly choose an action from $\mathcal{A}(s)_{\text{id}}$
      else
        $\tilde{a} \leftarrow \arg\min_{\tilde{a}}\{Q(s, \tilde{a})\}$
      end if
      $s' \leftarrow$ determine next state based on $P^{\tilde{a}}_{ss'}$ that defines user behavior ▷ Online Learning
      $Q[s, \tilde{a}] \mathrel{+}= \lambda_{\text{episode}} \cdot (R^{\tilde{a}}(s, s') + \gamma \cdot \min_{\tilde{a}'}\{Q[s', \tilde{a}']\} - Q[s, \tilde{a}])$
      $s \leftarrow s'$
      $t\_s[s] \mathrel{+}= 1$
    end while
    cost_per_round[episode] $\leftarrow$ Expected Cost given $\pi^{\text{temp}}_Q(s) = \arg\min_{\tilde{a}}\{Q(s, \tilde{a})\}, \ \forall s \in \mathcal{S}$
  end for
  $\pi_Q(s) = \arg\min_{\tilde{a}}\{Q(s, \tilde{a})\}, \ \forall s \in \mathcal{S}$
  return Q, π_Q
end function

In Algorithm 5 we see that the update rule is

$$Q_{t+1}(s,\tilde{a}) = (1-\lambda_t)\cdot \underbrace{Q_t(s,\tilde{a})}_{\text{current value}} + \lambda_t \cdot \left(\underbrace{R^{\tilde{a}}(s,s') + \gamma \cdot \underbrace{\min_{\tilde{a}'} Q_t(s',\tilde{a}')}_{\text{optimal future value estimate}}}_{\text{new value (temporal difference target)}} \right) \tag{12}$$

The Q-learning serves as a:

- Moving Average Model: Given a sequence of values $x_1, x_2, \ldots, x_t$, the weighted moving average at time t is calculated as:

$$\text{WMA}_t = \sum_{i=1}^{t} w_i \cdot x_i \tag{13}$$

  where w_i represents the weight assigned to each value x_i. The weights can be any set of real numbers that sum up to 1. As an illustration of (13), when the weights decline exponentially over time, the weighted moving average assigns greater significance to recent values, allowing it to capture short-term patterns or fluctuations. Conversely, if all the weights are identical, the equation simplifies to a straightforward arithmetic average, treating each value equally in the calculation. In (12) we have the previous Q-value estimate $Q_t(s,\tilde{a})$ weighted by $(1-\lambda_t)$ and the new information weighted by λ_t. The new information consists of two components: the immediate reward $R^{\tilde{a}}(s,s')$ and the estimated minimum future reward $\min_{\tilde{a}'} Q_t(s',\tilde{a}')$. These components are combined using multiplication and addition to update the Q-value estimate.

- Model Free Learning method: We estimate the expected rewards directly from samples, given that sample $= R^{\tilde{a}}(s,s') + \gamma \cdot \min_{\tilde{a}'} Q_t(s',\tilde{a}')$. Note here that in Algorithm 5 we have the same update as in equation (12) $\iff$ $Q_{t+1}(s,\tilde{a}) = Q_t(s,\tilde{a}) + \lambda_t \cdot \underbrace{\left(R^{\tilde{a}}(s,s') + \gamma \cdot \min_{\tilde{a}'} Q_t(s',\tilde{a}') - Q_t(s,\tilde{a}) \right)}_{\text{TD error}}$.

  The expression $R^{\tilde{a}}(s,s') + \gamma \cdot \min_{\tilde{a}'} Q_t(s',\tilde{a}')$ represents the sum of the immediate reward $R^{\tilde{a}}(s,s')$ obtained when transitioning from state s to a new state s' after taking action $\tilde{a}$, and the discounted future reward estimate. The expression $Q_t(s,\tilde{a})$ represents the current Q-value estimate for the state-action pair $(s,\tilde{a})$ at time step t. It denotes the expected cumulative reward for taking action $\tilde{a}$ in state s based on the current Q-value estimates. The TD error measures the difference between the current estimate $Q_t(s,\tilde{a})$ and the updated estimate $Q_{t+1}(s,\tilde{a})$. It represents the discrepancy between the observed immediate reward and the estimated future rewards. Notice that the difference between the updated estimate $Q_{t+1}(s,\tilde{a})$ and the current estimate $Q_t(s,\tilde{a})$ is equal to the TD error, which is $\lambda_t \cdot R^{\tilde{a}}(s,s') + \lambda_t \cdot \gamma \cdot \min_{\tilde{a}'} Q_t(s',\tilde{a}')$. This error determines the amount by which the current estimate should be adjusted. The Q-learning update rule combines the TD error with the current estimate, weighted by the learning rate λ_t, to obtain the updated estimate. This rule iteratively refines the Q-value estimates based on observed rewards and estimated future rewards, facilitating the learning of an optimal policy.

The learning rate, denoted as λ_t, determines the balance between the previous estimate and the new information. It influences the weight given to the new observation relative to the previous estimate. By iteratively updating the Q-values using this rule, the agent gradually refines its estimates and

learns to associate higher Q-values with more rewarding state-action pairs. This process allows the agent to discover an optimal policy that maximizes cumulative rewards over time. Here a learning rate schedule is created to fine-tune the learning rate λ_t (see Algorithm 6).

Algorithm 6 // Learning Rate Schedule

Require: $\mathcal{S}$: list of states
Require: $\mathcal{A}$: list of actions
Require: num_episodes: number of episodes
Require: initial_learning_rate: initial learning rate
Require: discount_factor: discount factor

function LEARNING-RATE-SCHEDULE($\mathcal{S}$, $\mathcal{A}$, num_episodes, init_learning_rate, γ)
    learning_rate_schedule ← array of ones of size num_episodes × init_learning_rate
    learning_rate ← init_learning_rate
    performance_improvements ← array of zeros of size num_episodes
    threshold ← 0.1 ▷ Initial threshold value
    increase_factor ← 1.1 ▷ Initial increase factor
    decrease_factor ← 0.9 ▷ Initial decrease factor
    performance_improvement ← 0.0
    for episode **in** 0 **to** num_episodes − 1 **do**
        Q ← **Q_learning**($\mathcal{S}, \mathcal{A}$, 1, learning_rate_schedule[episode], discount_factor)
        if episode > 0 **then**
            prev_Q ← **Q_learning**($\mathcal{S}, \mathcal{A}$, 1, learning_rate_schedule[episode − 1], discount_factor)
            performance_improvement ← mean(|Q − prev_Q|)
            performance_improvements ← append(performance_improvements, performance_improvement)
            if performance_improvement > threshold **then**
                learning_rate ← learning_rate · increase_factor
            else
                learning_rate ← learning_rate · decrease_factor
            end if
            threshold ← mean(performance_improvements[−5 :]) ▷ 5 previous improvements
        end if
        learning_rate_schedule ← append(learning_rate_schedule, learning_rate)
    end for
    return learning_rate_schedule
end function

The learning rate schedule function (Algorithm 6) implements a schedule to adaptively adjust the learning rate in Q-learning based on observed performance improvements. A small learning rate leads to cautious and stable updates, avoiding overestimation of Q-values. Conversely, a large learning rate enables faster learning but may result in instability. Gradually decreasing the learning rate strikes a balance between exploration and exploitation, allowing the agent to refine Q-values in a controlled manner and converge towards optimal values, improving performance and reliability. A small learning rate is suitable for extended episodes in Q-learning, providing stability and cautious updates. Conversely, a higher learning rate is riskier but may yield better results for shorter episodes, enabling faster learning and adaptation. The choice of learning rate depends on the desired trade-off between stability and speed of convergence in the learning process.

For the Example 3.2.1 we get the following results considering Q-learning (Algorithm 5 and the learning rate schedule in Algorithm 6) where we set the optimal/minimum $Q_{\pi_Q}(s, a)$ values as red dots (•) in Figure 6. Here the policy of Q-learning algorithm is exactly the same as the policy from the Value Iteration algorithm and Policy iteration algorithm. The expected cost we get in every episode of the Q learning algorithm is depicted in Figure 7. By setting $a = 0.6$ and $q = 0.3$ while $u\_\min = 0.6$ we make our problem even harder but Q learning still finds the optimal policy. For this case, as we see in Figure 8, it takes ten thousand more episodes to converge to a minimum cost. In general cost$= -1$ means that only one content is cached and cost$= -2$ means that all contents for all possible recommendation batches of size $N = 2$ are cached contents.

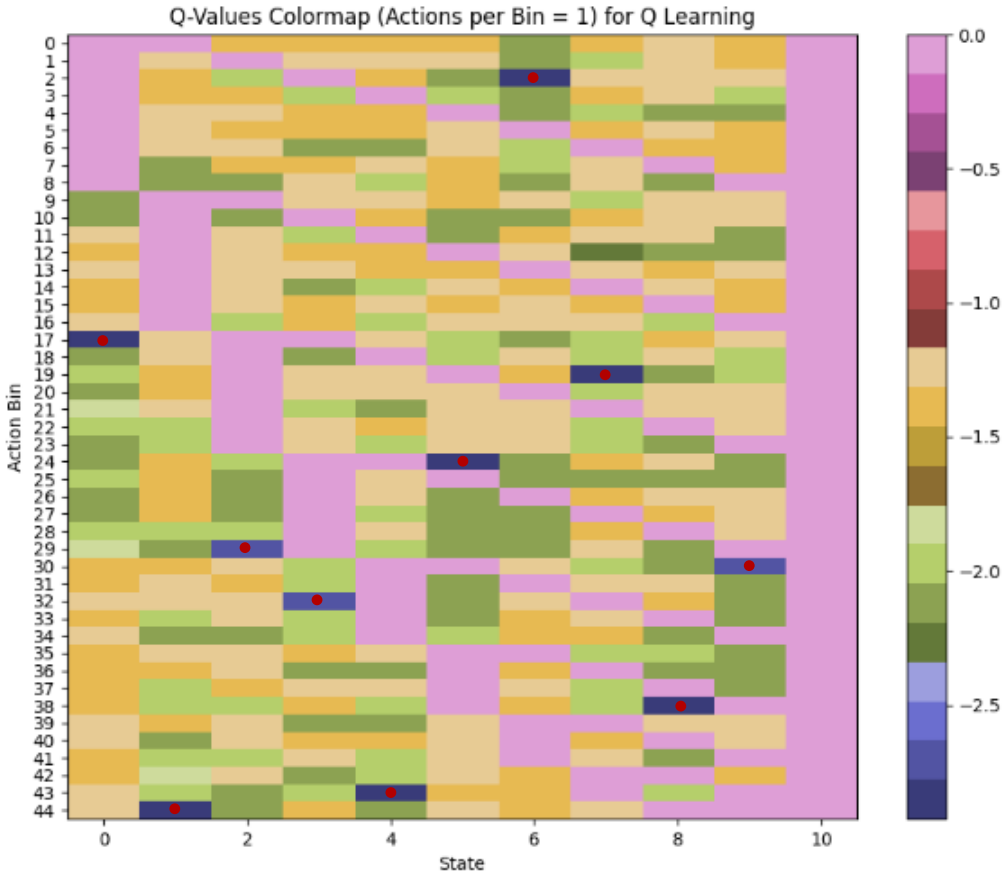


Figure 6: State-Action value estimate Q_{π_Q} after Q-learning

Based on Figure 9 we can say that Q-learning "breaks" if we decrease probability a sufficiently and/or if we increase relevance measure $u\_\min$ and/or probability q a lot. This is one way to "break" Q-learning given that the amount of episodes is not enough. Another way to do it is by setting a high value of contents k. If that happens we can "break" Q-learning again because the action space expands exponentially with the number of available choices being k (the action space grows combinatorially with respect to k), as determined by the combination formula $\binom{k-1}{N}$ meaning that for a great value of k we might need billions of episodes that would take a lifetime to run on a simple CPU in order to have a solution to the network friendly recommendations problem.

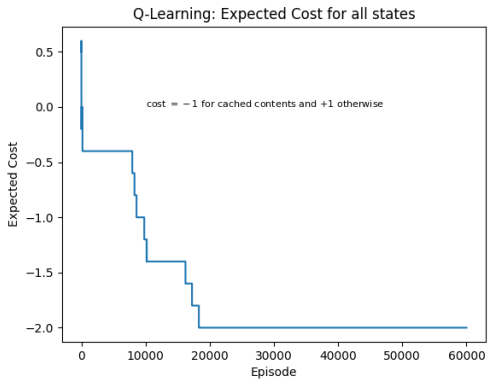


Figure 7: Expected cost for recommended contents (cached or not) after each episode considering $a = 0.9$, $u\_\min = 0.5$ and $q = 0.2$.

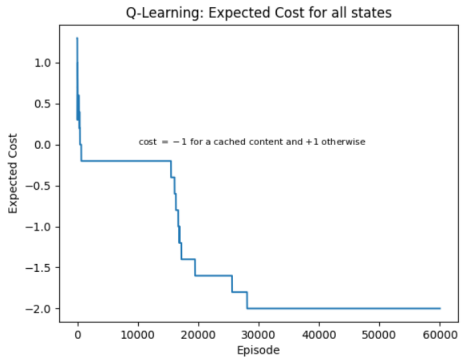


Figure 8: Expected cost for recommended contents (cached or not) after each episode considering $a = 0.6$, $u\_\min = 0.6$ and $q = 0.3$.

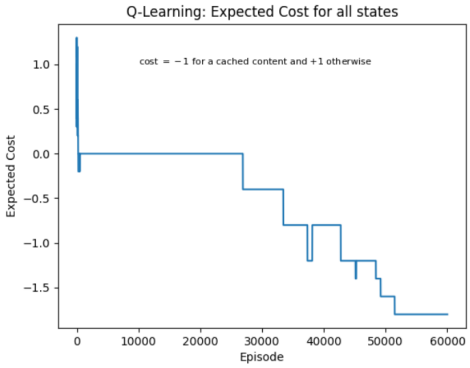


Figure 9: Expected cost for recommended contents (cached or not) after each episode considering $a = 0.2$, $u\_\min = 0.6$ and $q = 0.6$.