Markov Decision Processes: Value Iteration, Policy Iteration and Linear Programming methods

done by

Jean-Claude S. MITCHOZOUNOU

Africa Business School, UM6P, Morocco

Master of Quantitative and Financial Modelling (QFM)

Project of second semester

Supervisor: Prof. Omar Saadi

Academic Year 2023-2024

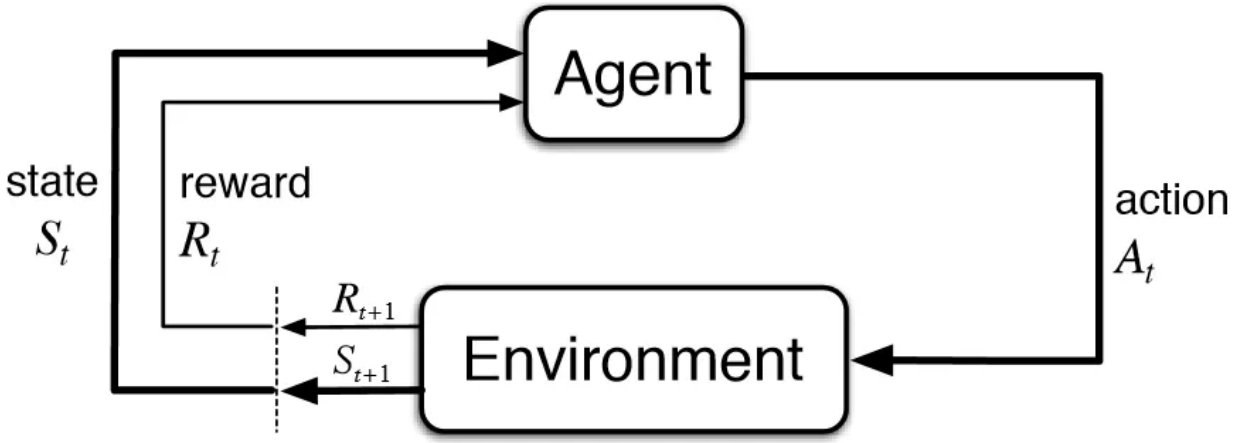
CONTENTS

Reinforcement Learning and Markov Decisions processes

1.1 Overviews on Markov decisions processes

Reinforcement Learning (RL) and Markov Decision Processes (MDPs) are closely related concepts, and MDPs provide a formal mathematical framework that underpins much of RL theory and practice.

Reinforcement Learning is the science of decision-making. It is about learning the optimal behavior in an environment to obtain maximum reward. MDP gives us a way to formalize sequential decision-making. In other way, a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a Markov decision process, or MDP, and consists of a set of states (with an initial state s_0); a set $ACTIONS(s)$ of actions in each state; a transition model $P(s' \mid s, a)$; and a reward function $R(s)$. This formalization is the basis for structuring problems that are solved with reinforcement learning.



In an MDP, we have a decision maker, called an agent, that interacts with the environment it's placed in as it is represented in the figure above. These interactions occur sequentially over time. At each time step, the agent will get some representation of the environment's state. Given this representation, the agent selects an action to take. The environment is then transitioned into a new state, and the agent is given a reward as a consequence of the previous action. Lets now discuss about the components of RL used in MDP:

1. **Agents :** Agents are a central concept in the field of RL. An RL agent is an autonomous entity or computational system that interacts with an environment to learn and make decisions in order to achieve specific goals or maximize cumulative rewards. These agents

can be implemented in various domains, from robotics to game playing to recommendation systems. For instance, an agent can be a player in the game whose motive is to win the game or gain maximum rewards.

2. **Environment:** refers to what the world looks like and the rules of the world. It is a well-defined and structured system that an RL agent interacts with over time. It represents the world or domain in which the agent's actions have consequences, and it provides feedback to the agent based on those actions.

3. **States (S):** refers to the state of our agent, not the state of our environment.The environment has a set of possible states that represent different situations, configurations, or conditions it can be in.In MDP we use s to denote state and in our example we call them $s = 0, 1, 2, \ldots$.

4. **Action (A):** refers to the decisions that can be taken by RL agent in each state. The set of possible actions is not necessarily the same for each state.

5. **Rewards (R):** After each action, the environment provides the RL agent with a numerical signal called a reward. The reward indicates the immediate desirability or quality of the action taken. The agent's goal is often to maximize the cumulative reward over time.

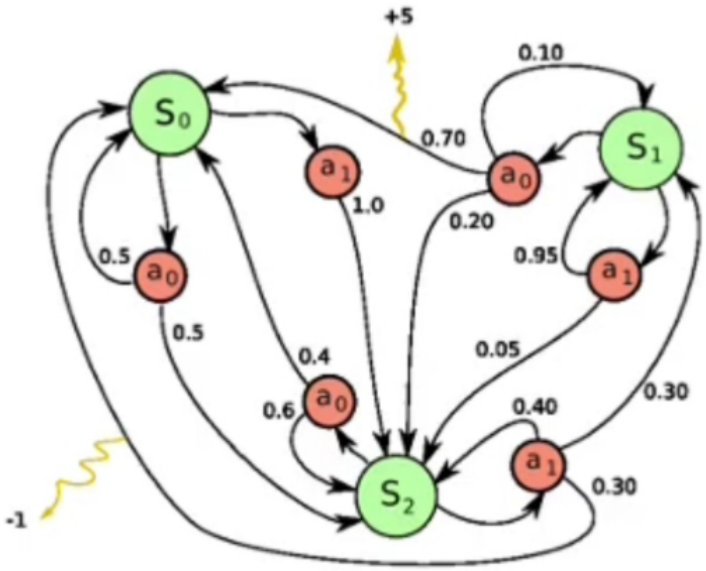
6. **Trajectory:** The process of selecting an action from a given state, transitioning to a new state, and receiving a reward happens sequentially over and over again, and creates something called a trajectory that shows the sequence of states, actions, and rewards

7. **Policy:** is a decision-making strategy in which the agent chooses actions adaptively based on the history of observations;

1.2 Mathematicals concepts in MDPs

1.2.1 Defining MDPs

In reinforcement learning, the interactions between the agent and the environment are often described by an infinite-horizon, discounted Markov Decision Process (MDP) $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma, \mu)$, specified by:



- A state space $\mathcal{S}$, which may be finite or infinite. For mathematical convenience, we will assume that $\mathcal{S}$ is finite or countably infinite.
- An action space $\mathcal{A}$, which also may be discrete or infinite. For mathematical convenience, we will assume that $\mathcal{A}$ is finite.
- A transition function $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$, where $\Delta(\mathcal{S})$ is the space of probability distributions over $\mathcal{S}$ (i.e., the probability simplex). $P(s' \mid s, a)$ is the probability of transitioning into state s' upon taking action a in state s. We use $P_{s,a}$ to denote the vector $P(\cdot \mid s, a)$.
- A reward function $r : \mathcal{S} \times \mathcal{A} \to [0, 1]$. $r(s, a)$ is the immediate reward associated with taking action a in state s. More generally, $r(s, a)$ could be a random variable (where the distribution depends on s, a).
- A discount factor $\gamma \in [0, 1)$, which defines a horizon for the problem.
- An initial state distribution $\mu \in \Delta(\mathcal{S})$, which specifies how the initial state s_0 is generated. In many cases, we will assume that the initial state is fixed at s_0, i.e., μ is a distribution supported only s_0.

1.2.2 Policies and Discounted Total Reward

In a given MDP $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma, \mu)$, the agent interacts with the environment according to the following protocol:
The agent starts at some state $s_0 \sim \mu$; at each time step $t = 0, 1, 2, \ldots$, the agent takes an action $a_t \in \mathcal{A}$, obtains the immediate reward $r_t = r(s_t, a_t)$, and observes the next state s_{t+1} sampled according to $s_{t+1} \sim P(\cdot \mid s_t, a_t)$.
The interaction record at time t is: $\tau_t = (s_0, a_0, r_0; s_1, a_1, r_1, \ldots, s_t, a_t.r_t)$ is called a trajectory, which includes the observed state at time t.

A policy is a (possibly randomized) mapping from a trajectory to an action, i.e. $\pi : H \to \Delta(\mathcal{A})$ where H is the set of all possible trajectories (of all lengths) and $\Delta(\mathcal{A})$ is the space of probability distributions over $\mathcal{A}$. A stationary policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ specifies a decision-making strategy in which the agent chooses actions based only on the current state, i.e. $a_t \sim \pi(\cdot \mid s_t)$. A deterministic, stationary policy is of the form $\pi : \mathcal{S} \to \mathcal{A}$.

For a given: policy π
initial state s
SARSA sequence: $s, \pi(s), r(s, \pi(s)), s \to s', \pi(s'), r(s', \pi(s')), \ldots, \infty$
State sequence: $s, s', s'', \ldots, \infty$
Reward sequence: $r(s, \pi(s)), r(s', \pi(s')), r(s'', \pi(s'')), \ldots, \infty$

Discounted total reward:

$$\begin{aligned} R(s, \pi) &= r(s, \pi(s)) + \gamma r(s', \pi(s')) + \gamma^2 r(s'', \pi(s'')) + \ldots, \infty \\ &= \text{immediate reward} + \text{discounted future rewards} \end{aligned}$$

1.2.3 Objective Of MDPs:

The objective in an MDP is typically to find a policy π^* that maximizes the expected cumulative reward over time. This is often expressed as the expected return

$$\mathrm{E}(R(s, \pi)) = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$$

1.2.4 Markov Property

Andrey Andreyevich Markov (1856-1922)
Russian mathematician known for his work on stochastic processes

The Markov Property: given the present, the future and the past are independent

- i.e., Everything you need to know about the past is included in the present state

For MDPs, the Markov property means that:

$$P(S_{t+1} = s' \mid S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0) = P(S_{t+1} = s' \mid S_t = s_t, A_t = a_t)$$

Independent of past states and actions

1.2.5 State Value Function of Policies

For more information on State Value Function, please see ([1, 5, 13].)
Given MDP (S, A, P, R, γ):

Value of a state s under policy π:

$V^\pi(s)$ = expected utility starting in s and acting according to π

$$V^\pi(s) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid \pi, s_0 = s\right)$$

Sequence of rewards generated by following π

Optimal value of a state s:

$V^*(s)$ = expected utility starting in s and acting optimally

$$V^*(s) = V^{\pi^*}(s) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid \pi, s_0 = s\right)$$

Rewards generated by following π^*(optimal policy.)

1.2.6 Action Value Functions of Policies

For more information on action value Function, please see ([1, 5, 13]).
It is also helpful to define action-value functions.

Q-value of taking action a in the state s then following policy π:

$Q^*(s,a)$ = expected utility taking action a in the state s and then following policy π

$$\boxed{Q^\pi(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t,a_t) \mid \pi, s_0 = s, s_0 = a\right)}$$

• The optimal Q-value of taking action a in the state s is the Q-value optaine by following the policy associated to V^*
• π^* can be greedily determined from Q^* : $\boxed{\pi^*(s) = \underset{a\in A}{argmax} Q^*(s,a)}$

Remark 1.1. Since $r(s,a)$is bounded between 0 and 1, we can easily show that $0 \leq V^\pi(s) \leq \dfrac{1}{1-\gamma}$ and $0 \leq Q^\pi(s,a) \leq \dfrac{1}{1-\gamma}$.

1.2.7 Bellman Equations

The information and analyses presented in this section are based on previous work and research available in the literature, see([1, 13]).

Bellman's equations represent important results in MDPs. These results will be of great use throughout our paper.In particular, these equations will help us to implement the various methods for finding the optimal policy.

Lemma 1.1. *Suppose that π is a stationary policy. Then V^π and Q^π satisfy the following equations: for all $s \in S$, $a \in A$,*

$$V^\pi(s) = Q^\pi(s,\pi(s)).$$
$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s'\sim P(\cdot|s,a)}[V^\pi(s')].$$

Bellman consistency equations for stationnary policies

$$V^\pi(s) = Q^\pi(s,\pi(s)) \quad (i).$$
$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s'\sim P(\cdot|s,a)}[V^\pi(s')] \quad (ii).$$

Proof. (i)- By definition one has:

$$Q^\pi(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t,a_t) \mid \pi, s_0 = s, a_0 = a\right)$$

. So for $a = \pi(s)$ one obtains:

$$\begin{aligned} Q^\pi(s,\pi(s)) &= \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t,a_t) \mid \pi, s_0 = s, a_0 = \pi(s)\right) \\ &= \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r(s_t,a_t) \mid \pi, s_0 = s\right) \text{ because } \pi \text{ is stationnary and we're starting at } s. \\ &= V^\pi(s) \end{aligned}$$

(ii)-

$$\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}\left(\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t) \mid \pi, s_0=s, a_0=a\right)\\
&= \mathbb{E}\left(r(s_0,s_0)+\sum_{t=1}^{\infty}\gamma^t r(s_t,a_t) \mid \pi, s_0=s, a_0=a\right)\\
&= r(s,a)+\mathbb{E}\left(\sum_{t=1}^{\infty}\gamma^t r(s_t,a_t) \mid \pi, s_0=s, a_0=a\right)\\
&= r(s,a)+\gamma\mathbb{E}\left(\sum_{t=1}^{\infty}\gamma^{t-1} r(s_t,a_t) \mid \pi, s_0=s, a_0=a\right)\\
&= r(s,a)+\gamma\mathbb{E}\left(\sum_{t=1}^{\infty}\gamma^{t} r(s_{t+1},a_{t+1}) \mid \pi, s_0=s, a_0=a\right)\\
&= r(s,a)+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[\mathbb{E}\left(\sum_{t=1}^{\infty}\gamma^{t-1} r(s_{t+1},a_{t+1}) \mid \pi, s_1=s'\right)\right]\\
&\text{by using total expected formula:}\mathbb{E}(X)=\mathbb{E}(\mathbb{E}(X|Y))\\
&= r(s,a)+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[\mathbb{E}\left(\sum_{t=1}^{\infty}\gamma^{t-1} r(s_{t},a_{t}) \mid \pi, s_0=s'\right)\right]\\
&\text{by using homogeneity of Markov chain}\\
&= r(s,a)+\gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[V^\pi(s')\right]
\end{aligned}$$

Hence the results.

Remark 1.2. In matrix form, Q^π can be written as:

$$Q^\pi = r + \gamma P V^\pi$$

If we define P^π to be the transition matrix on state-action pairs induced by a stationary policy π, specifically:

$$P^\pi_{(s,a),(s',a')} := P(s'|s,a)\pi(a'|s').$$

In particular, for deterministic policies π defined by:

$$\pi(a'/s') := \begin{cases} 1 \text{ if } \pi(s') = a' \\ 0 \text{ if not} \end{cases};$$

we have:

$$P^\pi_{(s,a),(s',a')} := \begin{cases} P(s'|s,a) & \text{if } a' = \pi(s') \\ 0 & \text{if } a' \neq \pi(s') \end{cases}$$

With this notation, it is straightforward to verify:

$$Q^\pi = r + \gamma P V^\pi$$

$$Q^\pi = r + \gamma P^\pi Q^\pi.$$

Indeed, $\forall (s,a),(s{\cdot}a') \in S \times A$

$$\begin{aligned}
Q^\pi(s,a) &= r(s,a) + \gamma\mathbb{E}_{s'\sim P(\cdot|s,a)}[V^\pi(s')]\\
&= r(s,a) + \gamma\sum_{s'} P(s'|s,a)V^\pi(s')\\
&= r(s,a) + \gamma(PV^\pi)(s,a)\\
&\iff Q^\pi = r + \gamma P V^\pi
\end{aligned}$$

and

$$\begin{aligned}(P^\pi Q^\pi)(s,a) &= \sum_{s',a'} P^\pi_{(s,a),(s',a')} Q^\pi(s',a') \\ &= \sum_{s',a'} P(s'|s,a)\pi(a'|s')Q^\pi(s',a') \\ &= \sum_{s'} P(s'|s,a)\pi(\pi(s')|s')Q^\pi(s',\pi(s')) \\ &= \sum_{s'} P(s'|s,a)V^\pi(s') \\ &= (PV^\pi)(s,a)\end{aligned}$$

i.e $PV^\pi = P^\pi Q^\pi \iff r + \gamma P^\pi Q^\pi = r + \gamma P V^\pi = Q^\pi$

Corollary 1.1. *Suppose that π is a stationary policy. We have that:*

$$Q^\pi = (I - \gamma P^\pi)^{-1} r$$

where I is the identity matrix.

Proof. Since P is Markovian, one has $||Px||_\infty \leq ||x||_\infty \, \forall x$. And further $\gamma \in [0,1)$. Consequently, $(I - \gamma P^\pi)$ is invertible.

Theorem 1.1. *Let Π be the set of all non-stationary and randomized policies. Define:*

$$V^*(s) := \sup_{\pi \in \Pi} V^\pi(s)$$

$$Q^*(s,a) := \sup_{\pi \in \Pi} Q^\pi(s,a).$$

which is finite since $V^\pi(s)$ and $Q^\pi(s,a)$ are bounded between 0 and $\frac{1}{1-\gamma}$.
There exists a stationary and deterministic policy π^+ such that for all $s \in S$ and $a \in A$,

$$V^{\pi^*}(s) = V^*(s)$$

$$Q^{\pi^*}(s,a) = Q^*(s,a).$$

We refer to such a π^+ as an optimal policy.

Proof. Indication:

Theorem 1.2. *(Bellman optimality equations) We say that a vector $Q \in \mathbb{R}^{|S||A|}$ satisfies the Bellman optimality equations if:*

$$Q(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\max_{a' \in A} Q(s',a') \right].$$

For any $Q \in \mathbb{R}^{|S||A|}$, Q is said to be optimal($Q = Q^*$) if and only if Q satisfies the Bellman optimality equations.

To proof this theorem we will use the following lemma.

Lemma 1.2. $V^*(s) = \max_{a \in A} Q^*(s,a)$

Definition 1.2.1. Let π_Q denote the greedy policy with respect to a vector $Q \in \mathbb{R}^{|S||A|}$, i.e.,

$$\pi_Q(s) := \arg\max_{a \in A} Q(s,a).$$

With this notation, by the above theorem, the optimal policy π^* is given by:

$$\pi^* = \pi_{Q^*}.$$

Let us also use the following notation to turn a vector $Q \in \mathbb{R}^{|S||A|}$ into a vector of length $|S|$:

$$V_Q(s) := \max_{a \in A} Q(s,a).$$

Definition 1.2.2. The Bellman optimality operator $T_M : \mathbb{R}^{|S||A|} \to \mathbb{R}^{|S||A|}$ is defined as:

$$TQ := r + \gamma P V_Q.$$

This allows us to rewrite the Bellman optimality equation in the concise form:

$$Q = TQ,$$

and, so, the previous theorem states that $Q = Q^*$ if and only if Q is a fixed point of the operator T.

Remark 1.3. The equation $Q = TQ$ has solution because the operator T is γ-Lipchitz with $\gamma \in [0, 1)$.Indeed, for any two vectors $Q, Q' \in \mathbb{R}^{|S||A|}$,

$$\|TQ - TQ'\|_\infty \leq \gamma \|Q - Q'\|_\infty$$

Proof. First, let us show that for all $s \in S$, $|V_Q(s) - V_{Q'}(s)| \leq \max_{a \in A} |Q(s,a) - Q'(s,a)|$. Assume $V_Q(s) > V_{Q'}(s)$ (the other direction is symmetric), and let a be the greedy action for Q at s. Then

$$\begin{aligned} |V_Q(s) - V_{Q'}(s)| &= Q(s,a) - \max_{a' \in A} Q'(s,a') \\ &\leq Q(s,a) - Q'(s,a) \\ &\leq \max_{a \in A} |Q(s,a) - Q'(s,a)|. \end{aligned}$$

Using this,

$$\begin{aligned} \|TQ - TQ'\|_\infty &= \gamma \|PV_Q - PV_{Q'}\|_\infty \\ &= \gamma \|P(V_Q - V_{Q'})\|_\infty \\ &\leq \gamma \|V_Q - V_{Q'}\|_\infty \\ &= \gamma \max_s |V_Q(s) - V_{Q'}(s)| \\ &\leq \gamma \max_s \max_a |Q(s,a) - Q'(s,a)| \\ &= \gamma \|Q - Q'\|_\infty \end{aligned}$$

Numerical study of solving methods of MDPs

To solve an MPDs, we will use severals methods such as **Value Iteration, Policyiteration and Linear Programming algorithms**. Here is their theorical complexity.

Algorithms	Poly?	Strongly Poly?
Value Iteration	$\frac{\lvert S\rvert^2\lvert A\rvert}{1-\gamma} L(P,r,\gamma)\log\frac{1}{1-\gamma}$	$\times$
Policy Iteration	$\frac{\lvert S\rvert^3+\lvert S\rvert^2\lvert A\rvert}{1-\gamma} L(P,r,\gamma)\log\frac{1}{1-\gamma}$	$(\lvert S\rvert^3+\lvert S\rvert^2\lvert A\rvert)\cdot\min\left(\frac{\lvert A\rvert^{\lvert S\rvert}}{\lvert S\rvert}, \frac{\lvert S\rvert^2\lvert A\rvert}{1-\gamma}\log\frac{\lvert S\rvert^2}{1-\gamma}\right)$
LP-Algorithms	$\lvert S\rvert^3\lvert A\rvert L(P,r,\gamma)$	$\lvert S\rvert^4\lvert A\rvert^4\log\frac{\lvert S\rvert}{1-\gamma}$

Table 2.1: Computational complexities of various approaches

2.1 Value Iteration

Value iteration algorithm is very well detailed in the various studies and research available in the literature as indicated in the following references (see [1, 2, 3, 7, 8, 9, 6]).

Complexity: For a fixed value of γ, Value iteration algorithm is polynomial time algorithm. Here $|S|^2|A|$(see Tab 2.1) is the assumed runtime per iteration of value iteration

Value iteration updates the value function V using the Bellman optimality equation (Theorem 1.2). It is an algorithm used to compute the optimal policy and the value function of an MDP. It iteratively updates the value of each state until the values converge to the optimal values.

Starting at some $Q(Q = 0,\ \text{generally})$, we iteratively apply $T(\textit{Bellman operator})$:

$$Q \leftarrow TQ,$$

untill convergence

Pseudo-code: Value iteratoin

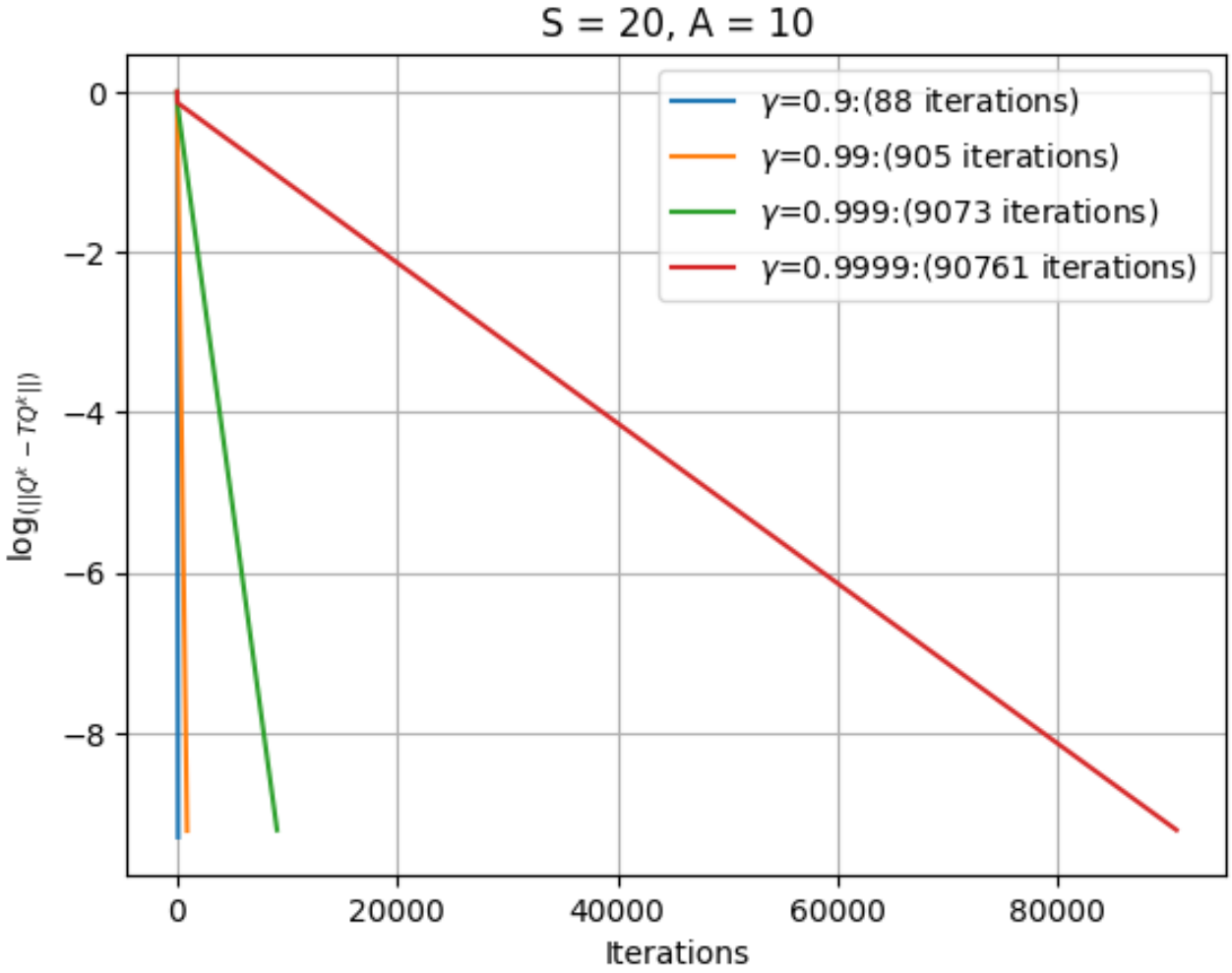
input: transition probabilities P, reward function r, Q discount factor γ and θ,
output: optimal policy π^*, value function V
initialize: $Q(s,a) = 0, \forall (s,a) \in S \times A$
repeat
    $\Delta \leftarrow 0$
    TQ=$BellmanOperator(Q, R, P, \gamma)$
    $\delta = \max(|(Q - TQ)|)$
    $Q = TQ$
    if $\Delta \leq \theta$ **then**
        break

end if
until the policy is optimal and the function converges
for $s \in S$ **do**
    $V^*(s) = \max_{a \in A} Q(s, a)$
    $\pi^*(s) = \arg\max_{a \in A} Q(s, a)$
end for
return V^*, π^*

At least, we have the optimal policy π^* and optimal value function V^*.

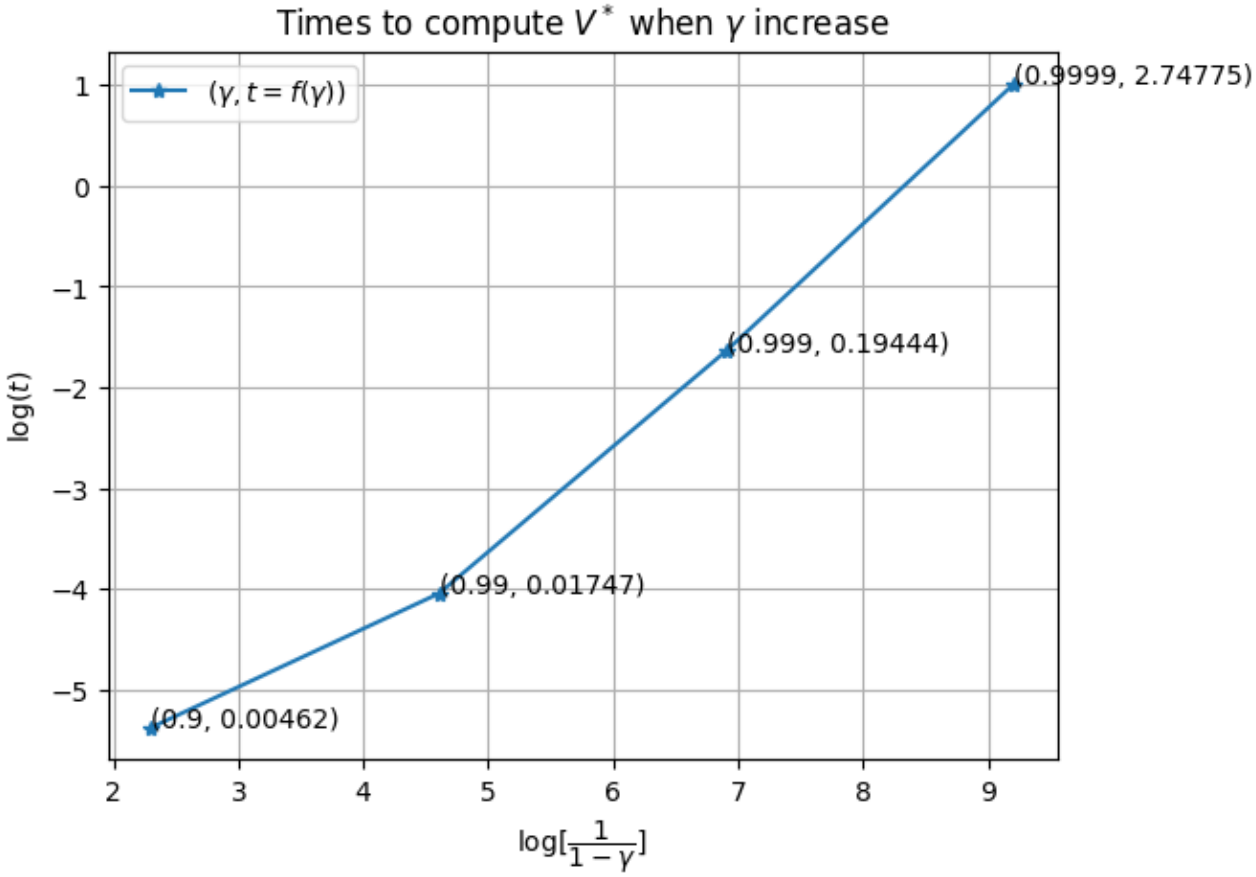
2.1.1 Numerical results and interpretation

The effects of γ on convergence.



1. γ **close to 0**:
    - The Value Iteration algorithm converges quickly because it does not look far into the future.
    - When γ is close to 0, immediate rewards are much more important than future rewards.
2. γ **close to 1**:
    - We can see that as we get closer to 1, the number of iterations is multiplied by 10. This means that the algorithm becomes 10 times slower in terms of convergence as we get closer to 1.

- The solutions found may be more optimal in the long term, but the convergence time can be significantly longer as you can see on the following figure which plots the evolution of the solution's calculation time as a function of the evolution of gamma

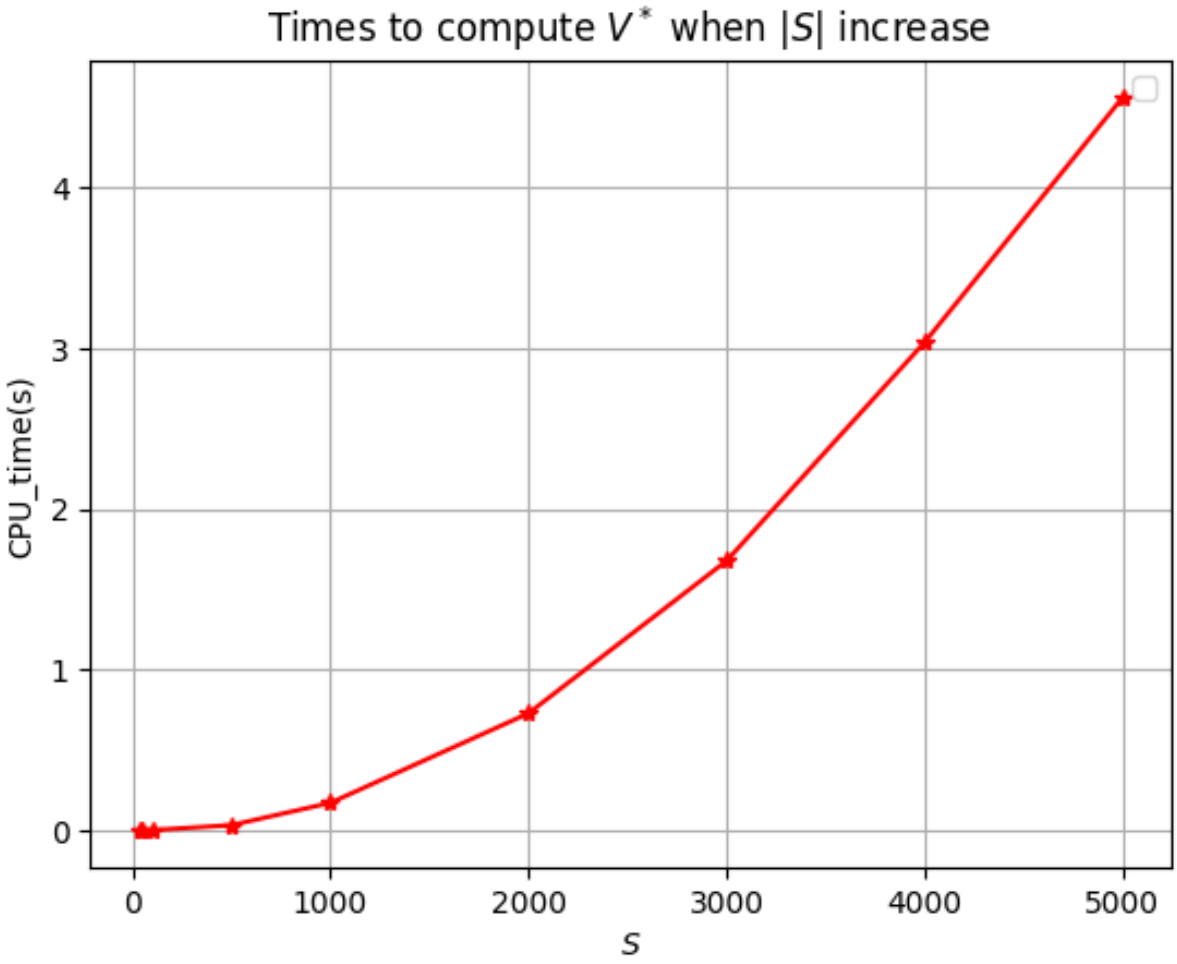


Note that, as before, the closer you get to 1, the calculation time is also multiplied by 10.

Impact of the Number of States on Convergence Speed

The number of iterations remains unchanged despite the increasing number of states in the MDPs. One might think that the variation in the number of states has no impact on the convergence of the method, but far from it. This constancy in the number of iterations in relation to the number of states hides a number of problems ranging from computational complexity, convergence time and storage memory.

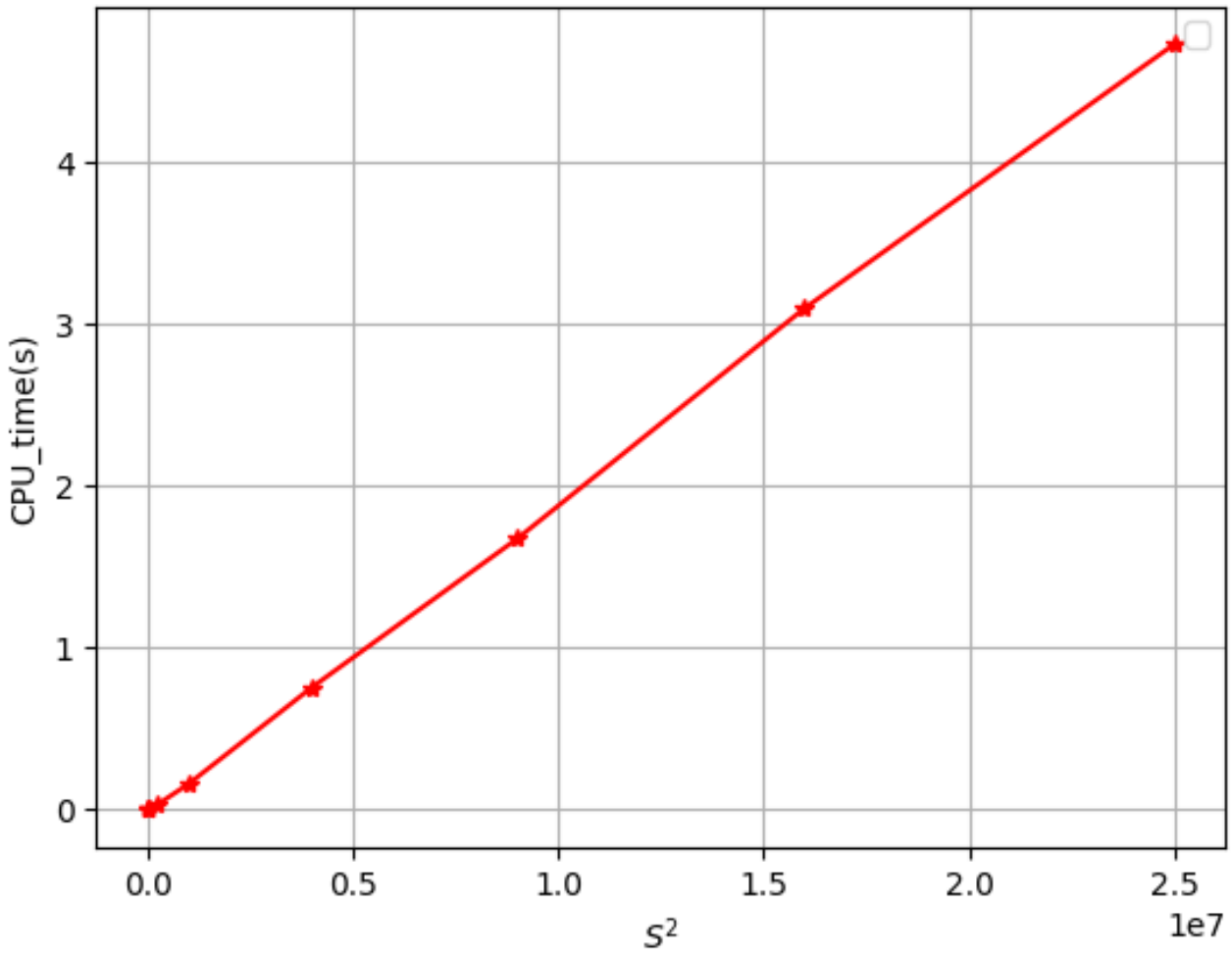
1. **Computational Complexity**: As the number of states increases, the number of computations required to update state values or policies increases. This is because each state must be updated based on all possible actions and transitions to all other possible states.
2. **Convergence Duration**:
   - **Value Propagation**: In Value Iteration algorithms, state values $V(s)$need to propagate through the entire state space. More states mean it takes longer for values to propagate and converge to optimal values.
   - **Bellman Updates**: Each update according to the Bellman equation requires calculations that include all states and all actions, meaning the computation time for each iteration increases with the number of states as you can see in the following figure.



3. **Memory and Storage**:

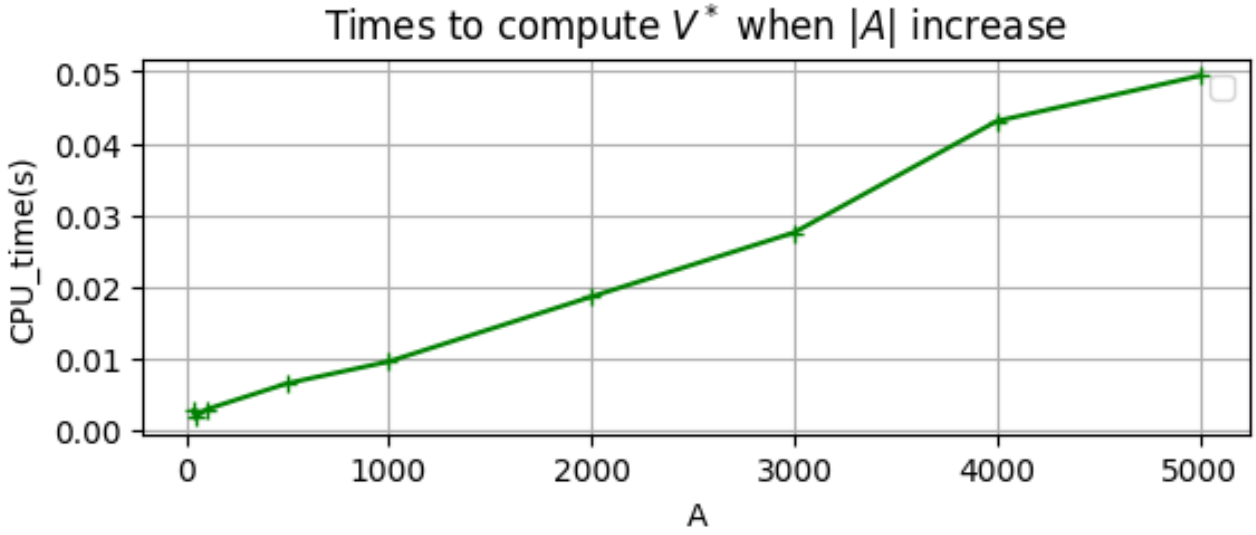
   - **Value and Policy Tables**: Storing value ($V(s)$) and policy ($\pi(s)$) tables increases linearly with the number of states. More states require more memory, which can also affect computation efficiency if memory resources are limited.

By observing the curve of the evolution of the calculation time of v as the number of states increases, we can see that this evolution describes the shape of a parabolic function. More precisely, it behaves almost like the function($x \mapsto x^2$) shown in the comparison figure below



Impact of the Number of Actions on Convergence Speed

Increasing the number of actions in a Markov Decision Process (MDP) while keeping the number of states fixed has a significant impact on the computation time required to determine the optimal value. The relationship between the number of actions and the computation time can be described as follows:



The figure illustrates the near-linear increase in computation time as the number of actions increases. In other words,the resulting curve is almost linear, indicating that as the number

of actions increases, the computation time increases proportionally. We can therefore say that this curve behaves almost like the straight line with equation($y = x$).

2.2 Policy Iteration

Policy iteration algorithm is also very well detailed in the many studies and research available in the literature as indicated in the following references (see [1, 2, 3, 7, 8, 6]).

Complexity: For a fixed value of γ, Policy iteration algorithm is both polynomial and strongly polynomial time algorithm. Here $|S^3|+|S|^2|A|$ (see Tab 2) is the assumed runtime per iteration of policy iteration

The policy iteration for discounted MDPs starts from an arbitrary policy π_0 and repeat the following steps until convergence:
1- **Policy evaluation** : keep current policy π fixed, find $V^{\pi_k}(.)$
Here, we have two way tofind $V^{\pi_k}(.)$.
• $a-$ **Exact computation of** $V^{\pi_k}(.)$.
We use:
$\boxed{V^{\pi_k} = (I - \gamma P^{\pi_k})^{-1} r^{\pi_k}}$ $with$ $P^{\pi_k}_{s,s'} = P(s'/s, \pi_k(s))$

Here we have to be very careful: the matrix P^π is a 2-dimensional matrix and r^π is a vector. In fact, the matrix P as we know it contains all the probabilities of taking any action in state s to transit to a state s' and the matrix r contains all the rewards associated with any action taken while in state s. These objects are obtained by extraction. This is a direct consequence of *lemma 1.1 and corollary 1.1*.

Remark: Using this method can be very costly as the number of states increases, since the greater the size of P^π, the more complicated it is to calculate its inverse, which is of complexity $\mathcal{O}(|S|^3)$. In this case, itś better to use the following iterative method to find V^π

• $b-$ **Approximate computation of** $V^{\pi_k}(.)$ Iterate simplified Bellman update until values converge:

$$\boxed{V^{\pi_k}_{i+1}(s) = r(s, \pi_k(s)) + \gamma \sum_{s'} P(s'/s, \pi(s)) V^{\pi_k}_i(s')}$$

Here we assume that $V^{\pi_0}_0(s) = 0$

2- **Policy improvement**: find the best action for $V^{\pi_k}(.)$ via one-step lookahead

$$\forall s \in \mathbb{S}, \pi_{k+1}(s) = \underset{a \in \mathbb{A}}{argmax} \left[r(s,a) + \sum_{s'} P(s'/s,a) V^{\pi_k}(s') \right].$$

To implement policy iteration, we need functions for both policy evaluation and policy improvement.

Approximate Policy evaluation

1: **input:** reward function r, transitional model P, discounted factor γ, convergence threshold θ, policy π_k, initialized vector $v^{\pi_k}_0$
2: **output:** converged value v
3: **while** True **do**

$\Delta \leftarrow 0$
for $s \in \mathcal{S}$ **do**
$temp \leftarrow v(s)$
$v_{i+1}^{\pi_k}(s) \leftarrow r(s, \pi_k(s)) + \gamma \sum_{s'} P(s'/s, \pi(s)) v_i^{\pi_k}(s')$
$\Delta \leftarrow \max(\Delta, |temp - v_{i+1}^{\pi_k}|)$
end for
if $\Delta < \theta$ **then**
break
end if
end while
v $\leftarrow v_{i+1}^{\pi_k}$
return v

Policy improvement

input: transitional model P,reward function r, vector v
output: updated policy π
for $s \in \mathcal{S}$ **do**
$\pi(s) \leftarrow \arg\max_a [r(s,a) + \gamma \sum_{s'} P(s'|s,a) v(s')]$
end for
return π

Main body of the algorithm

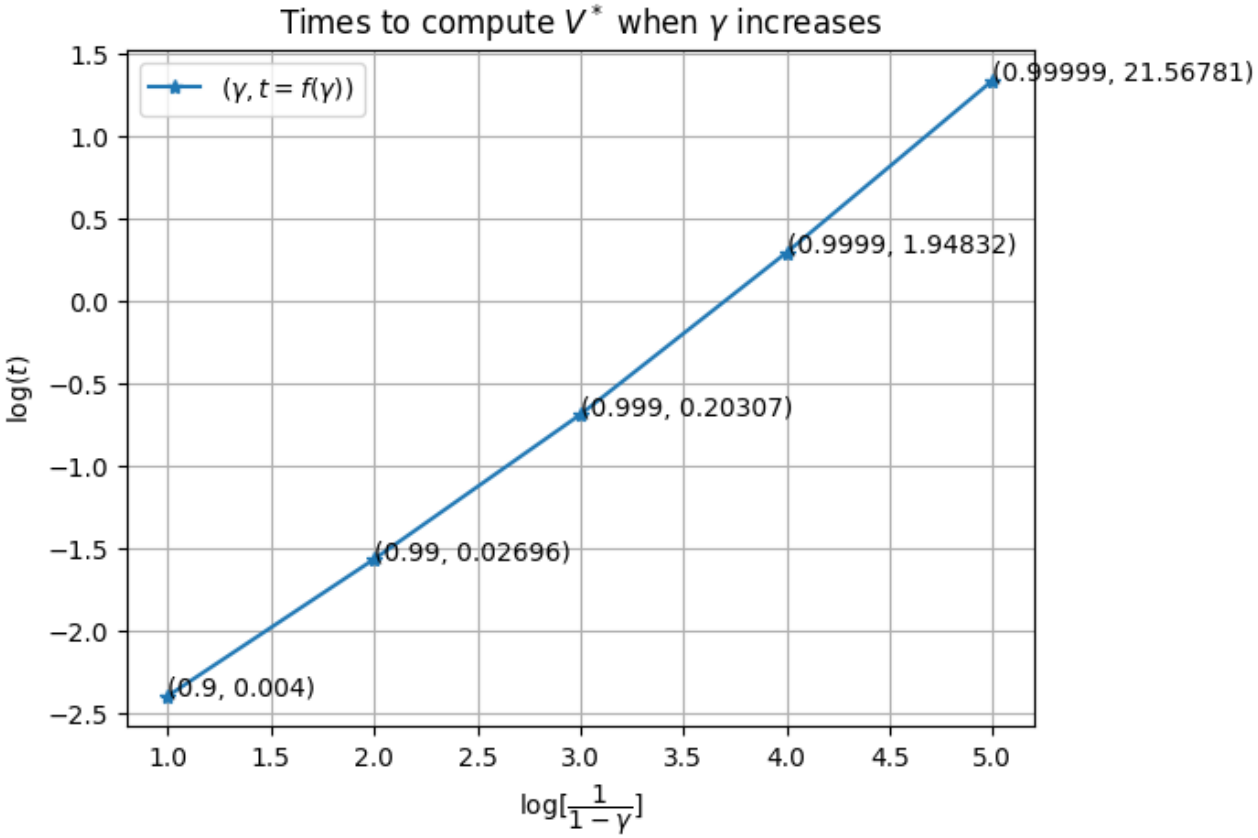
input: reward function r, transitional model P, discounted factor γ, random initialized policy π^0, vector v ,convergence threshold θ
output: optimal policy π^*
initialize $\pi(s)$ randomly
initialize v with zeros
change $\leftarrow$ True
while True **do**
$v \leftarrow$ policy evaluation$(r, P, \gamma, \theta, v)$
$\pi^k \leftarrow$ policy improvement(r, P, v)
if $\pi^k = \pi^{k+1}$(change $\leftarrow$ False) **then**
break
end if
end while
$\pi^* \leftarrow \pi$
return π^*

2.2.1 Numerical results and interpretation

The effects of γ on convergence.

Policy iteration algorithm converges in few iterations generally. For this reason, the effect of varying γ values is not noticeable if we decide to reason in terms of the number of iterations. In fact, the first step of *Policy Iteration*(**Policy Evaluation**) is actually an execution of the *Value Iteration's algorithm*, and this takes place at each iteration of the outer loop. These operations bring us very close to the optimal solution very quickly, which reduces the number of iterations of the *Policy Iteration's* algorithm to converge. It's therefore more convenient to look at the

execution time of the algorithm to see more clearly the impact of variations in γ values. The following figure give us a description about it.



As you can see from the figure, as gamma increases, so does calculation time. The smaller the gamma, the faster the algorithm converges. But as gamma gets closer and closer to 1, the algorithm takes longer to converge; and in particular, convergence time increases almost tenfold as it gets closer and closer to 1.

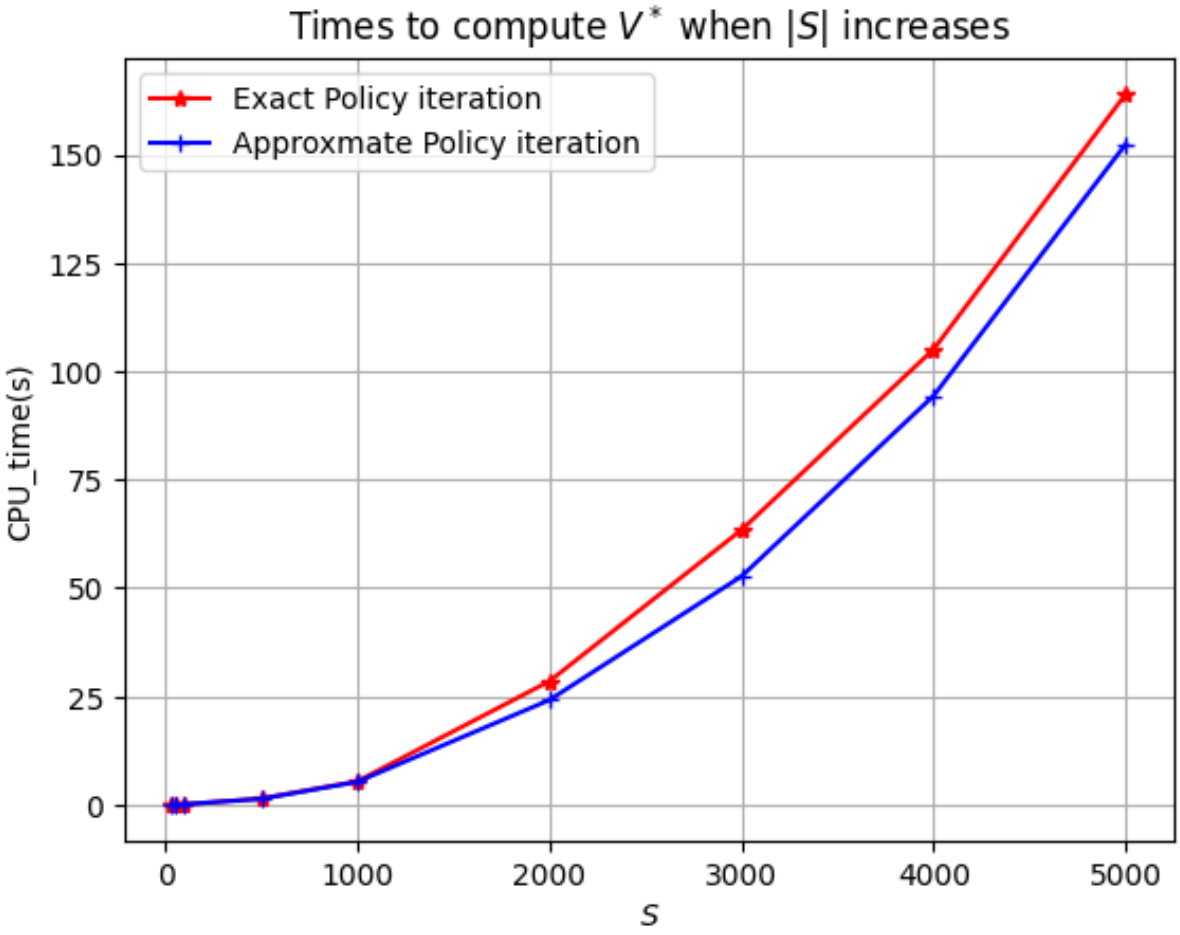
Impact of the Number of States on Convergence Speed

Policy Iteration's complexity being $\mathcal{O}(|S|^3 + |S|^2|A|)$, then increasing the number of states could have several consequences on the algorithm's performance. As mentioned in case of *Value Iteration*, these problems concern, among other things, computational complexity, convergence time and memory storage.

1. **Computational Complexity**: When the step of *Policy Evaluation* is executed, if the number of states increases, the number of computations required to update state values or to invert the matrix $(I - \gamma P^{\pi_k})$ for each policy π_k increases.

2. **Convergence Duration**:

   - **Value Propagation**: In Policy Iteration algorithms, state values $V(s)$need to propagate through the entire state space at each step of *Policy Evaluation*. More states mean it takes longer for values to propagate.
   - **Bellman Updates during *Policy Evaluation***: Each update according to the Bellman equation requires calculations that include all states and their corresponding actions $\pi_k(s)$, meaning the computation time for each iteration increases with the number of states.

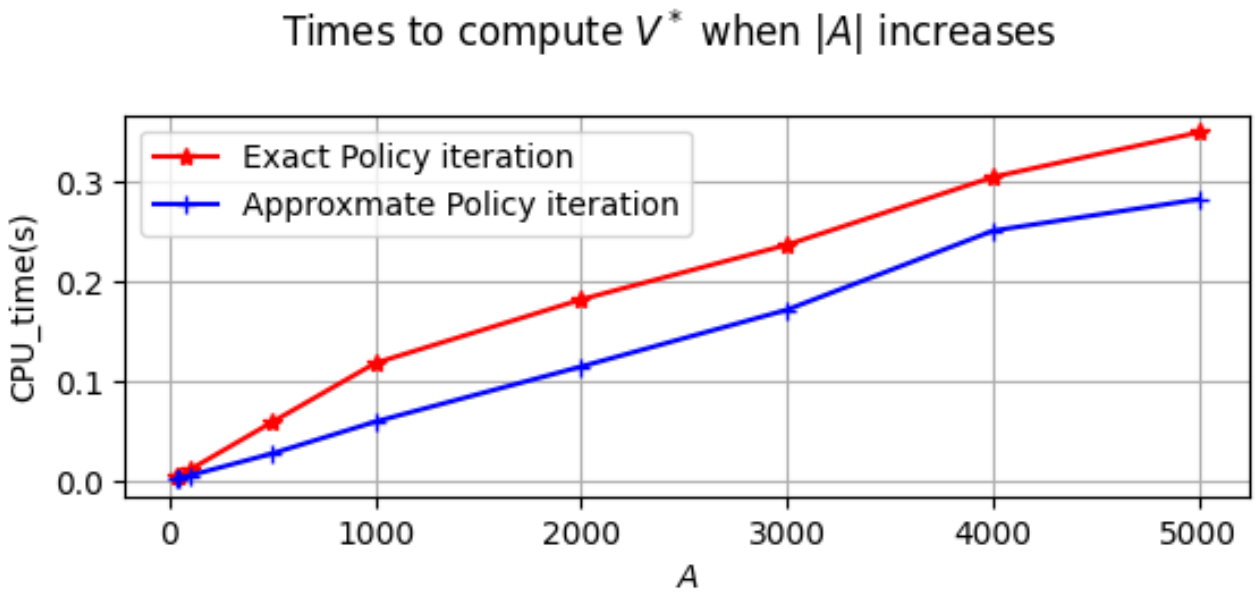


3. **Memory and Storage**:

   - **Value and Policy Tables**: Storing value ($V(s)$) and policy ($\pi(s)$) tables increases linearly with the number of states. More states require more memory, which can also affect computation efficiency if memory resources are limited.

Impact of the Number of Actions on Convergence Speed

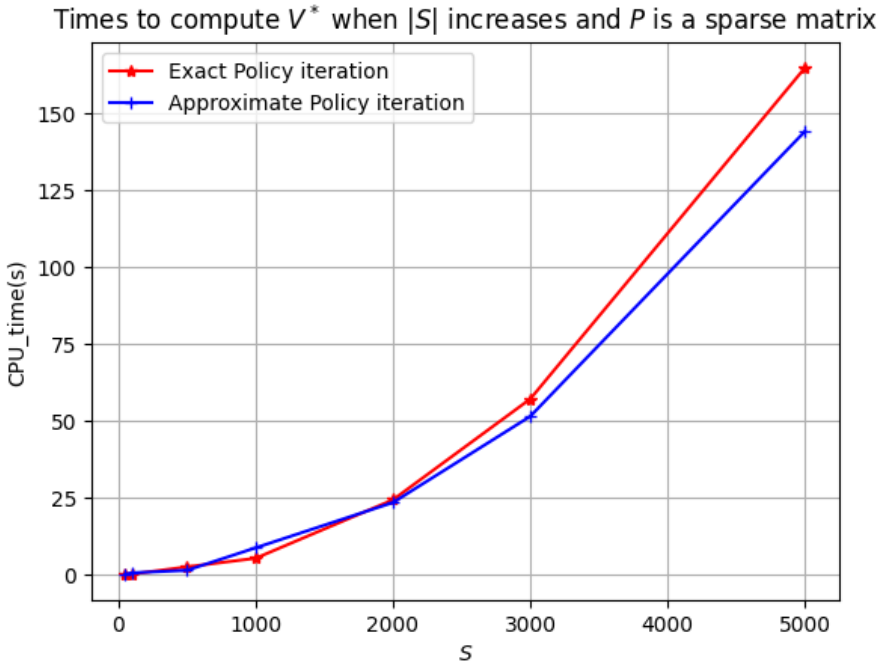
Increasing the number of available actions at each state also complicates calculations. In effect, to update the policy at the Policy Improvement stage, we need to propagate through all the action space for each state. The following figure shows the evolution of calculation time as the number of actions increases.



Remark 2.1. As can be seen from the two last previous figure above, the curve of the exact method is always above that of the approximate method. This implies that the exact method takes longer to converge than the approximate method as the number of actions or states increases. This is the cost of the matrix inversion that takes place at each iteration in the exact method during the PE step.

2.2.2 Test of Policy iteration methods on sparse matrix

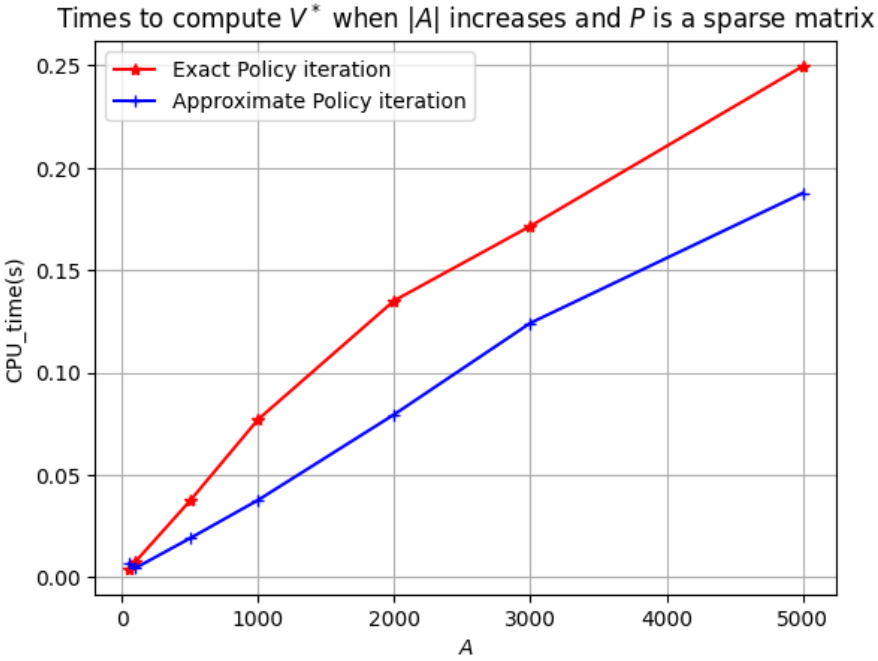
Impact of the Number of States on Convergence Speed when P is a sparse matrix



As you can see, the Approximate Policy Iteration method performs better than the Exact Policy Iteration method. Approximate Policy Iteration converges in less time on sparse matrices than on dense matrices (almost 10 seconds less than on dense matrices). On the other hand, the exact method shows no improvement despite the transition matrix being sparse. This is because the inversion of a matrix does not depend on its density. A sparse matrix generally has a dense

inverse. However, with sparse matrices, the gap between the exact and approximate methods becomes much wider as the number of states increases. We can easily see that from 5000 states upwards.

Impact of the Number of Action on Convergence Speed when P is a sparse matrix



2.3 Linear Programming Approach

For more details on Linear Programming approach to solve MDPs, please consult the following references(see [1, 7, 8, 6, 10]).

Complexity: For a fixed value of γ, Linear programming algorithm is both polynomial and strongly polynomial time algorithm. Here $|S^3||A|$ (see Tab 2) is the assumed runtime per iteration of Linear programming

As we've broached, the algorithms of PI and VI can be used to solve MDPs. But instead of using those approach, we can remark that MDPs can be reformulated as a linear programming problem. Let's describe this approach very quickly.
From Theorem 1.2 and Definition 1.2.2, we search for Q^* such that $Q^* = TQ^*$ and
$V^*(s) = \max_{a \in A} Q^*(s,a)$①.
$\forall (s,a) \in S \times A$, one has:

$$\begin{aligned} Q^*(s,a) &= r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\max_{a' \in A} Q(s',a') \right] \\ &= r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^*(s) \right] \quad ② \end{aligned}$$

② in ① gives:

$$\begin{aligned} V^*(s) &= \max_{a \in A} \left[r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^*(s') \right] \right] \\ &= \max_{a \in A} \left[r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right] \end{aligned}$$

Then we can define a new operator (also named *Bellman Operator* τ defined by:

$$\tau : \mathbb{R}^{|\mathbb{S}|} \underset{V \mapsto \tau(V)}{\to} \mathbb{R}^{|\mathbb{S}|}$$

with $\tau(V)(s) = \max_{a \in A} [r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V^*(s')]$
Now, the problem to solve is to find V such that $V = \tau(V)$ .This problem can be reformulated as the following optimization problem with variables $V \in \mathbb{R}^{|\mathbb{S}|}$:

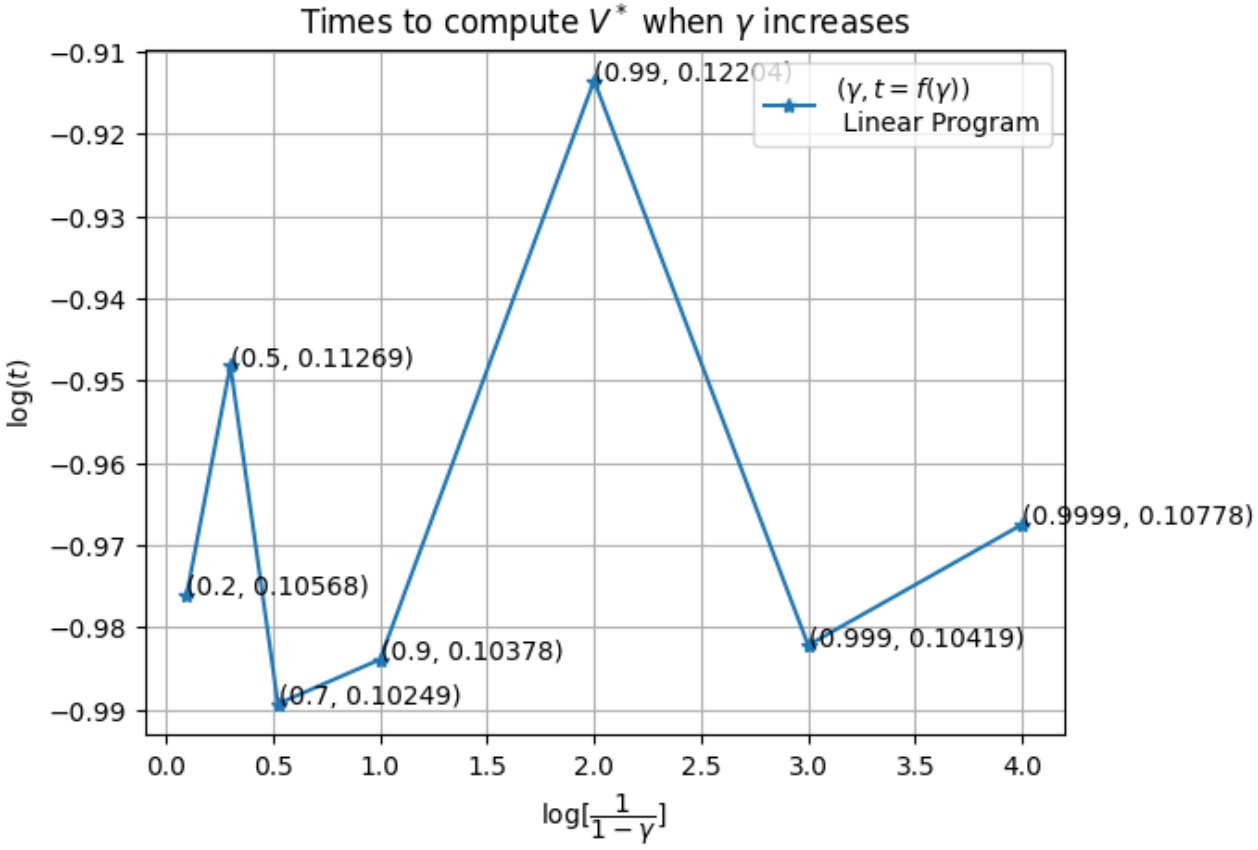
$$\begin{aligned} \min \quad & \sum_s \mu(s)V(s) \\ \text{subject to} \quad & V(s) \geq r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s') \\ & \forall a \in A, \forall s \in S \end{aligned}$$

This is a linear programming problem.
To solve this linear programming (LP) problem, we are going to use **Pulp** library in python as follows:

- **Import PuLP**: We import the PuLP library.
- **Define the Problem**: We create an LP problem instance with `pulp.LpProblem`.
- **Variables**: Define the decision variables using `pulp.LpVariable`.
- **Objective Function**: Add the objective function to the LP problem using `+=`.
- **Constraints**: Add constraints similarly using `+=`.
- **Solve the Problem**: Call the `solve()` method to solve the LP.
- **Results**: Print the status, values of the decision variables, and the objective function.

2.3.1 Numerical results and interpretation

Impact of γ on Convergence Speed



The impact of the discount factor γ on the computation time of the optimal solution V^* is not linear.Variations in γ influence the complexity of the problem in a non-trivial manner. Indeed ;

- Between 0.9 and 0.99, We observe a significant increase in calculation time up to 0.12204 indicating that very high values of γ significantly increase computation time.
- But, between 0.99 to 0.999, there here is a significant reduction in calculation time to 0.10419, which may seem counter-intuitive, but perhaps indicates reduced complexity in some specific algorithm and data configurations.

The behavior observed in the curve could indeed be influenced by some of the open problems and well-known challenges in the field of linear programming, particularly those related to the simplex algorithms and other solution methods. Here are some specific points to consider:

- **Complexity of the Simplex**[11, 12, 16, 15, 17]

  The simplex algorithm, while efficient in practice for many problems, has exponential complexity in the worst case. This means that, for certain specific configurations of γ, the computation time can increase unpredictably.

- **Conditioning of the Matrix**[11, 12]

  High values of γ can affect the conditioning of the constraint matrix of the linear program. Poor conditioning can lead to convergence difficulties and increases in computation time. This is a common problem in the solution of linear systems and directly affects the performance of linear programming algorithms.

- **Cycling and Degenerate Pivoting**[11, 12]

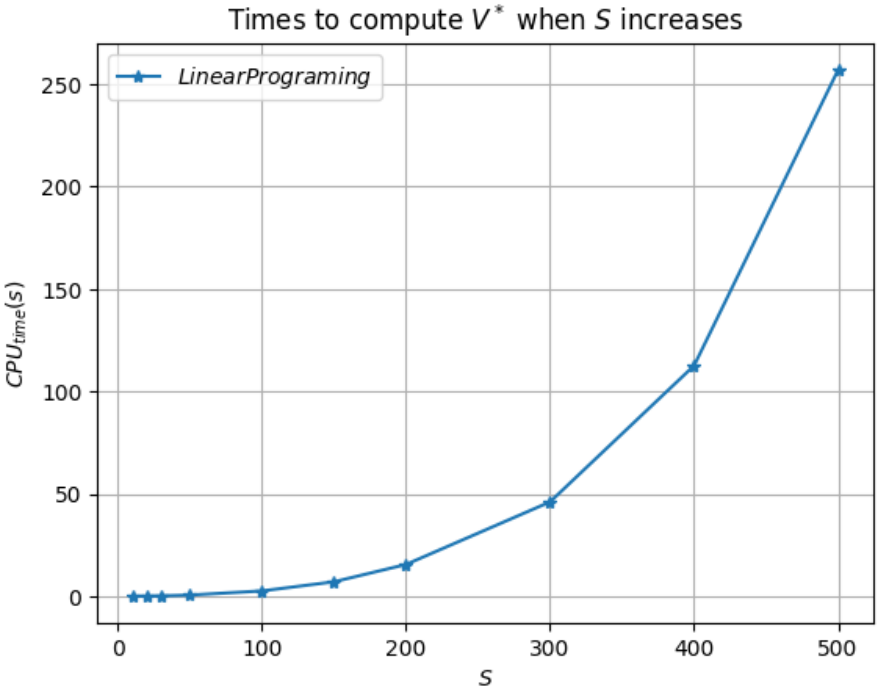
  The phenomenon of cycling, where the simplex algorithm can revisit the same basic solutions without making progress, is a known problem that can significantly lengthen computation time. Specific values of γ can exacerbate these conditions, especially if they lead to degenerate solutions (where multiple optimal solutions exist).

- **Oscillations in Computation Time**

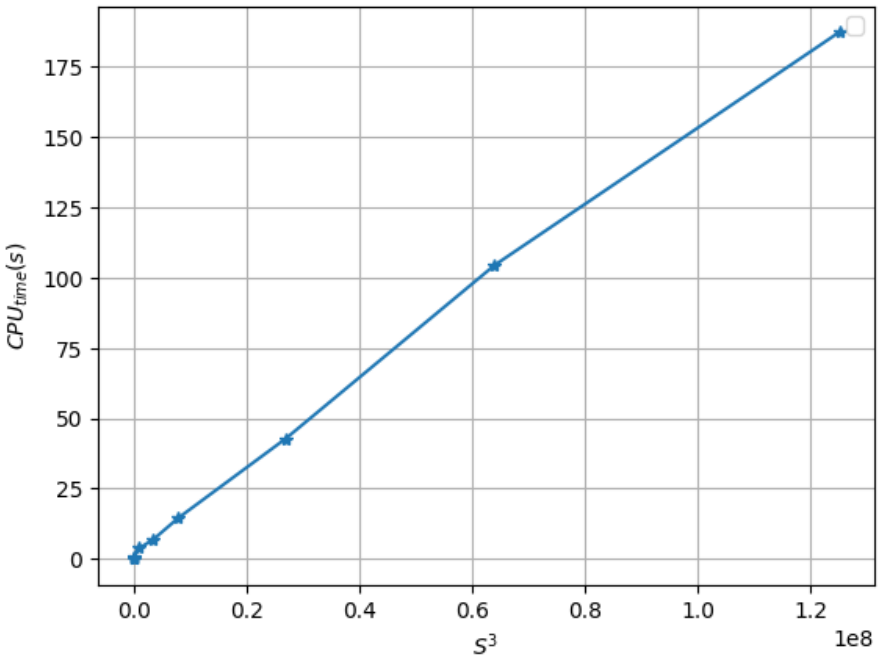
  Non-linear variations and oscillations in computation time with changes in γ can also be due to complex interactions between the specific characteristics of the solution algorithm and the particular structure of the MDP being considered. For example, certain states and transitions may become more or less important at different levels of γ, affecting the difficulty of the problem in a non-linear way.

This suggests that, in practice, it may be useful to test several values of γ to strike a balance between future reward accuracy and computational efficiency. Algorithms for solving MDPs by linear programming may react differently depending on the specific parameters of the problem

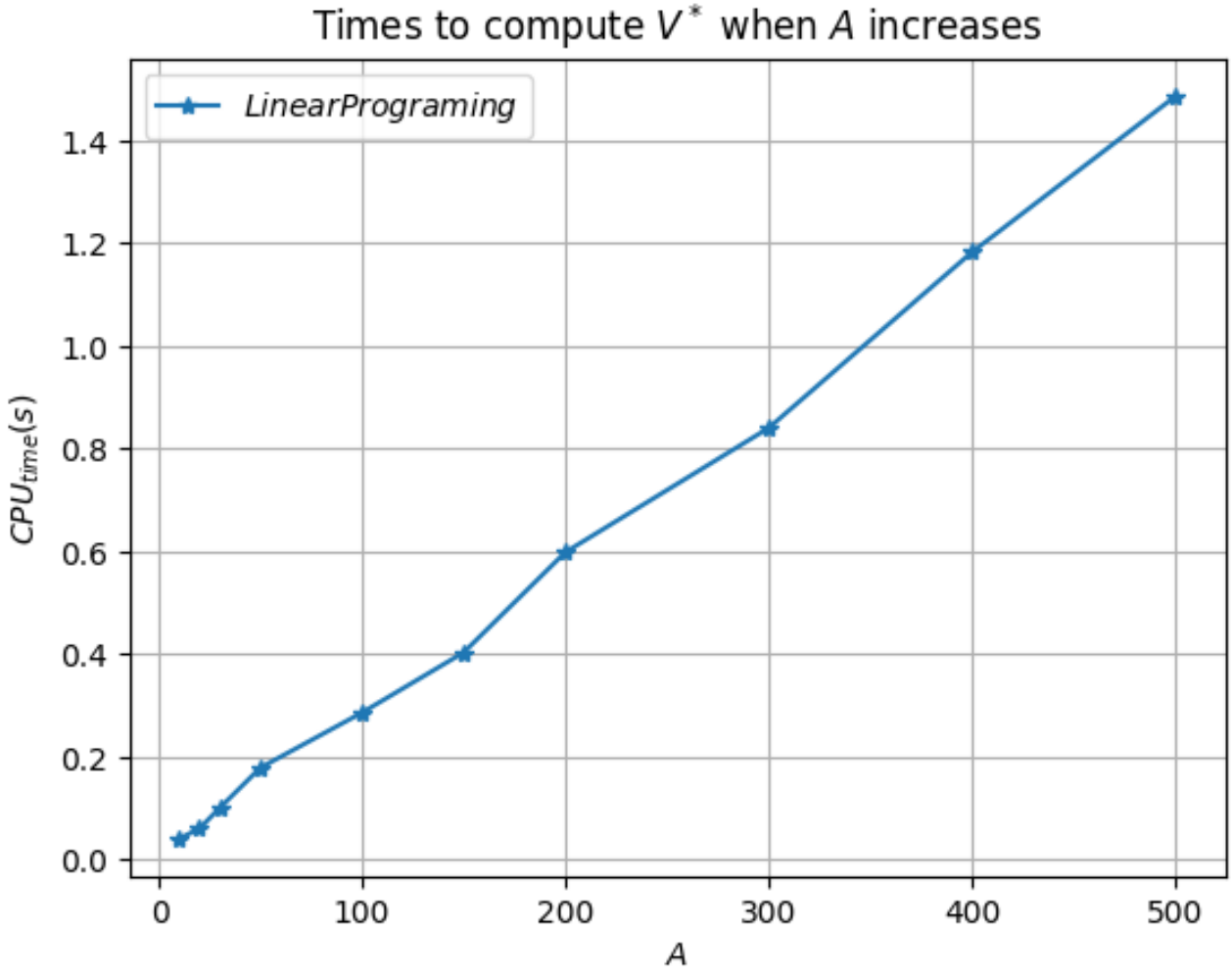
Impact of the Number of States on Convergence Speed



LP-algorithm becomes much more complex to solve as the number of states increases, so it takes longer to converge. Indeed, as the state space increases, so does the number of degrees of freedom (unknowns:$|S|$) in the cost function and the number of constraints($|S|.|A|$) in the problem, making the solution much more complex and more complicated to find.
The calculation time behaves like a cubic function ($x \mapsto x^3$:see figure below). This corresponds exactly to the theoretical results (refer to Table 2)



Impact of the Number of Actions on Convergence Speed



Value Iteration vs Linear Programming vs Policy Iteration

Comparison table						
States	Value Iteration		Approximate Policy Iteration		Linear Programming	
S_i	**Value** $V(s_i)$	**Policy** π^*	**Value** $V(s_i)$	**Policy** π^*	**Value** $V(s_i)$	**Policy** π^*
s_0	9.1055857	a_3	9.100605	a_3	9.1064532	a_3
s_1	9.2387334	a_3	9.229618	a_3	9.2396009	a_3
s_2	9.2370624	a_2	9.202891	a_2	9.2379299	a_2
s_3	9.2802963	a_3	9.290088	a_3	9.2811638	a_3
s_4	9.2639249	a_2	9.256704	a_2	9.2647924	a_2
s_5	9.1822204	a_4	9.169880	a_4	9.1830879	a_4
s_6	9.3022003	a_1	9.290648	a_2	9.3030678	a_1
s_7	9.2214848	a_0	9.211004	a_0	9.2223523	a_0
s_8	9.2665669	a_0	9.258590	a_0	9.2674345	a_0
s_9	9.0516525	a_4	9.043167	a_4	9.05252	a_4

Exact Policy Iteration vs Approximate Policy Iteration

Comparison table				
States	Exact Policy Iteration		Approximate Policy Iteration	
S_i	**Value** V	**Policy** π^*	**Value** V	**Policy** π^*
s_0	9.101454	a_3	9.100605	a_3
s_1	9.230468	a_3	9.229618	a_3
s_2	9.203740	a_2	9.202891	a_2
s_3	9.290938	a_3	9.290088	a_3
s_4	9.257554	a_2	9.256704	a_2
s_5	9.170729	a_4	9.169880	a_4
s_6	9.291498	a_2	9.290648	a_2
s_7	9.211854	a_0	9.211004	a_0
s_8	9.259439	a_0	9.258590	a_0
s_9	9.044017	a_4	9.043167	a_4

The values $V(s_i)$ from these methods are very close to each other, indicating that these methods are effectively converging to similar optimal values. Slight differences in values are due to differences in numerical precision and the algorithms' convergence criteria.

The optimal policies π^* (the actions associated with each state that maximize the expected reward) are also provided for each state. For most states, the policies derived from both methods are identical. This indicates a strong agreement between the two methods on the best actions to take from each state. There are a few differences in policies for some states (e.g., s_6) where **Policy Iteration** takes action a_2 and **Linear programming & Value Iteration** take action a_1.
One might think that the optimal policy provided by these methods are not equivalent, but this is not true. In fact, these are two optimum policies

2.4 Python code of the methods

You can access the Google Colab notebook of all methods by clicking on the following links:

- Value Iteration's algorithm
- Policy Iteration's algorithm
- Linear Programming' algorithm

For those who will print the document
Value Iteration:
https://colab.research.google.com/drive/1M-Bxsr-Xwh0_Q1M-0Ps5ub7y5GjUXJLX?usp=sharing
Policy Iteration:
https://colab.research.google.com/drive/1r8W5EpijgxuojrJ9V6cN7pIsAC_zpB9X?usp=sharing
Linear Programming' algorithm:
https://colab.research.google.com/drive/1dBiOB2JzkZ6uffLmBeXbU8nZ6UXaUtO7?usp=sharing

Conclusion

Markov Decision Processes (MDPs) offer a powerful framework for modeling decision-making in stochastic environments. Among the various methods for solving MDPs, value iteration, policy iteration and linear programming stand out as the most widely used. In our work, we have studied and verified the theoretical complexity of these different methods through numerical simulations by varying the parameters of the algorithms. This practical approach allowed us to observe how changes in parameters impact the performance and convergence of each method.
Summarizing each of the three MDP resolution methods studied, it can be seen that:
Value Iteration is an iterative algorithm that focuses on updating the value function until convergence. It is relatively simple to implement and guarantees convergence to the optimal value function. However, it can require a large number of iterations to achieve an accurate solution, especially in complex MDPs.
Policy Iteration combines policy evaluation and policy improvement steps to iteratively refine the policy. Starting with an arbitrary policy, it alternates between evaluating the current policy and improving it. This method typically converges faster than Value Iteration in terms of the number of iterations, but each iteration is computationally more expensive due to the need for solving systems of linear equations during policy evaluation.
Linear Programming offers a different approach by formulating the MDP as a linear optimization problem. This method directly solves for the optimal value function or policy by leveraging linear programming solvers.
Each method has its own strengths and weaknesses, making them suitable for different types of MDPs and computational environments. Value Iteration is advantageous for its simplicity and ease of implementation, Policy Iteration for its efficiency in practice, and Linear Programming for its precision.
In conclusion, the choice of method depends on the specific requirements of the problem at hand, including the size of the state and action spaces, the desired accuracy, and the available computational resources. Understanding the trade-offs and applications of these methods allows practitioners to effectively leverage MDPs for decision-making in uncertain and dynamic environments.

BIBLIOGRAPHY

[1] Alekh Agarwal, Nan Jiang, Sham M. Kakade ,Wen Sun, *Reinforcement Learning: Theory and Algorithms, January 31, 2022*

[2] Amir massoud Farahmand, Remi Munos, Csaba Szepesvari, *Error Propagation for Approximate Policy and Value Iteration*

[3] Alekh Agarwal, Mikael Henaff, Sham Kakade, and Wen Sun. *Pc-pg: Policy cover directed exploration for provable policy gradient learning. NeurIPS, 2020a.*

[4] Alekh Agarwal, Sham Kakade, Akshay Krishnamurthy, and Wen Sun. Flambe: *Structural complexity and representa- tion learning of low rank mdps. NeurIPS, 2020b.*

[5] Sutton and Barto, *Reinforcement Learning: An Introduction*

[6] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. *Learning and Planning in Markov Decision Processes"*

[7] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. *On the Complexity of Solving Markov Decision Problems*

[8] Christos H. Papadimitriou and John N. Tsitsiklis. *The complexity of Markov decision processes.Math. Oper. Res., 12(3):441–450, 1987.URL: https://doi.org/10.1287/moor.12.3.441, doi:10.1287/moor.12.3.441.*

[9] Brafman and Tennenholtz, *On the Complexity of Value Iteration*

[10] Dimitri P. Bertsekas and John Tsitsiklis, *Neuro-Dynamic Programming*

[11] Donald Goldfarb, *On the Complexity of the Simplex Method*

[12] Y. Disser and M. Skutella. *In defense of the simplex algorithm's worst-case behavior.CoRR, abs/1311.5935, 2013.*

[13] Martin L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*

[14] O. Friedmann, *An exponential lower bound for the parity game strategy improvement algorithm as we know it.In Proc. of LICS, pages 145–156, 2009.*

[15] O. Friedmann, *An exponential lower bound for the latest deterministic strategy iteration algorithms.Logical Methods in Computer Science, 7(3), 2011.*

[16] O. Friedmann. *A subexponential lower bound for Zadeh's pivoting rule for solving linear programs and games.In Proc. of IPCO, pages 192–206, 2011.*

[17] O. Friedmann, T. D. Hansen, and U. Zwick. *A subexponential lower bound for the random facet algorithm for parity games.In Proc. of SODA, pages 202–216, 2011.*

[18] O. Friedmann, T. D. Hansen, and U. Zwick. *Subexponential lower bounds for randomized pivoting rules for the simplex algorithm.In Proc. of STOC, pages 283–292, 2011.*

Codes of different algorithm

- Value Iteration

```python
# -*- coding: utf-8 -*- #


import numpy as np
import matplotlib.pyplot as plt

##----------------------------------Define the MDP components-------------------------------##

num_states = 20
num_actions = 10
states = [f's{i}' for i in range(num_states)]
actions = [f'a{i}' for i in range(num_actions)]

###------------------------ Transition probability matrix P[s, a, s']-------------------------###

# Random generation of matrix
def Markov_decision_matrix(num_states, num_actions):
  Markov = []
  for s in range(num_states):
    matrix = np.random.rand(num_actions,num_states)
    # Normalization of each row
    matrix = matrix/matrix.sum(axis=1, keepdims=True)
    Markov.append(matrix)
  return np.array(Markov)
np.random.seed(42)
P = Markov_decision_matrix(num_states, num_actions)




##-------------------------Reward function R[s, a]-------------------------------------##
R = np.round(np.random.rand(num_states, num_actions),2)  # random generation of R


# Action_value Function Q(initial value)
Q = np.zeros((num_states, num_actions))

# Discount factor
```

```
gamma = 0.7

Q.shape, R.shape, P.shape

V = np.zeros(num_states)
#np.dot(P[a,:,:],V)
np.dot(P,V).shape # The product make really sense

##------------ Definition of Bellman operator-----------------------##

def Bellman_Operator(Q,R,P,gamma):
  V_Q = np.max(Q, axis=1)
  return R + gamma*np.dot(P,V_Q)

#Bellman_Operator(Q,R,P,gamma)# Test pour Q = Zeros_23

##-------------------------Value iteration algorithm----------------------##

def Matrix_value_iteration(Q, P, R, gamma, theta=0.0001):
  iter = 0
  err = []
  while True:
    TQ=Bellman_Operator(Q,R,P,gamma)
    delta = np.max(np.abs(Q - TQ))
    err.append(delta)
    Q = TQ
    iter+=1
    if delta < theta:
      break
  V_star = np.max(Q, axis=1)
  return V_star, iter, err, R.shape

# Commented out IPython magic to ensure Python compatibility.
# %timeit Matrix_value_iteration(Q,P, R, gamma)

##----------------------------Compute the optimal value function----------------##

optimal_values, k, err,_ = Matrix_value_iteration(Q,P, R, gamma)
print("Optimal Value Function:")
for i, value in enumerate(optimal_values):
    print(f"V({states[i]}) = {value:.2f}")

print("Nombre d'itérations:",k)
print("Erreur",err)

"""> *After convergence, the optimal policy is derived by selecting the action that maximizes the expected value for each state*.
$$\forall s\in \mathbb{S}, \pi^*(s) = \underset{a\in\mathbb{A}}{argmax}Q^*(s,a) = \underset{a\in\mathbb{A}}{argmax}\left[r(s,a)+\gamma\underset{s'}{\sum}P(s'/s,a)
"""

```

```python
##----------------------Optimal policy---------------------------------##

def optimal_policy(P, R, V, gamma):
    num_states, num_actions, _ = P.shape
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        policy[s] = np.argmax([
            R[s,a] + sum(gamma *P[s, a, s_next] * (
             ↪  V[s_next])#np.argmax(Q[:,s])
                  for s_next in range(num_states))
               for a in range(num_actions)])
    return policy

policy_star = optimal_policy(P, R, optimal_values, gamma)
print("\nOptimal Policy:")
for i, action in enumerate(policy_star):
    print(f"({states[i]}) = {actions[action]}")

##----------------------Plotting function---------------------##

def loss_Plot(k,err,gamma,sh,type_err):
    K = np.arange(k)
    if type_err == "log":
      plt.plot(K, np.log(err), label = f"$\gamma$={gamma}:({k}
       ↪  iterations)")
      plt.xlabel("Iterations")
      plt.ylabel("$\log_{(||Q^k - TQ^k||)}$")
      plt.title(f"S = {sh[0]}, A = {sh[1]}")
      plt.grid(visible=True)
      plt.legend()
    else:
      plt.plot(K, err, label = f"$\gamma$={gamma}:({k} iterations)")
      plt.xlabel("Iterations")
      plt.ylabel("Error")
      plt.title(f"S = {sh[0]}, A = {sh[1]}")
      plt.grid(visible=True)
      plt.legend()
    return 0

import time
###----------------------Analyze of convergence based on gamma's
 ↪  value--------------------###
Gamma = [1-1e-1, 1-1e-2, 1-1e-3, 1-1e-4]
Times = []
for gam in Gamma:
  start = time.time()
  _, k, err,sh = Matrix_value_iteration(Q,P, R, gam)
  end = time.time()
  Times.append((end-start))
  loss_Plot(k,err,gam,sh,"log")

###----------------------Plotting of Gamma/times curve--------------###
Gamma_log = [np.log10(1/(1-g)) for g in Gamma] # Log(times)
Times_log = [np.log10(t) for t in Times] # Log(1/(1-gamma))
```

```
plt.figure()
for i in range(len(Times)):
  plt.text(Gamma_log[i],Times_log[i],f'{Gamma[i],round(Times[i],5)}')

plt.plot(Gamma_log,Times_log, "-*", label="$(\gamma,t=f(\gamma))$")
plt.xlabel("$\log[\dfrac{1}{1-\gamma}]$")
plt.ylabel("$\log(t)$")
plt.title("Times to compute $V^*$ when $\gamma$ increase ")
plt.legend()
plt.grid(1)

###----------------------Analyze of convergence by varying the number of states--------------------###

gam = 0.7 # Here gamma is constant
actions = 10 # Number of action is constant
Times_state = []
States = [35, 50, 100,500, 1000, 2000, 3000, 4000, 5000] # List of states

for state in States:
  np.random.seed(42)
  P_s = Markov_decision_matrix(state, actions)
  R_s = np.round(np.random.rand(state, actions),2)
  Q_s = np.zeros((state, num_actions))
  start = time.time()
  _, k, err,sh = Matrix_value_iteration(Q_s,P_s, R_s, gam)
  end = time.time()
  Times_state.append((end-start))
  #loss_Plot(k,err,gam,sh,"")

plt.figure()
plt.plot(States, Times_state, "-*r", label="")
plt.xlabel("$S$")
plt.ylabel("CPU_time(s)")
plt.title("Times to compute $V^*$ when $|S|$ increase ")
plt.grid(1)
plt.legend()

##--------------Proof that CPU_time of VI when number of state behave like quaratic function-------#
plt.figure()
ss = [s**2 for s in States]

plt.plot(ss, Times_state, "-*r", label="")
plt.xlabel("$S^2$")
plt.ylabel("CPU_time(s)")
plt.grid(1)
plt.legend()



###----------------------Analyze of convergence by varying the number of actions--------------------###

```

```
gam = 0.7 # Here gamma is constant
state = 10 # Number of state is constant
Times_action = []
Actions =[35, 50, 100,500, 1000, 2000, 3000, 4000, 5000] # List of states

for i, action in enumerate(Actions):
  np.random.seed(42)
  P_a = Markov_decision_matrix(state, action)
  R_a = np.round(np.random.rand(state, action),2)
  Q_a = np.zeros((state, action))
  start = time.time()
  _, k, err,sh = Matrix_value_iteration(Q_a,P_a, R_a, gam)
  end = time.time()
  Times_action.append(end-start)
  #loss_Plot(k,err,gam,sh,"")

plt.figure()
#for i in range(len(Times_state)):
  ↪ #plt.text(Times_action[i],States[i],f'{States[i],round(Times_action[i],5)}')
plt.subplot(2, 1, 1)
plt.plot(Actions,Times_action, "-+g", label="")
plt.xlabel("A")
plt.ylabel("CPU_time(s)")
plt.title("Times to compute $V^*$ when $|A|$ increase")
plt.legend()
plt.grid(1)

#--------------
plt.figure()
plt.subplot(2, 1, 1)
plt.plot(Actions,Times_action, "-*g", label="")
plt.ylabel("CPU_time(s)")
plt.legend()
plt.grid(1)
plt.subplot(2, 1, 2)
plt.plot(Actions,Actions, "-b", label="$y=x$")
plt.xlabel("A")
plt.ylabel("A")
plt.legend()
plt.grid(1)
```

- Policy Iteration

```
            # -*- coding: utf-8 -*-


import numpy as np
import matplotlib.pyplot as plt
import time
from scipy import sparse


```

```
##-----------------------------------Define the MDP components---------------------------------##

num_states = 10
num_actions = 5
states = [f's{i}' for i in range(num_states)]
actions = [f'a{i}' for i in range(num_actions)]


###------------------------ Transition probability matrix P[s, a, s']-------------------------###

# Random generation of matrix
def Markov_decision_matrix(num_states, num_actions):
  Markov = []
  for s in range(num_states):
    matrix = np.random.rand(num_actions,num_states)
    # Normalization of each row
    matrix = matrix/matrix.sum(axis=1, keepdims=True)
    Markov.append(matrix)
  return np.array(Markov)
np.random.seed(42)
P = Markov_decision_matrix(num_states, num_actions)




##-------------------------Reward function R[s, a]-------------------------------------##
R = np.round(np.random.rand(num_states, num_actions),2)  # random generation of R


# Action_value Function Q(initial value)
Q = np.zeros((num_states, num_actions))

# Discount factor
gamma = 0.9

Q.shape, R.shape, P.shape

"""
# Policy Iteration
Repeat steps until convergence:

1- $\textbf{Policy evaluation}:$ keep current policy $\pi$ fixed, find $V^{\pi_k}(.)$
+ Approximative evalution of $V^{\pi^k}:$

Iterate simplified Bellman updaate until values converge:
$$\boxed{V_{i+1}^{\pi_k}(s')=r(s,\pi_k(s))+\gamma\underset{s'}{\sum}P(s'/s,\pi(s))V_
\text{(Le faire juste après comme Approximate Policy Iteartion, bien choisir le tetha de l'ordre de) (1-\gamma)^2 \epsilon/4}$$
 Here we assume that $V_0^{\pi_0}(s) = 0$
```

```
58
59
60  + Exact evaluation of $V^{\pi^k}:$
61
62 $$\boxed{V^{\pi_k} =(I-\gamma P^{\pi_k})^{-1}r^{\pi_k}} \ with \
   ↪  P^{\pi_k}_{s,s'}=P(s'/s,\pi_k(s))
63 $$
64
65
66 2- $\textbf{Policy improvement}$: find the best action for $V^{\pi_k}(.)$
   ↪  via one-step lookahead
67
68 $$\forall s\in \mathbb{S}, \pi_{k+1}(s) =
   ↪  \underset{a\in\mathbb{A}}{argmax}\left[R(s,a)+\underset{s'}{\sum}P(s'/s,a)V^{\pi_
69 """
70
71 np.random.seed(42)
72 ##---------Random initialization of pi--------------##
73 pi = np.random.randint(num_actions, size = num_states)
74 pi
75
76 I = np.eye(num_states)
77 #I
78
79 ##----------Function to evaluate the utility V on the the arbitrary
   ↪  policy--------------##
80
81 def Exact_Policy_evaluation(pi, P, R, gamma):
82     num_states, num_actions, _ = P.shape
83     P_pi = np.array([P[s,pi[s],:] for s in range(num_states)])
84     R_pi = np.array([R[s,pi[s]] for s in range(num_states)])
85
86     I = np.eye(num_states)
87     V = np.dot(np.linalg.inv(I - gamma*P_pi), R_pi)
88
89     return V
90
91 def Approximate_Policy_evaluation(pi,V, P, R, gamma, theta):
92     count = 0
93     num_states, num_actions, _ = P.shape
94     P_pi = np.array([P[s,pi[s],:] for s in range(num_states)])
95     R_pi = np.array([R[s,pi[s]] for s in range(num_states)])
96
97     while True:
98         delta = 0
99         TV=R_pi + gamma*sparse.coo_array.dot(P_pi,V)
100        delta=np.max(np.abs(V-TV))
101        V=TV
102        count+=1
103        #print(dt)
104
105        if delta <=theta:
106            break
107    #print("Number of iteration loop intern approximate",count)
```

```
    return V

##-----------------------Function to compute the greedy polyci-------------------------##

def Policy_improvement(V, P, R, gamma):
  num_states, num_actions, _ = P.shape
  policy = np.zeros(num_states, dtype=int)
  for s in range(num_states):
      policy[s] = np.argmax([R[s,a]+sum(gamma *P[s, a, s_next] * (V[s_next])
                  for s_next in range(num_states))#R(s,a)+...
              for a in range(num_actions)
      ])
  return policy

##-----------------------The main algorithm-----------------------------##

def Policy_iteration(pi,V,P,R, gamma, method="Exact", theta = 0.0001):
  iter = 0

  if method=="Exact":
    while True:
      V_Q = Exact_Policy_evaluation(pi, P, R, gamma)
      greedy_pi = Policy_improvement(V_Q, P, R, gamma)
      iter+=1
      if greedy_pi.all() == pi.all():
        break
      pi = greedy_pi

  elif method=="Approx":
      V_Q = np.copy(V)

      while True:
        V_old = np.copy(V_Q)
        V_Q = Approximate_Policy_evaluation(pi,V_old, P, R, gamma, theta)
        greedy_pi = Policy_improvement(V_Q, P, R, gamma)
        iter+=1

        if greedy_pi.all() == pi.all():
          break

        pi = greedy_pi
  return pi, V_Q, iter

V = np.zeros(num_states)

optimal_policy, optimal_values , k= Policy_iteration(pi,V,P,R,gamma,"Approx", theta=0.0001)
print("Optimal Value Function:")
for i, value in enumerate(optimal_values):
    print(f"V({states[i]}) = {value:7f}")

```

```
158 print("Nombre d'itérations:",k)
159
160 print("\nOptimal Policy:")
161 for i, action in enumerate(optimal_policy):
162   print(f"({states[i]}) = {actions[action]}")
163
164 ###----------------------Analyze of convergence based on gamma's
    ↪  value--------------------###
165
166 Gamma = [1-1e-1, 1-1e-2, 1-1e-3, 1-1e-4, 1-1e-5]
167 Policy_times = []
168 for gam in Gamma:
169   start = time.time()
170   _ , _, k = Policy_iteration(pi,V,P, R,gam,"Approx", theta=0.0001)
171   end = time.time()
172   Policy_times.append(end-start)
173
174 Gamma_log = [np.log10(1/(1-g)) for g in Gamma] # Log(1/(1-gamma))
175 Times_log = [np.log10(t) for t in Policy_times] # Log(times)
176 plt.figure()
177 for i in range(len(Policy_times)):
178
      ↪  plt.text(Gamma_log[i],Times_log[i],f'{Gamma[i],round(Policy_times[i],5)}')
179
180 plt.plot(Gamma_log,Times_log, "-*", label="$(\gamma,t=f(\gamma))$")
181 plt.xlabel("$\log[\dfrac{1}{1-\gamma}]$")
182 plt.ylabel("$\log(t)$")
183 plt.title("Times to compute $V^*$ when $\gamma$ increases")
184 plt.legend()
185 plt.grid(1)
186
187 print(Policy_times)
188
189 ###----------------------Analyze of convergence by varying the number of
    ↪  states--------------------###
190
191 gam = 0.7 # Here gamma is constant
192 actions = 10 # Number of action is constant
193 Times_state = []
194 States = [35, 50, 100,500, 1000, 2000, 3000, 4000, 5000] # List of states
195 Methods = ["Exact", "Approx"]
196 for m in Methods:
197   T = []
198   for state in States:
199     np.random.seed(42)
200     P_s = Markov_decision_matrix(state, actions)
201     R_s = np.round(np.random.rand(state, actions),2)
202     pi_s = np.random.randint(actions, size = state)
203     V_s = np.zeros(state)
204     start = time.time()
205     _ , _, k = Policy_iteration(pi_s,V_s,P_s, R_s,gam,m, theta=0.0001)
206     end = time.time()
207     T.append((end-start))
208   Times_state.append(T)
```

```
209
210 plt.figure()
211 plt.plot(States, Times_state[0], "-*r", label=f"{Methods[0]} Policy
    ↪ iteration")
212 plt.plot(States, Times_state[1], "-+b", label=f"{Methods[1]}imate Policy
    ↪ iteration")
213 plt.xlabel("$S$")
214 plt.ylabel("CPU_time(s)")
215 plt.title("Times to compute $V^*$ when $|S|$ increases")
216 plt.grid(1)
217 plt.legend()
218
219
220
221 ###----------------------Analyze of convergence by varying the number of
    ↪ actions--------------------###
222
223 gam = 0.7 # Here gamma is constant
224 state = 10 # Number of state is constant
225 Times_action = []
226 Actions =[35, 50, 100,500, 1000, 2000, 3000, 4000, 5000] # List of states
227
228 Methods = ["Exact", "Approx"]
229 for m in Methods:
230   T = []
231   for actions in Actions:
232     np.random.seed(42)
233     P_a = Markov_decision_matrix(state, actions)
234     R_a = np.round(np.random.rand(state, actions),2)
235     pi_a = np.random.randint(actions, size = state)
236     V_a = np.zeros(state)
237     start = time.time()
238     _ , _, k = Policy_iteration(pi_a,V_a,P_a, R_a,gam,m, theta=0.0001)
239     end = time.time()
240     T.append((end-start))
241   Times_action.append(T)
242
243 plt.figure()
244 plt.subplot(2,1,1)
245 plt.plot(States, Times_action[0], "-*r", label=f"{Methods[0]} Policy
    ↪ iteration")
246 plt.plot(States, Times_action[1], "-+b", label=f"{Methods[1]}mate Policy
    ↪ iteration")
247 plt.xlabel("$A$")
248 plt.ylabel("CPU_time(s)")
249 plt.suptitle("Times to compute $V^*$ when $|A|$ increases")
250 plt.grid(1)
251 plt.legend()
252
253 """> # Test of Policy iteration methods on Sparse matrix"""
254
255 #----------------------Function to create sparse transition
    ↪ matrix--------------#
256
```

```python
def create_random_sparse_3d_transition_matrix(nb_states, nb_actions, nb_next_states, density=0.3):
    """
    Create a random sparse 3D transition matrix for an MDP.

    Parameters:
    - nb_states: Number of states.
    - nb_actions: Number of actions.
    - nb_next_states: Number of next states.
    - density: The proportion of non-zero elements (default is 0.1).

    Returns:
    - A 3D numpy array representing the transition matrix.
    """
    data = []
    row_indices = []
    col_indices = []

    rng = np.random.default_rng()

    for s in range(nb_states):
        for a in range(nb_actions):
            # Determine how many transitions to generate for this state-action pair
            num_transitions = int(density * nb_next_states)
            if num_transitions < 1:
                num_transitions = 1

            # Randomly choose next states and probabilities
            next_states = rng.choice(nb_next_states, size=num_transitions, replace=False)
            probabilities = rng.random(size=num_transitions)
            probabilities /= probabilities.sum()  # Normalize to sum to 1

            for s_prime, p in zip(next_states, probabilities):
                row_index = s * nb_actions + a
                row_indices.append(row_index)
                col_indices.append(s_prime)
                data.append(p)

    # Convert to numpy arrays
    data = np.array(data)
    row_indices = np.array(row_indices)
    col_indices = np.array(col_indices)

    # Create sparse matrix in COO format
    transition_matrix_2d = sparse.coo_matrix((data, (row_indices, col_indices)),
                                             shape=(nb_states * nb_actions, nb_next_states))
   # Convert to a dense 3D numpy array
    transition_matrix_3d = np.zeros((nb_states, nb_actions, nb_next_states))
```

```
304
305     for row, col, val in zip(row_indices, col_indices, data):
306         s = row // nb_actions
307         a = row % nb_actions
308         transition_matrix_3d[s, a, col] = val
309
310     return transition_matrix_3d
311
312 # Example usage
313 nb_states = 5
314 nb_actions = 2
315 nb_next_states = 5
316 density = 0.3  # 30% of transitions will be non-zero
317
318 transition_matrix_3d =
    ↪  create_random_sparse_3d_transition_matrix(nb_states, nb_actions,
    ↪  nb_next_states, density)
319
320 # Display the 3D transition matrix
321 print("3D Transition Matrix:\n", transition_matrix_3d[:,:,0].shape)
322
323 """## When $|S|$ increases"""
324
325 gam = 0.7 # Here gamma is constant
326 actions = 10 # Number of action is constant
327 Density = 0.3
328 Sp_Times_state = []
329 States = [50,100,500,1000,2000, 3000,4000,5000] # List of states
330 Methods = ["Exact", "Approx"]
331 for m in Methods:
332   T = []
333   for state in States:
334     np.random.seed(42)
335     P_s = create_random_sparse_3d_transition_matrix(state, actions, state,
      ↪  Density) # Sparse transition matrix
336     R_s = np.round(np.random.rand(state, actions),2)
337     pi_s = np.random.randint(actions, size = state)
338     V_s = np.zeros(state)
339     start = time.time()
340     _ , _, k = Policy_iteration(pi_s,V_s,P_s, R_s,gam,m, theta=0.0001)
341     end = time.time()
342     T.append((end-start))
343   Sp_Times_state.append(T)
344
345 plt.figure()
346 plt.plot(States, Sp_Times_state[0], "-*r", label=f"{Methods[0]} Policy
    ↪  iteration")
347 plt.plot(States, Sp_Times_state[1], "-+b", label=f"{Methods[1]}imate
    ↪  Policy iteration")
348 plt.xlabel("$S$")
349 plt.ylabel("CPU_time(s)")
350 plt.title("Times to compute $V^*$ when $|S|$ increases and $P$ is a sparse
    ↪  matrix")
351 plt.grid(1)
```

```
plt.legend()

"""## When $|A|$ increases"""

gam = 0.7 # Here gamma is constant
state = 10 # Number of action is constant
Density = 0.6
ASp_Times_state = []
Action = [50,100,500,1000,2000,3000,4000,5000] # List of states
Methods = ["Exact", "Approx"]
for m in Methods:
  T = []
  for actions in Action:
    np.random.seed(42)
    P_a = create_random_sparse_3d_transition_matrix(state, actions, state, Density) # Sparse transition matrix
    R_a = np.round(np.random.rand(state, actions),2)
    pi_a = np.random.randint(actions, size = state)
    V_a = np.zeros(state)
    start = time.time()
    _ , _, k = Policy_iteration(pi_a,V_a,P_a, R_a,gam,m, theta=0.0001)
    end = time.time()
    T.append((end-start))
  ASp_Times_state.append(T)

plt.figure()
plt.plot(Action, ASp_Times_state[0], "-*r", label=f"{Methods[0]} Policy iteration")
plt.plot(Action, ASp_Times_state[1], "-+b", label=f"{Methods[1]}imate Policy iteration")
plt.xlabel("$A$")
plt.ylabel("CPU_time(s)")
plt.title("Times to compute $V^*$ when $|A|$ increases and $P$ is a sparse matrix")
plt.grid(1)
plt.legend()
```

- Linear Program

```
        # -*- coding: utf-8 -*-

# Please install by taping: !pip install pulp

import numpy as np
import matplotlib.pyplot as plt
import pulp
import time
# Test
P1 = np.array([[[0.07200801, 0.18278161, 0.14073106, 0.11509637, 0.0299957 ,
         0.02999106, 0.01116699, 0.16652855, 0.11556865, 0.13613201],
        [0.00520773, 0.24538041, 0.21060217, 0.05372031, 0.04600045,
         0.04640006, 0.07697116, 0.13275971, 0.10927907, 0.07367894],
        [0.15281528, 0.03483974, 0.07296552, 0.09150188, 0.11390722,
```

```
         0.19610414, 0.04987017, 0.12843427, 0.1479604 , 0.01160137],
        [0.11929704, 0.03348399, 0.01277348, 0.18632243, 0.18961076,
         0.15873628, 0.05981372, 0.01917882, 0.13435547, 0.086428  ],
        [0.03016589, 0.12239978, 0.00850029, 0.22476941, 0.06396626,
         0.16376487, 0.07704997, 0.12855247, 0.135138  , 0.04569305]],

       [[0.16851056, 0.13471549, 0.16328177, 0.15551799, 0.10391301,
         0.16021865, 0.0153797 , 0.03406116, 0.00786035, 0.05654132],
        [0.08316254, 0.05805864, 0.17731912, 0.07633199, 0.06010957,
         0.11611685, 0.03015256, 0.17164043, 0.01595108, 0.21115723],
        [0.16981899, 0.04369819, 0.00121433, 0.17932246, 0.15544009,
         0.16031091, 0.16960471, 0.01628265, 0.07882771, 0.02547996],
        [0.16460084, 0.11886802, 0.06310494, 0.01212109, 0.05930686,
         0.0620151 , 0.13914183, 0.12158739, 0.16919868, 0.09005523],
        [0.02655733, 0.15838451, 0.16894139, 0.12463829, 0.17120245,
         0.1096532 , 0.11607906, 0.09494058, 0.00564462, 0.02395858]],
          [[0.16790231, 0.04428548, 0.02678531, 0.09048039, 0.18220764,
         0.04474641, 0.12425118, 0.14079323, 0.04392975, 0.1346183 ],
        [0.0868652 , 0.14934175, 0.14963081, 0.12654245, 0.02132517,
         0.19728671, 0.07576374, 0.04405305, 0.00963052, 0.13956061],
        [0.1538449 , 0.00376636, 0.11627368, 0.05142717, 0.14649021,
         0.0395909 , 0.1568814 , 0.08781049, 0.21268995, 0.03122492],
        [0.06412921, 0.02133593, 0.17386608, 0.16496227, 0.04849963,
         0.1240939 , 0.1536587 , 0.10439197, 0.09958787, 0.04547442],
        [0.01431886, 0.1379885 , 0.138481  , 0.09736869, 0.05214154,
         0.05370715, 0.11164933, 0.13797227, 0.13643065, 0.119942  ]],

       [[0.19263895, 0.02524585, 0.04849603, 0.26960749, 0.18195654,
         0.00275954, 0.03044612, 0.19908098, 0.00151871, 0.04824979],
        [0.0973444 , 0.12274097, 0.11565676, 0.03978498, 0.12633932,
         0.04208756, 0.05772533, 0.13242624, 0.11524372, 0.15065072],
        [0.12930379, 0.11174424, 0.01841889, 0.07230249, 0.05214568,
         0.04797471, 0.19131917, 0.07729323, 0.17539954, 0.12409826],
        [0.18013797, 0.11391889, 0.1307509 , 0.11162541, 0.04425035,
         0.16373831, 0.06363493, 0.00551103, 0.14629141, 0.0401408 ],
        [0.12822126, 0.13005775, 0.12473177, 0.0504671 , 0.00210734,
         0.1265661 , 0.05837824, 0.13179283, 0.13137907, 0.11629853]],

          [[0.19263895, 0.02524585, 0.04849603, 0.26960749, 0.18195654,
         0.00275954, 0.03044612, 0.19908098, 0.00151871, 0.04824979],
        [0.0973444 , 0.12274097, 0.11565676, 0.03978498, 0.12633932,
         0.04208756, 0.05772533, 0.13242624, 0.11524372, 0.15065072],
        [0.12930379, 0.11174424, 0.01841889, 0.07230249, 0.05214568,
         0.04797471, 0.19131917, 0.07729323, 0.17539954, 0.12409826],
        [0.18013797, 0.11391889, 0.1307509 , 0.11162541, 0.04425035,
         0.16373831, 0.06363493, 0.00551103, 0.14629141, 0.0401408 ],
        [0.12822126, 0.13005775, 0.12473177, 0.0504671 , 0.00210734,
         0.1265661 , 0.05837824, 0.13179283, 0.13137907, 0.11629853]],

       [[0.06042058, 0.07902161, 0.17465227, 0.06503203, 0.03477972,
         0.11425498, 0.19209789, 0.14282452, 0.1169759 , 0.01994051],
        [0.10363778, 0.16683866, 0.02360622, 0.08734618, 0.14785029,
         0.12483043, 0.11745742, 0.11837892, 0.06057956, 0.04947455],
        [0.1099875 , 0.11008973, 0.11783012, 0.12410411, 0.06948848,
```

```
         0.06815317, 0.1084837 , 0.08832634, 0.09539324, 0.10814362],
        [0.21203704, 0.08052479, 0.0894798 , 0.02239049, 0.13777087,
         0.00856297, 0.11092521, 0.12928098, 0.06826629, 0.14076156],
        [0.00848645, 0.01039183, 0.22888188, 0.1002201 , 0.03535355,
         0.14530992, 0.21424441, 0.06005044, 0.17331418, 0.02374723]],

          [[0.00962717, 0.09897969, 0.10070844, 0.11873918, 0.13525485,
         0.18177979, 0.0961754 , 0.06015969, 0.14812571, 0.05045009],
        [0.10902578, 0.01948593, 0.00629627, 0.23908958, 0.20762943,
         0.17285665, 0.1015702 , 0.0430405 , 0.03885372, 0.06215194],
        [0.09584904, 0.12470868, 0.11521524, 0.04885304, 0.16663961,
         0.12877508, 0.09674385, 0.10675528, 0.07322707, 0.04323311],
        [0.1028186 , 0.2188951 , 0.00415739, 0.03352624, 0.01328733,
         0.01176404, 0.24708992, 0.20324345, 0.13695964, 0.02825827],
        [0.11520358, 0.11095176, 0.04058753, 0.10166731, 0.09338424,
         0.14431621, 0.14882567, 0.01061639, 0.08778544, 0.14666187]],

       [[0.10018177, 0.17053963, 0.13115552, 0.03244262, 0.01405127,
         0.12791506, 0.00527879, 0.11663647, 0.18721357, 0.11458531],
        [0.07064585, 0.11707668, 0.08340076, 0.09930075, 0.17134399,
         0.07026961, 0.17493403, 0.16477132, 0.03563345, 0.01262357],
        [0.03099827, 0.00560485, 0.02904968, 0.21008584, 0.02189689,
         0.09811361, 0.25987493, 0.00715821, 0.2505221 , 0.08669562],
        [0.02010703, 0.11855739, 0.10702145, 0.14931138, 0.12508031,
         0.136721  , 0.04799124, 0.03019326, 0.12772524, 0.13729169],
        [0.14971506, 0.06236725, 0.05623061, 0.11735498, 0.05151252,
         0.14068416, 0.12974926, 0.06484254, 0.11349432, 0.11404931]],

          [[0.0208345 , 0.18234617, 0.10207804, 0.16697232, 0.06466083,
         0.18092594, 0.07863188, 0.00218957, 0.18291774, 0.01844299],
        [0.05393382, 0.16047065, 0.16056274, 0.09685679, 0.10672076,
         0.0757449 , 0.0495249 , 0.05551324, 0.11359204, 0.12708017],
        [0.1731693 , 0.17274033, 0.01995265, 0.10816155, 0.01259181,
         0.12021735, 0.09659115, 0.19419806, 0.07676771, 0.02561009],
        [0.03495161, 0.18613685, 0.1511117 , 0.02471752, 0.02055832,
         0.17133863, 0.01778554, 0.20088813, 0.17262754, 0.01988417],
        [0.01468904, 0.17082958, 0.06480231, 0.06417404, 0.14073043,
         0.16400931, 0.17071903, 0.13044204, 0.06514666, 0.01445755]],

       [[0.19799489, 0.1422655 , 0.10807968, 0.23091328, 0.02832995,
         0.12550684, 0.00289259, 0.11940138, 0.01434447, 0.03027142],
        [0.02035325, 0.11243055, 0.12920041, 0.10102808, 0.1666295 ,
         0.06492027, 0.04947976, 0.15042441, 0.03872243, 0.16681134],
        [0.00218722, 0.17453113, 0.00776669, 0.16036252, 0.0949606 ,
         0.17868549, 0.0132798 , 0.0996669 , 0.17442743, 0.09413223],
        [0.10386453, 0.11481374, 0.07500921, 0.10356079, 0.09642463,
         0.14871076, 0.00749965, 0.04636507, 0.15683866, 0.14691298],
        [0.09149112, 0.12451615, 0.05569525, 0.03777276, 0.0931058 ,
         0.07094944, 0.11719205, 0.0156083 , 0.1956483 , 0.19802082]]])

P1.shape

#Reward function
R1 = np.array([
```

```
      [0.7 , 0.54, 0.31, 0.81, 0.68],
       [0.16, 0.91, 0.82, 0.95, 0.73],
       [0.61, 0.42, 0.93, 0.87, 0.05],
       [0.03, 0.38, 0.81, 0.99, 0.15],
       [0.59, 0.38, 0.97, 0.84, 0.84],
       [0.47, 0.41, 0.27, 0.06, 0.86],
       [0.81, 1.  , 1.  , 0.56, 0.77],
       [0.94, 0.85, 0.25, 0.45, 0.13],
       [0.95, 0.61, 0.23, 0.67, 0.62],
       [0.36, 0.11, 0.67, 0.52, 0.77]])

R1.shape

##-----------------------------------Define the MDP components-------------------------------##

num_states = 20
num_actions = 10

###------------------------ Transition probability matrix P[s, a, s']-------------------------###

# Random generation of matrix
def Markov_decision_matrix(num_states, num_actions):
  Markov = []
  for s in range(num_states):
    matrix = np.random.rand(num_actions,num_states)
    # Normalization of each row
    matrix = matrix/matrix.sum(axis=1, keepdims=True)
    Markov.append(matrix)
  return np.array(Markov)

#------------Transition probabilities------------#

np.random.seed(42)
P = Markov_decision_matrix(num_states, num_actions)




##-------------------------Reward function R[s, a]-------------------------------------##

R = np.round(np.random.rand(num_states, num_actions),2)  # random generation of R

# Discount factor
gamma = 0.9

# Coefficient of v in objective function
mu = np.random.randint(1,num_actions, size = num_states)

##-------------------Function to solve MDPs using linear programm------------##
```

```
172
173 def Linear_Programming(P,R,gamma, nb_states,nb_actions,disp = "F", mu =
    ↪  None):
174   states = [f's{i}' for i in range(nb_states)]
175   actions = [f'a{i}' for i in range(nb_actions)]
176
177     # Initializing a linear programming problem
178   lp = pulp.LpProblem("MDP", pulp.LpMinimize)
179
180   # Definition of the value variables
181   V = pulp.LpVariable.dicts("V", states, lowBound=None, cat='Continuous')
182
183   # Objective function: minimize the sum of state values
184   if mu is not None:
185     lp += pulp.lpSum([mu[s] * V[states[s]] for s in range(nb_states)])
186   else:
187     lp += pulp.lpSum([V[s] for s in states]), "Objective"
188
189   # Adding constraints for each state and action
190   for s in range(nb_states):
191       for a in range(nb_actions):
192           lp += V[states[s]] >= R[s,a] + gamma *
              ↪  pulp.lpSum([P[s,a,s_prime] * V[states[s_prime]] for s_prime
              ↪  in range(nb_states)]) #, f"Constraint_{s}_{a}"
193
194   # Solving the linear program
195   cplex_path = "/path/to/CPLEX_StudioXXX/cplex/bin/x86-64_linux/cplex"
196
197   solver1 = pulp.CPLEX_CMD(path=cplex_path)
198
199   solver2 = pulp.PULP_CBC_CMD()
200   #lp.solve(solver1)
201   lp.solve(solver2)
202
203
204
205   if disp=="T":
206
207     # Print the optimal values
208     for s in states:
209         print(f"V({s})= {pulp.value(V[s])}")
210
211     # Deriving the optimal policy
212     optimal_policy = {}
213     for s in range(nb_states):
214         best_action = None
215         best_value = float('-inf')
216         for a in range(nb_actions):
217             value = R[s, a] + gamma * sum(P[s,a,s_prime] *
                ↪  pulp.value(V[states[s_prime]]) for s_prime in
                ↪  range(nb_states))
218             if value > best_value:
219                 best_value = value
220                 best_action = a
```

```
        optimal_policy[states[s]] = best_action

    print("\n Optimal policy:")
    for s in range(nb_states):
        print(f"{s}: a{optimal_policy[states[s]]}")

  return lp,V

pulp.listSolvers(onlyAvailable=False)

#Prog_L,V = Linear_Programming(P,R,gamma, num_states,num_actions,"F")

Pl, _ = Linear_Programming(P1,R1,gamma, 10,5,"T",mu)

Pl

##------------- Impact of gamma on the resolution of Linear Program
 ↪  ----------------##
Gamma = [0.2,0.5,0.7,1-1e-1, 1-1e-2, 1-1e-3,1-1e-4]
LP_times = []

for gam in Gamma:
  start = time.time()
  _ ,_ = Linear_Programming(P,R,gam, num_states,num_actions)
  end = time.time()
  LP_times.append(end-start)

Gamma_log = [np.log10(1/(1-g)) for g in Gamma] # Log(1/(1-gamma))
Times_log = [np.log10(t) for t in LP_times] # Log(times)
plt.figure()
for i in range(len(LP_times)):
  plt.text(Gamma_log[i],Times_log[i],f'{Gamma[i],round(LP_times[i],5)}')

plt.plot(Gamma_log,Times_log, "-*", label="$(\gamma,t=f(\gamma))$ \n
 ↪  Linear Program")
plt.xlabel("$\log[\dfrac{1}{1-\gamma}]$")
plt.ylabel("$\log(t)$")
plt.title("Times to compute $V^*$ when $\gamma$ increases")
plt.legend()
plt.grid(1)

###----------------------Analyze of convergence by varying the number of
 ↪  state--------------------###

LPState_times = []
S = [10,20,30,50,100,150,200,300,400,500] #states
a = 10 # fixed action
gamma = 0.7


for s in S:
  Pr = Markov_decision_matrix(s, a)
  Rr = np.round(np.random.rand(s, a),2)
  start = time.time()
```

```
  _ ,_ = Linear_Programming(Pr,Rr,gamma, s,a)
  end = time.time()
  LPState_times.append(end-start)

plt.figure()

plt.plot(S,LPState_times, "-*", label="$Linear Programing$")
plt.xlabel("$S$")
plt.ylabel("$CPU_{time}(s)$")
plt.title("Times to compute $V^*$ when $S$ increases")
plt.legend()
plt.grid(1)

plt.figure()
Sss = [s**3 for s in S]
plt.plot(Sss,LPState_times, "-*", label="")
plt.xlabel("$S^3$")
plt.ylabel("$CPU_{time}(s)$")
plt.grid(1)
plt.legend()

###----------------------Analyze of convergence by varying the number of
 ↪  action--------------------###

LPAction_times = []
A = [10,20,30,50,100,150,200,300,400,500] #states
s = 10 # fixed action
gamma = 0.7


for a in A:
  Pr = Markov_decision_matrix(s, a)
  Rr = np.round(np.random.rand(s, a),2)
  start = time.time()
  _ ,_ = Linear_Programming(Pr,Rr,gamma, s,a)
  end = time.time()
  LPAction_times.append(end-start)

plt.figure()

plt.plot(A,LPAction_times, "-*", label="$Linear Programing$")
plt.xlabel("$A$")
plt.ylabel("$CPU_{time}(s)$")
plt.title("Times to compute $V^*$ when $A$ increases")
plt.legend()
plt.grid(1)

```