De la programmation dynamique à l'apprentissage par renforcement

Abel Verley
11550

Sommaire

Motivations : Régimes de traitement dynamique

Maladies chroniques : ”Une maladie chronique est une maladie de longue durée, évolutive, souvent associée à une invalidité et à la menace de complications graves” - Ministère de la santé

Exemples

- Diabète
- Asthme
- Parkinson
- ...

Motivations : Régimes de traitement dynamique

Définition

Régimes de traitement dynamique (DTRs) : Séquence d'instructions qui détermine les étapes d'un traitement personnalisé tenant compte de l'historique médical d'un patient

Formalisation : Processus de décision markovien

Définitions

Définition

Processus de décision markovien (MDP) : On appelle processus de décision markovien un quadruplet (S, A, p, r) où :

- S est un ensemble fini d'états
- A est un ensemble fini d'actions
- p est une fonction de transition aléatoire : $p(s'|a, s)$
- r est la fonction de récompense : $r(s, a, s')$

Implémentation :

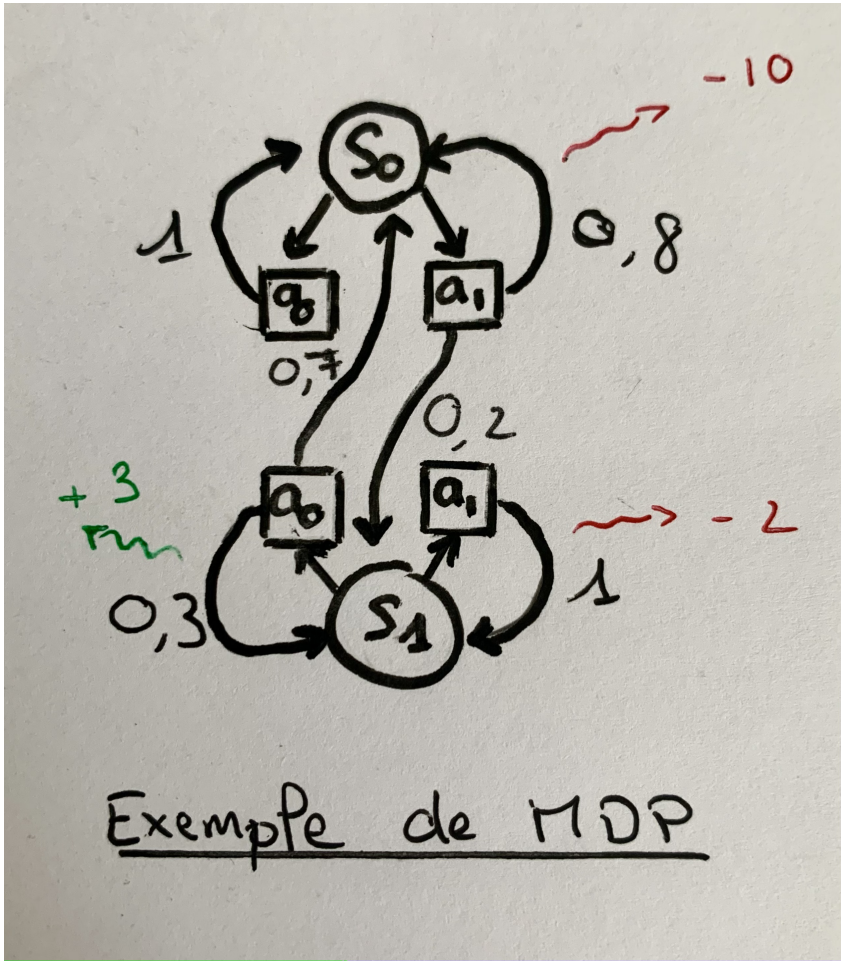
$$\forall a \in A, [P_a]_{s,s'} = p(s'|a,s) :$$

$$P_0 = \begin{pmatrix} 1 & 0 \\ 0,7 & 0,3 \end{pmatrix}$$

$$P_1 = \begin{pmatrix} 0,8 & 0,2 \\ 0 & 1 \end{pmatrix}$$

$$\forall a \in A, [R_a]_{s,s'} = r(s,a,s')$$

$$R_0 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \; R_1 = \begin{pmatrix} -10 & 0 \\ 0 & -2 \end{pmatrix}$$



Formalisation : Processus de décision markovien

Définitions

Définition

Politique : Application $\pi : S \longrightarrow A$. On note $\mathcal{D}$ l'ensemble des politiques markoviennes déterministes

Pour comparer les politiques :

Définition

Fonction de valeur : Soit $\gamma \in]0,1[$, π une politique. On définit la fonction de valeur :

$$V_\gamma^\pi : s \in S \mapsto \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s]$$

Pour comparer les politiques :

Définition

Fonction de valeur : Soit $\gamma \in]0,1[$, π une politique. On définit la fonction de valeur :

$$V_\gamma^\pi : s \in S \mapsto \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s]$$

Remarque :

- γ sert à : faire **converger** la somme

Définition

Politique optimale : Sous réserve d'existence, on note π^* une politique optimale, c'est à dire,

$$\forall \pi \in \mathcal{D}, V^* = V_\gamma^{\pi^*} \geq V_\gamma^\pi$$

L'objectif est de déterminer π^* et V^*.

Propriété

Soit π une politique. Si

$$\forall s \in S,\ \pi(s) \in \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V^*(s'))$$

alors π est optimale
- cf.annexe 1.1

Il suffit donc de trouver V^*

Caractérisation de la fonction de valeur optimale :

Théorème

Equation de Bellman :

Soit $\gamma \in]0,1[$ alors , V^* est l'unique solution de l'équation

$$\forall s \in S,\ V(s) = \max_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V(s'))$$

- cf.annexe 1.2

Caractérisation de la fonction de valeur optimale :

Théorème

Equation de Bellman :

Soit $\gamma \in]0,1[$ alors , V^* est l'unique solution de l'équation

$$\forall s \in S,\ V(s) = \max_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V(s'))$$

- cf.annexe 1.2

Remarque : On en déduit une manière **itérative** d'estimer V^* :

$$V_{n+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V_n(s'))$$

Un algorithme de programmation dynamique

Itération des valeurs

Algorithm 1 Algorithme d'itération des valeurs

$n \leftarrow 0$
$V_n \leftarrow 0$
repeat
  $V_{n+1} \leftarrow V_n$
  for $s \in S$ **do**
    $V_{n+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V_n(s'))$
  end for
  $n \leftarrow n+1$
until $||V_{n+1} - V_n||_\infty < \varepsilon$
for $s \in S$ **do**
  $\pi(s) \in \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s') + \gamma V_n(s'))$
end for
return V, π

- cf.annexe 2.1

Un algorithme de programmation dynamique

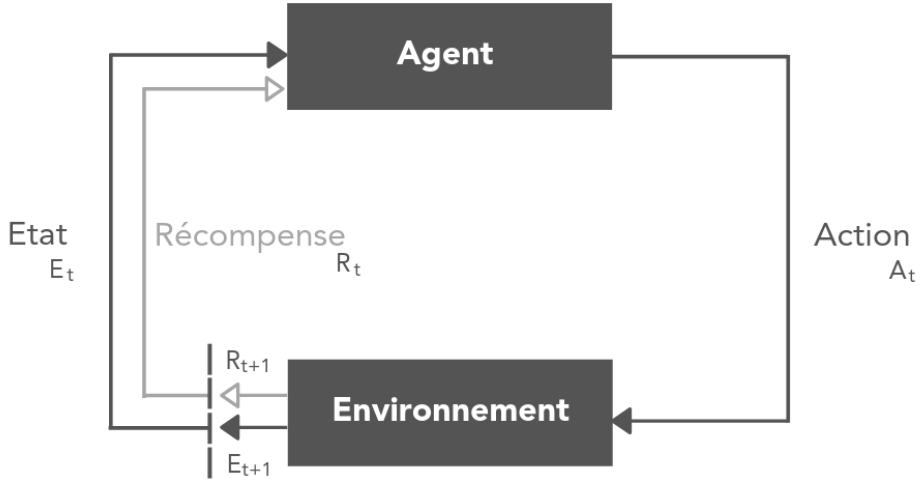
Limites du modèle

Dans un environnement réel, les fonctions de **transition** et de **récompense** sont initialement **inconnues**

Apprentissage par renforcement

Pour parer à ce problème : **l'apprentissage par renforcement**

Un **agent** interagit avec son **environnement** pour améliorer sa connaissance de l'environnement et prendre des meilleures actions
$\rightarrow$ **Essais et erreurs :** Estimer la fonction valeur



source : teahouse.fifty-five.com

Lorsque l'environnement est inconnu, on ne peut déduire π^* de V^*

Définition

Fonction de valeur Q : Pour une politique π, on définit la fonction Q^π telle que :

$$\forall s \in S, a \in A, Q^\pi(s,a) = \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a]$$

Lorsque l'environnement est inconnu, on ne peut déduire π^* de V^*

Définition

Fonction de valeur Q : Pour une politique π, on définit la fonction Q^π telle que :

$$\forall s \in S, a \in A, Q^\pi(s,a) = \mathbb{E}^\pi[\sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a]$$

Remarque : En notant Q^* pour la politique optimale, on a $\forall s \in S$:

- $V^*(s) = \max_{a \in A} Q^*(s,a)$
- $\pi^*(s) \in \operatorname{argmax}_{a \in A} Q^*(s,a)$

L'apprentissage se fait sur une séquence d'épisodes de longueurs finies. On impose :

- Un état initial s_0
- Un état final s_f

1 : Le choix de l'action

- **Exploration :** Découvrir de nouvelles paires de $S \times A$ au risque de générer des pénalités

- **Exploitation :** Choisir la solution optimale d'après la connaissance partielle de l'environnement

1 : Le choix de l'action

- **Exploration :** Découvrir de nouvelles paires de $S \times A$ au risque de générer des pénalités

- **Exploitation :** Choisir la solution optimale d'après la connaissance partielle de l'environnement

$\rightarrow$ Stratégie ε-greedy :

- Avec une probabilité ε : l'action est choisie au hasard
- Avec une probabilité $1 - \varepsilon$: l'action est choisie de manière "gloutonne" : $a \in \text{argmax}_{a \in A} Q(s, a)$

2 : Estimation des fonctions de valeur : L'action a_n prise depuis l'état s_n génère une récompense r_n et fait passer à l'état s_{n+1}
Les fonctions valeur vérifient :

$$V(s_n) = \bar{r}(s_n, a_n, s_{n+1}) + \gamma V(s_{n+1})$$
$$Q(s_n, a_n) = \bar{r}(s_n, a_n, s_{n+1}) + \gamma Q(s_{n+1}, a_{n+1})$$

Or on n'a qu'une approximation :

$$\delta_n^V = r_n + \gamma V(s_{n+1}) - V(s_n) \approx 0$$
$$\delta_n^Q = r_n + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n) \approx 0$$

δ donne une évaluation de l'erreur faite sur l'approximation de V ou Q

2 : Estimation des fonctions de valeur :
Algorithme TD(0) :

$$V(s_n) \leftarrow V(s_n) + \alpha\delta_n^V$$

Algorithme SARSA :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha\delta_n^Q$$

Algorithme Q-learning :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma \max_{a \in A} Q(s_{n+1}, a) - Q(s_n, a_n)]$$

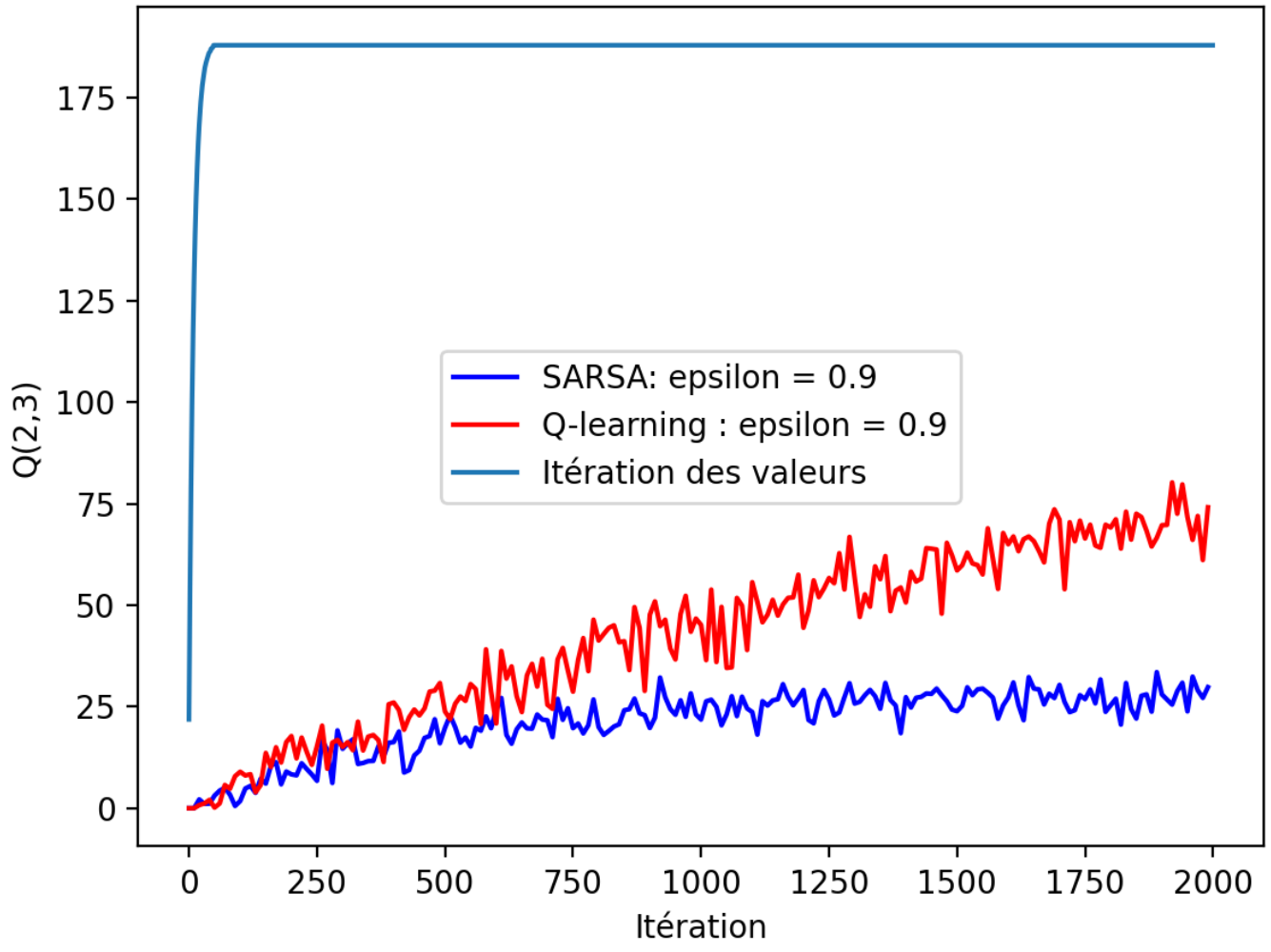
Avec α le **pas d'apprentissage**

Algorithm 2 Algorithme de Q-Learning

for épisodes 1 à N **do**
  $n \leftarrow 0$
  $s_n \leftarrow s_0$
  while $s_n \neq s_f$ **do**
    if random[0,1] $< \varepsilon$ **then**
      a_n est choisi aléatoirement
    else
      $a_n \leftarrow \text{argmax}_{a \in A} Q_n(s, a)$
    end if
    observer r_n et s_{n+1} d'après s_n et a_n
    $Q(s_n, a_n) \leftarrow Q(s_n, , a_n) + \alpha[r_n + \gamma \text{max}_{a \in A} Q(s_{n+1}, a) - Q(s_n, a_n)]$
    $n \leftarrow n + 1$
  end while
end for

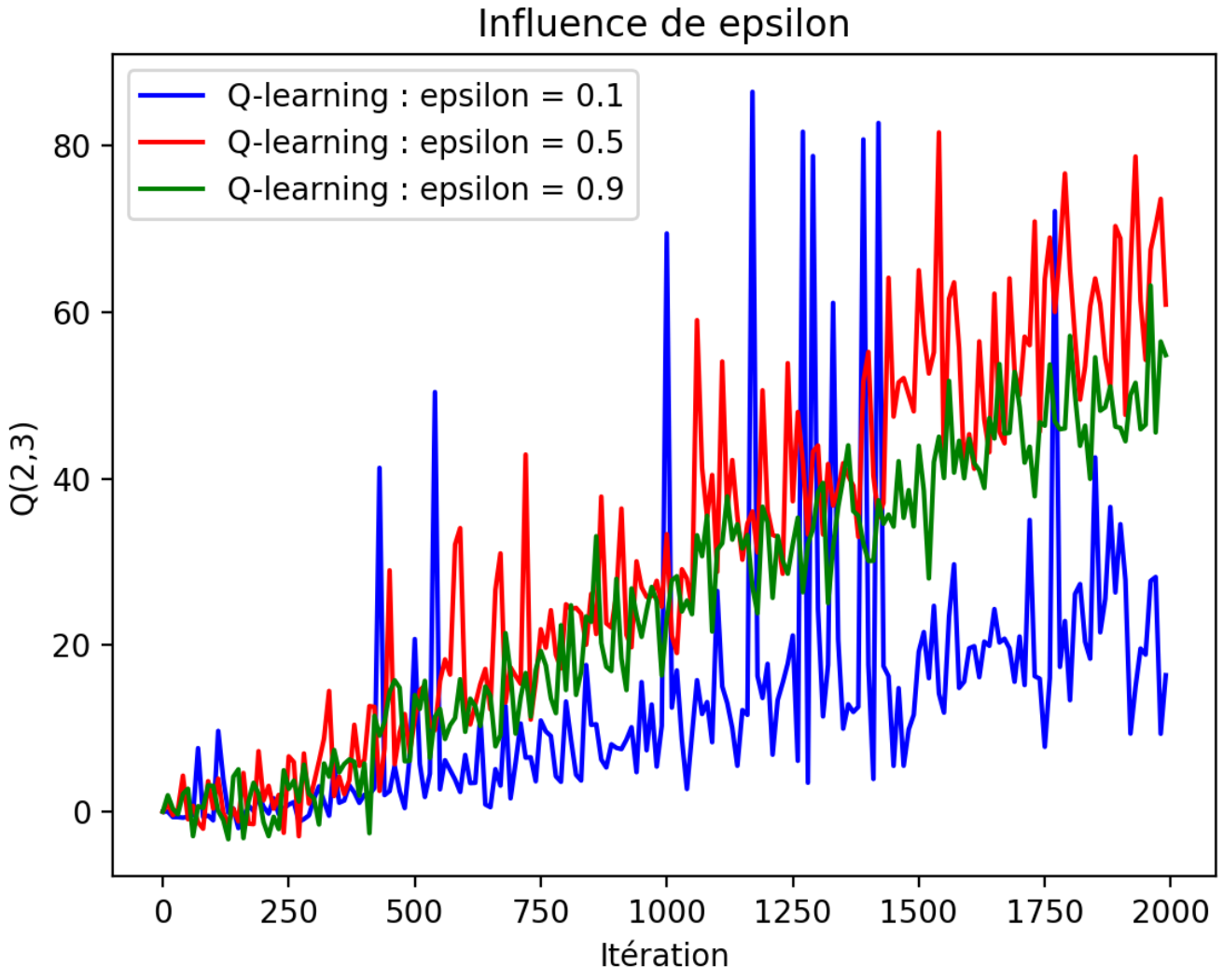
Apprentissage par renforcement

Résultats graphiques



Apprentissage par renforcement

Résultats graphiques



MERCI POUR VOTRE ATTENTION !

annexe 1.1 : Démonstration propriété 1

Quelques résultats pour la démonstration de la propriété :
Définition : Soit π une politique. La matrice $P_\pi \in \mathcal{M}_{|S|}(\mathbb{R})$ telle que $\forall s, s' \in S,\ [P_\pi]_{s,s'} = p(s'|s, \pi(s))$. De même on construit la matrice r_π telle que $[r_\pi]_s = \sum_{s' \in S} p(s'|s, \pi(s)) r(s, \pi(s), s')$
Définition : Soit π une politique, $0 < \gamma < 1$. Dans l'espace vectoriel normé $(E, ||.||_\infty)$ avec $E = \mathbb{R}^S$ on pose L_π l'opérateur tel que $\forall V \in E, L.V = r_\pi + \gamma P_\pi V$
Théorème : Soit π une politique, $0 < \gamma < 1$. V_γ^π est l'unique solution de l'équation $V = L_\pi.V$
Preuve : Soit V une solution de l'équation. On a $(I - \gamma P_\pi)V = r_\pi$. La matrice P_π étant stochastique, on montre que $|||P_\pi||| \leq 1$ et on en déduit par sous-multiplicité de $|||.|||$ que $\sum(\gamma^k P_\pi^k)$ converge absolument. De plus par télescopage, $(I - \gamma P_\pi)(\sum_{k=0}^{+\infty} \gamma^k P_\pi^k) = I$ donc $(I - \gamma P_\pi)^{-1} = \sum_{k=0}^{+\infty} \gamma^k P_\pi^k$. Ainsi $V = \sum_{k=0}^{+\infty} \gamma^k P_\pi^k r_\pi$

annexe 1.1 : Démonstration propriété 1

Par ailleurs, pour $s \in S$

$$\begin{aligned} V_\gamma^\pi(s) &= \sum_{t=0}^{+\infty} \gamma^t \mathbb{E}^\pi[r(s_t, a_t)|s_0 = s] \\ &= \sum_{t=0}^{+\infty} \gamma^t \sum_{s' \in S} \mathbb{P}^\pi(s_t = s'|s_0 = s) \sum_{s'' \in S} p(s''|s', \pi(s'))r(s', \pi(s), s'') \end{aligned}$$

Or P_π s'interprète comme la matrice d'adjacence d'une chaine de Markov, on montre alors que $\mathbb{P}^\pi(s_t = s'|s_0 = s) = P^t_{\pi,s,s'}$ et donc pour $s \in S$

$$\begin{aligned} V_\gamma^\pi(s) &= \sum_{t=0}^{+\infty} \gamma^t \sum_{s' \in S} P^t_{\pi,s,s'} r_\pi(s') \\ &= \sum_{k=0}^{+\infty} \gamma^k P_\pi^k r_\pi(s) \end{aligned}$$

Et donc $V = V_\gamma^\pi$. L'unicité est immédiate.

annexe 1.1 : Démonstration propriété 1

Démonstration de la propriété : Soit π telle que $\pi \in \text{argmax}_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^*$. On a alors

$$\begin{aligned} L_\pi V^* &= r_\pi + \gamma P_\pi V^* \\ &= \max_{\pi' \in \mathcal{D}} r_{\pi'} + \gamma P_{\pi'} V^* \\ &= LV^* \\ &= V^* \end{aligned}$$

cf.annexe 1.2. (opérateur de programmation dynamique et équation de Bellman)
D'après le théorème précédent, $V_\gamma^\pi = V^*$ puis π est optimale.

annexe 1.2 : Démonstration partielle équation de Bellman

On propose la démonstration de l'existence d'une solution. Cette dernière permet de comprendre la convergence des méthodes itératives introduites.
Définition : On définit sur E l'opérateur de programmation dynamique L :
$\forall V \in E, L.V = max_{\pi \in \mathcal{D}}(r_\pi + \gamma P_\pi V)$
On traduit l'équation de Bellman : V^* est l'unique solution de l'équation $V = L.V$
Théorème de point fixe de Banach : Soit E un env de dimension finie, f λ-lipschitzienne avec $0 \leq \lambda < 1$ alors il existe une unique solution u de l'équation $f(x) = x$ et de plus, les suites définies par $u_0 \in E$ et $\forall n \in \mathbb{N},\ u_{n+1} = f(u_n)$ convergent vers u.
Démonstration : On construit une suite (u_n) comme ci-dessus. (u_n) converge si et seulement si $\sum(u_{n+1} - u_n)$ converge. Or
$\forall n \geq 1,\ |u_{n+1} - u_n| = |f(u_n) - f(u_{n-1})| \leq \lambda |u_n - u_{n-1}|$ on montre alors par récurrence que $\forall n \in \mathbb{N}, |u_{n+1} - u_n| \leq \lambda^n |u_1 - u_0|$ qui est le terme général d'une série convergente. Donc $\sum(u_{n+1} - u_n)$ converge absolument ce qui implique, par dimension finie, la convergence de la série et donc de la suite (u_n) vers un vecteur u. Par ailleurs, f est lipschitzienne donc continue et par passage à la limite dans la définition récurrente de la suite : $u = f(u)$
Supposons avoir deux points fixes u, u' alors $|u - u'| = |f(u) - f(u')| \leq \lambda |u - u'|$. Et comme $\lambda < 1$ ceci n'est possible que si $u = u'$. D'où l'unicité.

annexe 1.2 : Démonstration partielle équation de Bellman

Théorème : L'opérateur L est une contraction sur E
Démonstration : Soient $U, V \in E$ et $s \in S$. Posons $a_s^* \in \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|a,s)(r(s,a,s' + \gamma V(s'))$. Par symétrie supposons, $L.V(s) \geq L.U(s)$. Alors

$$
\begin{aligned}
|L.V(s) - L.U(s)| &= L.V(s) - L.U(s) \\
&\leq \sum_{s' \in S} p(s'|a_s^*, s)(r(s, a_s^*, s' + \gamma V(s')) - \sum_{s' \in S} p(s'|a_s^*, s)(r(s, a_s^*, s' + \gamma U(s') \\
&\leq \gamma \sum_{s' \in S} p(s'|a_s^*, s')(V(s') - U(s')) \\
&\leq \gamma \sum_{s' \in S} p(s'|a_s^*, s')||V - U|| \\
&\leq \gamma ||U - V||
\end{aligned}
$$

Ainsi comme E est de dimension finie, le théorème de point fixe de Banach s'applique et on a l'existence et l'unicité d'une solution à l'équation de Bellman, ainsi que la convergence de la méthode itérative.
On admet que cette solution est la valeur optimale, la démonstration impliquant l'introduction de nombreux nouveaux objets et propriétés.

annexe 1.3 : Complément d'analyse algorithmique

Itération des valeurs :
-Terminaison et correction : Découle du théorème de point fixe de Banach
-Complexité : Chaque itération a une complexité $O(|S|^2|A|)$. Si p et r sont codés sur b bits, il est établi qu'il suffit d'un nombre d'itération maximale polynomial en $|S|, |A|, b, \frac{1}{(1-\gamma)\log(1/(1-\gamma))}$ pour avoir une politique optimale en sortie.

Autre algorithme : **itération des politiques** (cf.annexe 2.1)

Q-learning : Terminaison et correction :
Théorème : La convergence de l'algorithme de Q-learning vers Q^* est presque sûre si

- $$\sum_{i=0}^{+\infty} \alpha_n = +\infty$$
- $$\sum_{i=0}^{+\infty} \alpha_n^2 < +\infty$$

Annexe 2.1 : MDP.py

```python
import numpy as np
import matplotlib.pyplot as plt
import random

#Implementation des processus de decision markovien (MDP) stationnaire

class Proba():
    """Les probabilites du MDP sont representees par la liste des |A| matrices P_a """

    def __init__(self, list_proba )->None:
        self.P= list_proba

class Reward ():
    """Les recompenses sont representees par la liste des |A| matrices r_a"""

    def __init__(self,list_rewards)->None:
        self.R = list_rewards


class MDP ():
    """Les ensembles S (resp.A) sont caracterises par les entiers |S| (resp.|A|) """

    def __init__(self,s:int,a:int,p:Proba,r:Reward) -> None:

        assert len(p.P)== a and len(r.R)==a
        self.S = s
        self.A = a
        self.P = p.P
        self.R = r.R


    def value_iteration(self,gamma:float, eps:float)->tuple:
        """Algorithme d'iteration sur les valeurs. Les fonction valeur et les politique sont
        representees par des vecteurs de taille |S|"""

        V= np.array([0]*self.S)
        while True:
            newV = np.copy(V)
            for s in range(self.S):
                newV[s]=max([sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)]) for a in range(self.A)])
            if np.linalg.norm(newV-V)<=eps:
                break
            V = newV
        pi = np.array([0]*self.S)
        for s in range(self.S):
```

Annexe 2.1 : MDP.py

```
46          f = lambda a :  sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)])
47          pi[s]=argmax(f,self.A)
48
49
50      return V, pi
51
52
53
54
55  def graph_Q(self,s:int,a:int,gamma:float, N)->None:
56      """Pour tracer l'evolution de Q(s,a) au fil des itérations """
57
58      V= np.array([0]*self.S)
59      Q = []
60
61      for i in range(N):
62          newV = np.copy(V)
63          for s in range(self.S):
64              newV[s]=max([sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)]) for a in range(self.A)])
65          V = newV
66          Q.append(sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)]))
67
68      T = np.array(range(N))
69      Q = np.array(Q)
70
71
72      plt.plot(T,Q,label = 'Itération des valeurs')
73
74
75
76
77  def policy_iteration(self,gamma:float, eps:float)->tuple:
78      #Algotithme d'iteration sur politiques
79
80      pi = np.array([0]*self.S)
81      V=np.array([0]*self.S)
82
83      while True:
84          newpi = np.copy(pi)
85
86          #prediction:
87
88          while True:
89              newV = np.copy(V)
90              for s in range(self.S):
```

Annexe 2.1 : MDP.py

```
91                  newV[s]= sum([self.P[pi[s]][s,s2]*(self.R[pi[s]][s,s2]+gamma*V[s2])  for s2 in range(self.S)])
92              if np.linalg.norm(newV-V)<=eps:
93                  break
94              V = newV
95
96          #controle:
97
98          for s in range(self.S):
99              f = lambda a :  sum([self.P[a][s,s2]*(self.R[a][s,s2]+gamma*V[s2]) for s2 in range(self.S)])
100             pi[s]=argmax(f,self.A)
101         if np.linalg.norm(newpi-pi)==0:
102             break
103         pi = newpi
104
105     return V , pi
106
107
108
109 def argmax(f, A:int)->int:
110     """Détermine un élément de argmax f"""
111
112     a = 0
113     arg = f(a)
114     for a2 in range(A):
115         arg2 = f(a2)
116         if arg2>arg:
117             a = a2
118             arg = arg2
119     return a
120
121 def make_random_MDP(S:int,A:int,Rmax:float)-> MDP:
122     """Genere un MDP aleatoire"""
123
124     list_probas = []
125     list_rewards = []
126     for a in range (A):
127         Pa = np.random.rand(S, S)
128         Pa = Pa/Pa.sum(axis=1)[:,None]
129         list_probas.append(Pa)
130         Ra=Rmax*(np.random.rand(S,S))
131         for s in range(S):
132             for s2 in range(S):
133                 if random.random()<0.5:
134                     Ra[s][s2]= -Ra[s][s2]
135         list_rewards.append(Ra)
```

Annexe 2.1 : MDP.py

```
135             list_rewards.append(Ra)
136         P = Proba(list_probas)
137         R = Reward(list_rewards)
138         return MDP(S,A,P,R)
139
140
141
142
```

Annexe 2.2 : RL.py

```python
import numpy as np
from MDP_ens import *
import random
import matplotlib.pyplot as plt

class Agent():

    def __init__(self,s0:int, env:MDP)-> None:

        self.state = s0
        self.env = env


    def SARSA (self, s0:int, sf:int, eps:float, gamma:float, N:int, lr = 0.01)->np.ndarray:
        """s0 est l'etat initial, sf est l'état final absorbant, epsilon induit la probabilité d'exploration
        (eps-greedy), N est le nombre d'episodes """


        Q = np.array([[0]*self.env.A]*self.env.S, dtype= np.float32)

        for i in range(N):
            self.state = s0
            if random.random() < eps:
                a = random.randint(0,self.env.A-1)
            else:
                a = 0
                Qa = Q[self.state][0]
                for b in range (self.env.A):
                    Q2 = Q[self.state][b]
                    if Q2>Qa:
                        a = b
                        Qa = Q2

            while self.state != sf:
                s2 = etat_suivant(self.state,a,self.env)
                if random.random() < eps:
                    a2 = random.randint(0,self.env.A-1)
                else:
                    a2 = 0
                    Qa = Q[self.state][0]
                    for b in range (self.env.A):
                        Q2 = Q[self.state][b]
                        if Q2>Qa:
                            a2 = b
                            Qa=Q2
```

Annexe 2.2 : RL.py

```
46                  r = self.env.R[a][self.state,s2]
47
48                  newQ = np.copy(Q)
49                  newQ[self.state,a]=Q[self.state,a]+lr*(r+gamma*Q[s2,a2] -Q[self.state,a])
50
51                  a = a2
52                  self.state = s2
53                  Q = newQ
54
55
56          return Q
57
58
59      def Q_learning (self, s0:int, sf:int, eps:float, gamma:float, N:int, lr= 0.01)->np.ndarray:
60          """s0 est l'etat initial, sf est l'etat final absorbant, epsilon induit la probabilité d'exploration
61          (eps-greedy), N est le nombre d'episodes """
62
63          self.state = s0
64          Q = np.array([[0]*self.env.A]*self.env.S, dtype=np.float32)
65
66          # En commentaire: implementation softmax
67
68          #A = np.array([[0]*self.env.A]*self.env.S)
69          #A stocke les taux d'apprentissage selon la methode uncertainty estimation
70          for i in range(N):
71              self.state = s0
72              while self.state != sf:
73                  if random.random() < eps:
74                      a = random.randint(0,self.env.A-1)
75                  else:
76                      a = 0
77                      Qa = Q[self.state][0]
78                      for a2 in range (self.env.A):
79                          Q2 = Q[self.state][a2]
80                          if Q2>Qa:
81                              a = a2
82                              Qa=Q2
83                  #A[self.state,a]+=1
84                  s2 = etat_suivant(self.state,a,self.env)
85                  r= self.env.R[a][self.state,s2]
86                  newQ = np.copy(Q)
87                  newQ[self.state,a]=Q[self.state,a]+(lr)*(r+gamma*max([Q[s2][a2] for a2 in range(self.env.A)])-Q[self.state][a])
88                  #newQ[self.state,a]=Q[self.state,a]+(1/A[self.state,a])*(r+gamma*max([Q[s2][a2] for a2 in range(self.env.A)])-Q[self.state][a])
89                  self.state = s2
90
```

Annexe 2.2 : RL.py

```
91                Q = newQ
92
93        return Q
94
95
96    def graph_Q(self, s0:int, sf:int, eps:float, gamma:float,s2: int, a2:int, M:int, pas = 10)->None:
97        """Trace Q(a,s) en fonction du nombre d'episodes"""
98
99        #il faut que l'etat final soit associé à une recompense nulle
100       for a in range(self.env.A):
101           for s in range(self.env.S):
102               (self.env.R)[a][s,sf]=0
103
104       Q_sa = []
105       for i in range(0,M,pas):
106           Q = self.Q_learning(s0,sf,eps,gamma,i)
107           Q_sa.append(Q[s2,a2])
108           print(i)
109
110       T = np.array(range(0,M,pas))
111       Q_sa = np.array(Q_sa)
112       plt.plot(T,Q_sa, color ='red', label = 'Q-learning : epsilon = '+ str(eps))
113
114
115   def graph_SARSA(self, s0:int, sf:int, eps:float, gamma:float,s2: int, a2:int, M:int, pas=10)->None:
116       """Trace Q(a,s) en fonction du nombre d'episodes"""
117
118       #il faut que l'état final soit associe à une recompense nulle
119       for a in range(self.env.A):
120           for s in range(self.env.S):
121               (self.env.R)[a][s,sf]=0
122
123       Q_sa = []
124       for i in range(0,M,pas):
125           Q = self.SARSA(s0,sf,eps,gamma,i)
126           Q_sa.append(Q[s2,a2])
127           print(i)
128
129       T = np.array(range(0,M,pas))
130       Q_sa = np.array(Q_sa)
131       plt.plot(T,Q_sa, color = 'blue', label = 'SARSA: epsilon = '+ str(eps))
132
133
134
135
```

Annexe 2.2 : RL.py

```python
def etat_suivant(s:int,a:int, env:MDP)->int:
    """Determine l'etat suivant lorsqu'on prend l'action a depuis l'etat s"""

    p = random.random()
    list_probas = [env.P[a][s,s2] for s2 in range(env.S)]
    for i in range(env.S):
        if p<list_probas[i]:
            return i
        else:
            p-=list_probas[i]


if __name__ == '__main__':

    S = 10
    A = 5
    Rmax = 100
    G = make_random_MDP(S,A,Rmax)
    agent = Agent(0,G)

    si = 0
    sf = 9
    eps = 0.3
    gamma = 0.9
    s = 2
    a = 3
    iter  = 2000


    agent.graph_SARSA(si,sf,eps,gamma,s,a,iter)
    agent.graph_Q(si,sf,eps,gamma,s,a,iter )
    agent.env.graph_Q(s,a,gamma,iter)
    plt.xlabel('Itération')
    plt.ylabel('Q(2,3)')
    plt.legend()

    #agent.graph_Q(si,sf,0.1,gamma,s,a,iter)
    #agent.graph_Q(si,sf,0.5,gamma,s,a,iter)
    #agent.graph_Q(si,sf,0.9,gamma,s,a,iter)
    #plt.xlabel('Itération')
    #plt.ylabel('Q(2,3)')
    #plt.legend()
    #plt.title(label = 'Influence de epsilon')
    plt.show()
```