Institut Polytechnique de Paris
Télécom SudParis

Rapport du

Projet Apprentissage par renforcement

Sujet : Allocation dynamique des ressources dans le cloud

Auteurs :

Klabi Yakine

Azaiez Sabrine

Année Universitaire : 2023 /2024

Table des matières

Table des figures

Liste des tableaux

Introduction générale

L'allocation dynamique des ressources dans le cloud permet une gestion flexible et optimisée des capacités de calcul en adaptant les ressources disponibles aux besoins fluctuants des applications et des utilisateurs. Cependant, cette flexibilité entraîne des coûts importants, tels que les frais d'activation, de désactivation et de maintien en fonctionnement des ressources. Une stratégie d'allocation efficace doit donc non seulement garantir les performances et la disponibilité requises par les utilisateurs finaux, mais aussi optimiser les coûts opérationnels pour l'entreprise en équilibrant judicieusement les ressources actives en fonction de la demande réelle.

Le projet utilise des modèles de décision markoviens (MDP) pour modéliser le système et applique des algorithmes d'apprentissage par renforcement tels que l'algorithme de Bellman, Sarsa et Q-learning pour déterminer la politique de gestion optimale.

Le projet vise à réduire les coûts financiers associés à l'activation, à la désactivation et au maintien en fonctionnement des machines virtuelles virtuelles (VM) tout en gérant la file d'attente des tâches à exécuter.

Pour cela, notre rapport est organisé comme suit :

Le premier chapitre explique le système, initialise les paramètres et présente le code implémenté.

Le deuxième chapitre applique l'algorithme de Bellman, qui permet de calculer la Q valeur, qui est l'estimation de la récompense totale future qu'un agent peut recevoir en prenant une action particulière dans un état donné.

L'algorithme Sarsa sera mis en œuvre dans le troisième chapitre. L'algorithme SARSA est une méthode d'apprentissage par renforcement sur la base de la politique qui met à jour la table des valeurs Q en fonction de l'état actuel (S), de l'action prise (A), de la récompense reçue (R), de l'état suivant (S) et de l'action prochaine (A).

Dans le quatriéme chapitre, nous allons voir les résultats donnés suite à l'implémentation de l'algorithme Q-learning.

Chapitre 1

Description du système

Introduction

Dans ce chapitre, nous allons décrire le système tout en initialisant les paramètres du système et présenter le code implémenté.

1.1 Mise en situation

Le système modélise un environnement de cloud computing où les machines virtuelles (VMs) sont allouées dynamiquement pour traiter des paquets arrivant suivant une distribution de Poisson. Nous avons limité le nombre de machines virtuelles à 5 au maximum. Pour définir la probabilité PA des arrivées des paquets, nous avons choisi la distribution de Poisson parce qu'elle est capable de décrire avec précision des événements indépendants se produisant à un taux moyen constant sur une période donnée. Dans ce contexte, lambda représente le taux moyen d'arrivées des paquets par unité de temps. C'est un paramètre clé qui définit la fréquence à laquelle les demandes ou les tâches arrivent au système cloud pour être traitées par les machines virtuelles. Tandis que la taille maximale du batch (grand nombre de paquets qui peuvent arriver en une seule fois dans votre système) est fixée à 5. La capacité totale du système est de 15 (nombre d'arrivées dans la file + nombre de machines virtuelles). Chaque machine virtuelle a un débit fixé à 1.

1.2 Modélisation de La MDP

- L'état du système est défini par un couple (m, n) où m est le nombre de clients dans le système et n est le nombre des machines virtuelles actives. - La taille de l'espace d'état est définie par (B+1)*(K+1), où B est la capacité du système et K est le nombre de machines virtuelles. - L'action sera choisie pour l'exploration dans les phases d'apprentissage par la méthode d'Epsilon-greedy. Les actions peuvent être : Activation, désactivation ou exécution de la machine virtuelle. - La taille de l'espace d'action est définie par (K+1) où K est le nombre maximum de machines virtuelles. - Le passage d'un état s à un état s' pour une action donnée est modélisé comme suit :

A partir de l'état $s = (m, n)$ et pour une action α_j on a :

$$s = (m, n) \rightarrow s' = (\min\{B, \max\{0, m + i - j * d\}\}, j)$$
avec la proba $P_A(i)$

FIGURE 1.1 – Transition

1.3 La fonction de récompense

Le choix d'une action engendre un coût financier qui est la fonction de récompense. Dans notre cas, on cherche à maximiser la fonction de récompense (c'est-à-dire à minimiser le coût financier). Les coûts associés au coût total sont définis dans notre projet comme suit :

Coût	Fonctionnement (Cf)	Activation (Ca)	Désactivation (Cd)	Maintenance (Ch)
Valeur	1	5	3	2

TABLE 1.1 – Tableau des coûts.

Enfin, le coût final est calculé comme suit :

$$r(s, \alpha_j) = j * C_f + (j - n) * 1_{j>n} * C_a + (n - j) * 1_{j<n} * C_d + m * C_H$$

FIGURE 1.2 – Fonction coût

1.4 Implémentation

1.4.1 Importation des bibliothèques

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson
```

FIGURE 1.3 – Bibliothèques utilisées

1.4.2 Initialisation des paramètres du système

Le code de l'initialisation des paramétres de sytéme est ci-contre :

```
#System parameters :
K = 5  # Maximum number of VMs
B = 15  # System capacity (queue + number of VMs)
d = 1  # Throughput of each VM
b = 5  # Maximum batch size of packet arrivals
lambda_rate = 2  # Average rate of packet arrivals per time slot

PA = [poisson.pmf(k, lambda_rate) for k in range(b + 1)]
PA /= np.sum(PA)
```

FIGURE 1.4 – Initialisation des paramètres

1.4.3 Initialisation des coûts

Les couts associés aux différentes actions :

```
#Initializing Costs
Cf = 1  # Cost of running a VM per slot
Ca = 5  # Cost of activating a VM
Cd = 3  # Cost of deactivating a VM
CH = 2  # Cost of maintaining a client in the queue
```

FIGURE 1.5 – Initialisation des coûts

1.4.4 Implémentation de la MDP

```
#Représentation de la MDP :
state_space_size = (B + 1) * (K + 1)  # Encoding states as integers
action_space_size = K+1
Q = np.zeros((state_space_size, action_space_size)) # initialize Q table

#function allowing the update of the state index
def get_state_index(num_clients, num_active_vms):
    return num_clients * (K + 1) + num_active_vms

#Simulate the state transition and calculate reward
def take_action(state_index, action):
    m, n = divmod(state_index, K + 1)  # Decode the current state (m, n) from state index
    i = np.random.choice(range(len(PA)), p=PA) #Simulate packet arrivals based on PA

    # Calculate the next state based on the given action and packet arrivals
    next_m = min(B, max(0, m + i - action * d))
    next_n = action
    next_state_index = get_state_index(next_m, next_n)

    # Calculate the reward (cost to minimize)
    reward = action * Cf + (action - n) * (action > n) * Ca + (n - action) * (action < n) * Cd + m * CH

    return next_state_index, -reward  # Return negative reward because we aim to minimize cost
```

FIGURE 1.6 – Modélisation de la MDP

Chapitre 2

Implémentation de l'algorithme de Bellman

Introduction

Dans ce chapitre,nous allons implémenter l'algorithme de Bellman permettant de calculer la Q valeur qui représente l'estimation de la récompense totale future qu'un agent peut obtenir en prenant l'action dans un état donné et de donner la politique optimale.

2.1 Premiére expérimentation : Implémentation avec un facteur de discount =0.9

Cette implémentation cherche à résoudre un processus de décision de markov afin de trouver une politique optimale pour la gestion des machines virtuelles dans le cloud.

Description de la fonction "probability transition" :

Tout d'abord, d'aprés l'algorithme de Bellman (dans la figure ci-dessous), il intégre la probabilité de transition dans le calcul de la matrice Q. Cette derniére définit la probabilité qu'un systéme passe d'un état s à un état s', étant donné une action a spécifique tout en utilisant la probabilité PA qui représente la distributions des probabilités des arrivées des paquets. En effet, pour une actions prise, la fonction considére tous les paquets possibles à arriver et calcule les nouvelles valeurs de l'état S (m et n), si elles correspondent bien à l'état cible S' alors la probabilité PA est ajoutée à la probabilité totale de transition vers S'.

Description de la boucle de la mise à jour de la matrice Q :

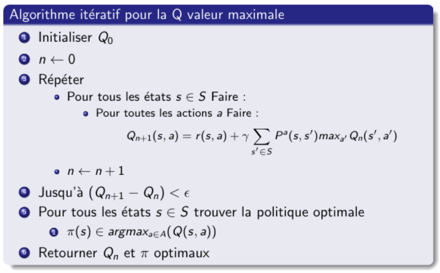
- Dans cette partie, nous avons implémenté l'algorithme comme décrit dans la figure 2.1. Pour ce faire, nous avons utilisé la fonction "take action" définit dans le chapitre 1 pour calculer la récompense qui va etre utilisée pour la mise à jour de Q. Bien évidemment, nous avons utilisé aussi la fonction "probabaility transition".

- Nous utilisons gamma (facteur de discount permettant de pondérer la valeur des récompenses futures par rapport aux récompenses immédiates). Dans notre cas, gamma est 0.9 car nous donnons plus de poids aux récompenses futures.

- Enfin, epsilon est défini comme seuil de convergence et la politique optimale est affiché

comme indiqué dans l'algorithme de Bellman.

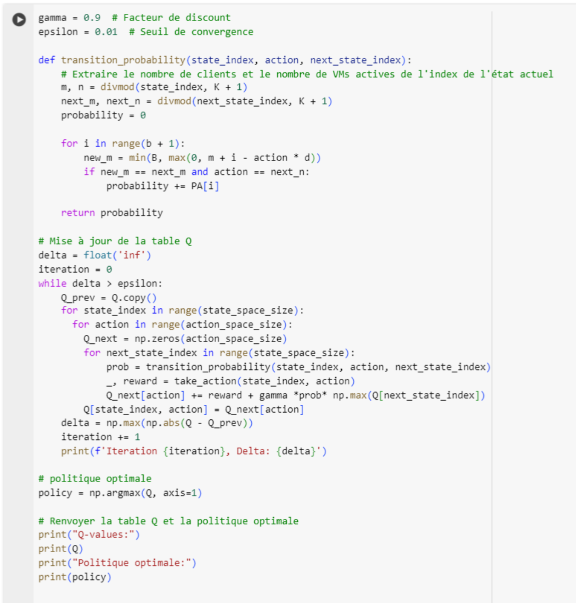
2.1.1 Code de la premiére expérimentation


Algorithme itératif pour la Q valeur maximale

1. Initialiser Q_0
2. $n \leftarrow 0$
3. Répéter
   - Pour tous les états $s \in S$ Faire :
     - Pour toutes les actions a Faire :

       $$Q_{n+1}(s,a) = r(s,a) + \gamma \sum_{s' \in S} P^a(s,s') max_{a'} Q_n(s',a')$$
   - $n \leftarrow n+1$
4. Jusqu'à $(Q_{n+1} - Q_n) < \epsilon$
5. Pour tous les états $s \in S$ trouver la politique optimale
   1. $\pi(s) \in argmax_{a \in A}(Q(s,a))$
6. Retourner Q_n et π optimaux

FIGURE 2.1 – Algorithme de Bellman


```
gamma = 0.9  # Facteur de discount
epsilon = 0.01  # Seuil de convergence

def transition_probability(state_index, action, next_state_index):
    # Extraire le nombre de clients et le nombre de VMs actives de l'index de l'état actuel
    m, n = divmod(state_index, K + 1)
    next_m, next_n = divmod(next_state_index, K + 1)
    probability = 0

    for i in range(b + 1):
        new_m = min(B, max(0, m + i - action * d))
        if new_m == next_m and action == next_n:
            probability += PA[i]

    return probability

# Mise à jour de la table Q
delta = float('inf')
iteration = 0
while delta > epsilon:
    Q_prev = Q.copy()
    for state_index in range(state_space_size):
      for action in range(action_space_size):
        Q_next = np.zeros(action_space_size)
        for next_state_index in range(state_space_size):
            prob = transition_probability(state_index, action, next_state_index)
            _, reward = take_action(state_index, action)
            Q_next[action] += reward + gamma *prob* np.max(Q[next_state_index])
        Q[state_index, action] = Q_next[action]
    delta = np.max(np.abs(Q - Q_prev))
    iteration += 1
    print(f'Iteration {iteration}, Delta: {delta}')

# politique optimale
policy = np.argmax(Q, axis=1)

# Renvoyer la table Q et la politique optimale
print("Q-values:")
print(Q)
print("Politique optimale:")
print(policy)
```

FIGURE 2.2 – Bellman avec un facteur de discount = 0.9

2.1.2 Résultats obtenus de la premiére expérimentation

Politique optimale et variation des q-valeurs minimales :

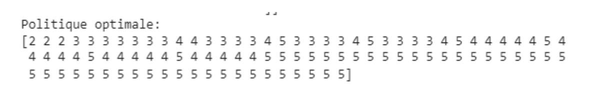

```
Politique optimale:
[2 2 2 3 3 3 3 3 3 3 4 4 3 3 3 3 4 5 3 3 3 3 4 5 3 3 3 3 4 5 4 4 4 4 4 5 4
 4 4 4 4 5 4 4 4 4 4 5 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
```

FIGURE 2.3 – La Politique optimale obtenue

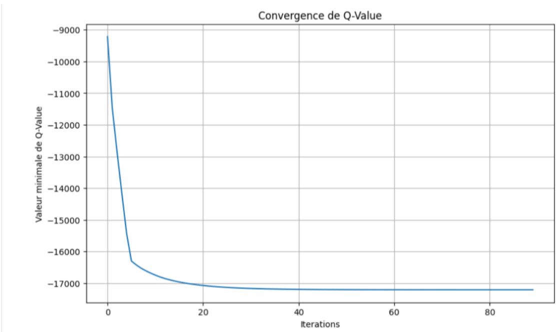


FIGURE 2.4 – Visualisation des q-valeurs min

Interprétation :

Tout d'abord, d'aprés la condition de convergence mise en oeuvre, le systéme a convergé aprés 90 itérations.

- La politique optimale (dans la figure 2.3 ci-dessous) donne un vecteur d'actions où chaque action correspond au nombre de machines virtuelles actives.On remarque que pour les premiers

états, il est préférable de maintenir moins de machines virtuelles actives,tandis que le nombre de machines virtuelles actives augmente dans les états suivants. Un scénario possible est que dans les états initiaux, il y a moins de clients, ce qui signifie que moins de VMs sont nécessaires pour gérer la charge, mais que lorsque le nombre de clients augmente, plus de VMs actives sont nécessaires pour gérer la charge.

Les valeurs q minimales sont représentées dans la courbe obtenue (figure 2.4 ci-dessus) au fil des itérations jusqu'à la convergence. La forme de la courbe indique qu'elle descend très rapidement au début, puis se stabilise et présente peu de changement, suggérant que les valeurs Q n'évoluent plus beaucoup et que la convergence est presque atteinte. Il est noté que même avec la meilleure politique trouvée par l'algorithme, le coût associé reste très élevé. En effet, ces valeurs peuvent etre du à la facon avec laquelle Q est mise à jour, qui, en plus d'utiliser un gamma=0.9 (ce qui accorde une grande importance aux récompenses futures), ne prend pas en compte le taux d'apprentissage, ce qui signifie que la nouvelle information remplace complètement l'ancienne valeur Q. Ainsi, elle ne donne pas importance quant à la valeur actuelle et à l'état actuel.

2.2 Deuxiéme expérimentation : Implémentation avec un facteur de discount =0.5

Le code utilisé dans cette expérience est identique à celui de la première expérience, sauf que la valeur du facteur de discount a été modifiée de 0,9 à 0,5.

2.2.1 Résultats obtenus de la deuxiéme expérimentation

Politique optimale et variation des q-valeurs minimales :

```
Politique optimale:                --
[0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0
 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1
 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5]
```

FIGURE 2.5 – La Politique optimale obtenue

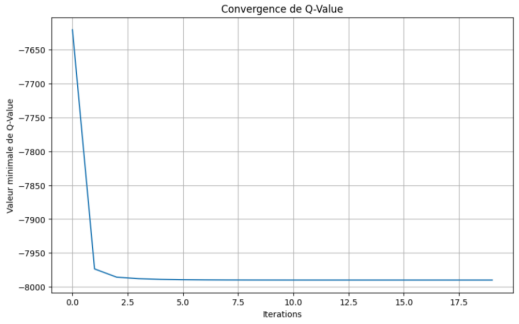


FIGURE 2.6 – Visualisation des q-valeurs min

Interprétation :

Les valeurs minimales de q ont été améliorées par rapport à la première expérience. Cela peut être dû au fait que nous avons modifié Gamma, ce qui a conduit à des couts plus minimes. En effet, cela diminue l'impact des récompenses futures sur la prise de décision actuelle. Cependant, les dépenses demeurent assez élevées, ce qui nécessite des ajustements supplémentaires.

Conclusion

Bien que la méthode discutée dans ce chapitre ait convergé, elle n'a pas produit des résultats satisfaisants. Il est possible de faire des modifications telles que l'application de la méthode d'epsilon greedy, qui permet de concilier l'exploration et l'exploitation, ainsi que l'ajout d'un taux d'apprentissage. Par conséquent, nous mettrons en œuvre d'autres algorithmes dans les sections à venir.

Chapitre 3

Implémentation de l'algorithme Sarsa

Introduction

Dans ce chapitre, nous allons implémenter l'algorithme Sarsa (State-Action-Reward-State-Action). L'algorithme SARSA est une méthode d'apprentissage par renforcement on-policy qui met à jour la table des valeurs Q basée sur l'état actuel (S), l'action prise (A), la récompense reçue (R), le nouvel état suivant (S'), et la prochaine action (A') choisie selon la même politique.

3.1 Stratégie de choix d'action

En apprentissage par renforcement et particulièrement dans notre cas, nous exigeons que l'agent soit capable d'apprendre avec succès la politique optimale qui lui permet de choisir des actions tout en maximisant la récompense accumulée. Nous avons donc convergé vers la mise en œuvre de **Epsilon-greed** basée sur la stratégie axée sur l'exploration et exploitation. Il choisit une action de probabilité epsilon (exploration) et une action la plus optimale selon les valeurs Q du tableau de probabilité 1epsilon (exploitation). Cette approche permet à l'agent d'explorer l'environnement pour découvrir d'alternatives tout en exploitant moins d'acquisition de connaissances pour maximiser moins de récompense. Dans notre cas, epsilon est égale à 0.1 ce qui permet d'établir un équilibre entre l'exploration (où l'agent utilise 10 pourcents du temps pour sélectionner une action au hasard) et l'exploitation (où l'agent utilise 90 pourcents du temps pour choisir l'action avec la plus haute valeur Q estimée) Le code qui permet de définir cette stratégie est :

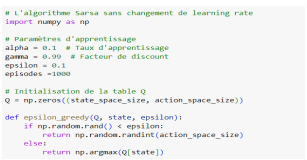


FIGURE 3.1 – La fonction Epsilon-greed

3.2 Implémentation de Sarsa avec un taux d'apprentissage fixe

L'algorithme Sarsa nécessite une condition de l'état s terminal. D'où la fonction "termination condition" la définit bien en prenant comme conditions : quand tous les clients ont été servis ou quand le systéme atteint sa capacité maximale.

3.2.1 Implémentation

Implémentation avec un taux d'apprentissage = 0.9

- Dans cette version de code, nous avons expérimenté deux probabilités différentes en modifiant la valeur du taux d'arrivée des paquets (lambda). De cette façon, nous pouvons conclure sur l'effet de charge sur le système.

- Nous avons pris comme paramétres d'apprentissage : Alpha = 0.9 (Taux d'apprentissage) | Gamma = 0.9 (Facteur de discount : choisi proche 1 pour donner plus de poids aux récompenses futures) | Epsilon = 0.1 | Nombre d'épisodes = 1000

```python
def termination_condition (state_index):
    m, n = divmod(state_index, K + 1)  # Convertir state_index en nombre de clients (m) et VMs actives (n)
    return m == 0 or m == B

# Définir les configurations des probabilités d'arrivée
configurations_lambda = [2, 4]  # Exemple : lambda_rate = 2 pour une charge plus légère, lambda_rate = 4 pour une charge plus lourde


for lambda_rate in configurations_lambda:
    PA = [poisson.pmf(k, lambda_rate) for k in range(b + 1)]
    PA /= np.sum(PA)

    Q = np.zeros((state_space_size, action_space_size))

    min_q_values_per_episode = []

    # Boucle d'apprentissage SARSA
    for episode in range(episodes):
        # Initialiser l'état S
        state_index = np.random.randint(state_space_size)
        # Sélectionner l'action A avec la politique epsilon-greedy
        action = epsilon_greedy(Q, state_index, epsilon)

        while True:
            # Exécuter l'action A, observer R, S'
            next_state_index, reward = take_action(state_index, action)

            # Sélectionner l'action A' depuis S' en utilisant la politique epsilon-greedy
            next_action = epsilon_greedy(Q, next_state_index, epsilon)
            # Mettre à jour Q(S,A)
            Q[state_index, action] += alpha * (reward + gamma * Q[next_state_index, next_action] - Q[state_index, action])
            state_index, action = next_state_index, next_action
            # Terminer si S est un état terminal
            if termination_condition(state_index):
                break
        min_q_values_per_episode.append(np.min(Q))


  # Visualisation de la convergence
    plt.figure(figsize=(10, 6))
    plt.plot(min_q_values_per_episode)
    plt.title('Convergence de Q-Value avec ajustement d\'Alpha pour Lambda rate ='+str(lambda_rate))
    plt.xlabel('Épisodes')
    plt.ylabel('Valeur minimale de Q-Value')
    plt.grid(True)
    plt.show()
```

FIGURE 3.2 – Code avec un taux d'apprentissage 0.9

Implémentation avec un taux d'apprentissage = 0.1 :

Nous avons utilisé le meme code avec changement de la valeur alpha=0.9 à alpha = 0.1.

3.2.2 Résultats

Avec un taux d'apprentissage = 0.9

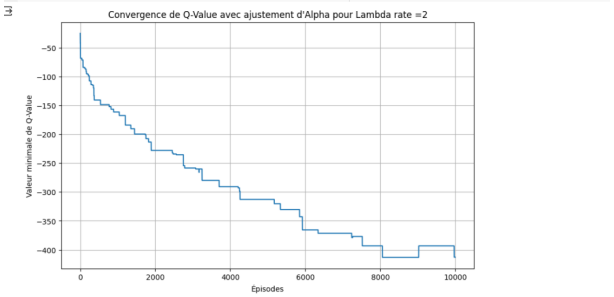


FIGURE 3.3 – Sarsa avec taux d'apprentissage 0.9 et lambda = 2

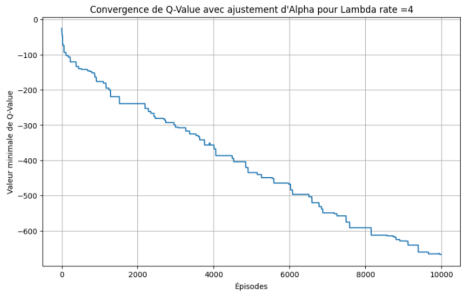


FIGURE 3.4 – Sarsa avec taux d'apprentissage 0.9 et lambda = 4

Avec un taux d'apprentissage = 0.1

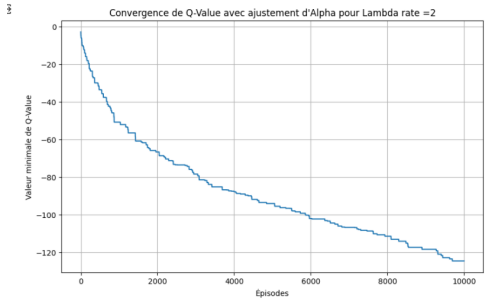


FIGURE 3.5 – Sarsa avec taux d'apprentissage 0.1 et lambda = 2

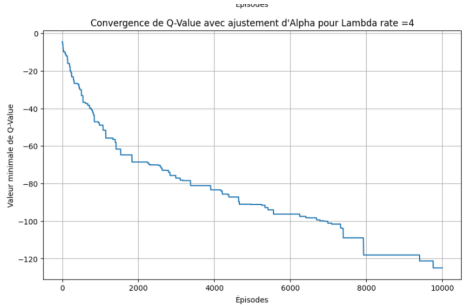


FIGURE 3.6 – Sarsa avec taux d'apprentissage 0.1 et lambda = 4

Interprétations :

- Examinons d'abord l'impact de l'ajout de la focntion epsilon-greedy sur le systéme. Les valeurs ont donc été considérablement améliorées par rapport aux valeurs obtenues dans le chapitre précédent. Cela est possible grâce à la stratégie d'exploration-exploitation, qui permet de combiner l'exploration de nouvelles actions avec l'exploitation d'actions connues pour maximiser les récompenses à court terme.

- En ce qui concerne la convergence, Sarsa est plus lente. Cependant, il tente finalement de converger et les valeurs tendent à se stabiliser mais n'atteint pas la convergence.

- En comparant les résultats donnés pour deux probabilités différentes, on remarque que les résultats avec un lambda=2 (taux moyen d'arrivées de paquets par unité de temps) sont plus améliorées qu'avec une valeur de lambda=4 (charge plus lourde). Ceci est attendu, vu que dans le deuxiéme cas le systéme est concu pour gérer des pics de charge plus fréquents et plus intenses. Ceci peut nécessiter plus de machines virtuelles et donc ceci va induire des couts plus élevés.

-Maintenant, nous comparons les résultats avec les deux taux d'apprentissage. En fait, une valeur alpha = 0,9 indique que l'algorithme s'adapte rapidement aux nouvelles informations,

c'est-à-dire que l'agent apprend rapidement de nouvelles stratégies. Cela peut également entraîner un surajustement sur des expériences récentes. La convergence plus fiable vers les Q optimales est suggérée par une valeur alpha = 0,1 et l'agent peut explorer plus en profondeur l'espace des politiques avant de converger. Les résultats des courbes soulignent donc cela et montrent que les q-valeurs avec un faible taux d'apprentissage ont changé de (-400 avec alpha = 0.9) à (-100 avec alpha = 0.1).

3.3 Implémentation de Sarsa avec un taux d'apprentissage variable

Nous avons utilisé l'ajustement dynamique du taux d'apprentissage. Au début, alpha est fixé à une valeur initiale pour permettre un apprentissage rapide. Au fil des épisodes, alpha est réduit exponentiellement selon un taux de décroissance bien défini pour stabiliser l'apprentissage à mesure que l'agent acquiert plus de connaissances sur l'environnement.

3.3.1 Implémentation

Nous avons implémenté le code en utilisant la fonction epsilon greedy, lambda = 2 (taux de paquets) et nous avons varié le taux d'apprentissage avec la méthode de décroissance exponentielle. Cette approche, permet au début un apprentissage relativement rapide, puis elle devient plus fine à mesure que l'algorithme se rapproche de la politique optimale. Le code soulignant ceci est donné ci-dessous :

```python
# L'algorithme Sarsa avec changement de learning rate
import numpy as np

# Paramètres d'apprentissage
alpha = 0.1# Taux d'apprentissage
alpha_min=0.01
alpha_decay =0.995
gamma = 0.9  # Facteur de discount
epsilon = 0.1
episodes =20000

lambda_rate = 2
PA = [poisson.pmf(k, lambda_rate) for k in range(b + 1)]
PA /= np.sum(PA)
# Initialisation de la table Q
Q = np.zeros((state_space_size, action_space_size))

def termination_condition (state_index):
    m, n = divmod(state_index, K + 1)  # Convertir state_index en nombre de clients (m) et VMs actives (n)
    return m == 0 or m == B


def epsilon_greedy(Q, state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(action_space_size)
    else:
        return np.argmax(Q[state])

min_q_values_per_episode = []
# Boucle d'apprentissage SARSA
for episode in range(episodes):
    # Initialiser l'état S
    state_index = np.random.randint(state_space_size)
    # Sélectionner l'action A avec la politique epsilon-greedy
    action = epsilon_greedy(Q, state_index, epsilon)

    while not termination_condition(state_index):
        # Exécuter l'action A, observer R, S'
        next_state_index, reward = take_action(state_index, action)

        # Sélectionner l'action A' depuis S' en utilisant la politique epsilon-greedy
        next_action = epsilon_greedy(Q, next_state_index, epsilon)

        alpha = max(alpha_min, alpha* (alpha_decay ** episode))  # Décroissance d'alpha

        # Mettre à jour Q(S,A)
        Q[state_index, action] += alpha * (reward + gamma * Q[next_state_index, next_action] - Q[state_index, action])

        state_index, action = next_state_index, next_action
    min_q_values_per_episode.append(np.min(Q))


plt.figure(figsize=(10, 6))
plt.plot(min_q_values_per_episode, label='Valeur Q minimale')
plt.xlabel('Épisode')
plt.ylabel('Valeur Q minimale')
plt.title('Évolution de la valeur Q minimale au fil des épisodes')
plt.legend()
plt.show()
```

FIGURE 3.7 – Code avec un taux d'apprentissage variable

3.3.2 Résultats obtenus

Visualisation :

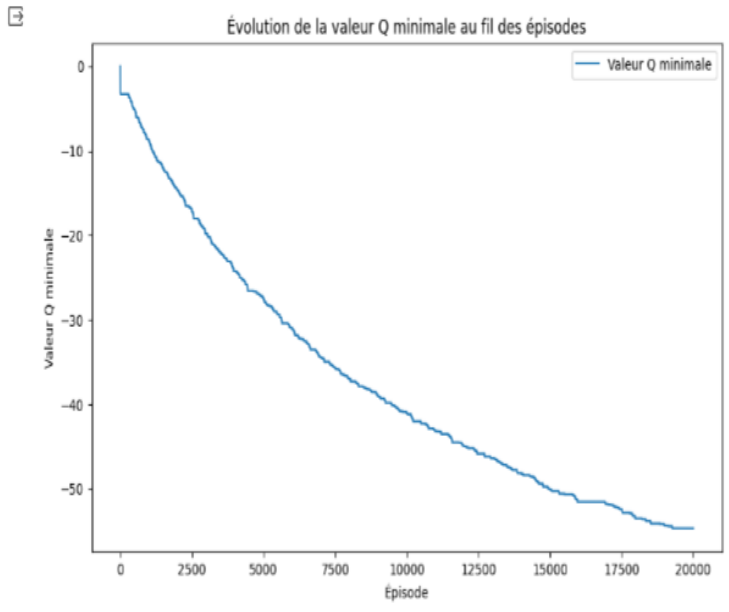


FIGURE 3.8 – Représentation des Q-valeurs minimales au fil des épisodes

Interprétations :

Nous avons constaté que la stratégie de la décroissance exponentielle du taux d'apprentissage a entraîné une amélioration significative des q-valeurs. Par conséquent, elle vise à réduire les dépenses et à converger vers des politiques plus efficaces. En effet, une valeur de (-55) a été obtenue à la fin, ce qui indique une amélioration par rapport à la situation où alpha est fixe.

Conclusion

Bien que les résultats soient encourageants, ils peuvent être améliorés.

Chapitre 4

Implémentation de l'algorithme Q-learning

Introduction

Dans ce chapitre, nous allons implémenter l'algorithme Q-learning. En effet, Le Q-learning est une puissante technique d'apprentissage par renforcement hors politique qui permet à un agent d'apprendre l'action optimale à entreprendre dans chaque état afin de maximiser les récompenses futures. Avec Q-learning, l'agent apprend un tableau de valeurs Q, qui estime la récompense future maximale pour chaque paire état-action.

4.1 Stratégie de choix d'action

Pour le choix d'action, nous avons utilisé la fonction "epsilon-greedy". La troixiéme section définit bien cette fonction.

4.2 Implémentation de Q-learning

4.2.1 Code de mise en œuvre avec deux lambdas distincts

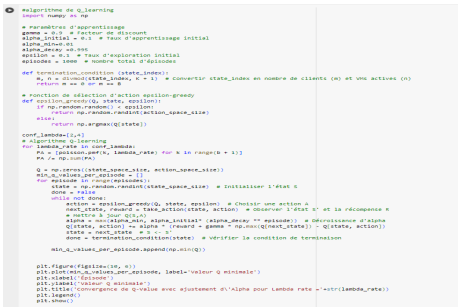


FIGURE 4.1 – Code d'implémentation de q-learning

4.2.2 Résultats obtenus avec deux probabilités distinctes

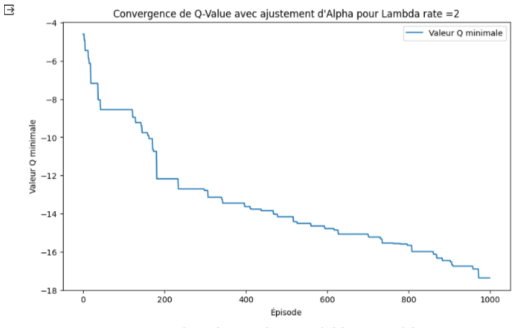


FIGURE 4.2 – Visualisation des q-valeurs minimales avec lambda = 2

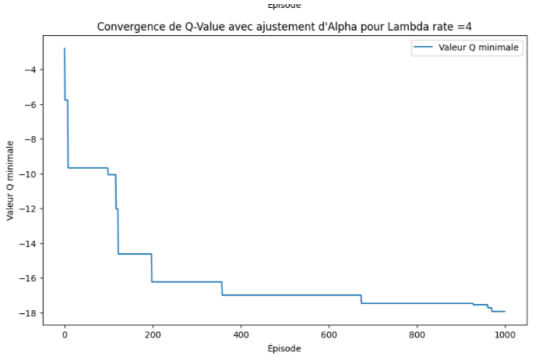


FIGURE 4.3 – Visualisation des q-valeurs minimales avec lambda = 4

Interprétation :

- La première courbe montre une baisse graduelle, indiquant que l'algorithme apprend continuellement tout au long des épisodes. La valeur de Q s'améliore au fil des épisodes, le systéme essaie de converger, mais mille itérations n'étaient pas suffisantes.

- Dans les premiers épisodes, la valeur de Q diminue rapidement sur la deuxième courbe et se stabilise ensuite. Il y a moins de variances dans la Q minimale, ce qui peut indiquer que l'algorithme a trouvé la politique la plus appropriée.

- Les valeurs minimales q ont considérablement augmenté par rapport au chapitre précédent. Elles atteignent une valeur de (-18) qui indique une diminution évidente des coûts par rapport au précédent exemple. L'algorithme d'apprentissage Q-learning (off-policy) a une nouvelle stratégie par rapport à celui de Sarsa (on-policy). Cette théorie repose sur le fait que la mise à jour des q-valeurs utilise le maximum des q-valeurs estimées pour toutes les actions possibles dans le nouvel état, sans tenir compte de l'action à venir. Par conséquent, l'algorithme évalue la meilleure action future en supposant une politique optimale. Cet aspect justifie sa performance, car il apprend la meilleure politique possible sans se limiter à la politique suivie pendant l'exploration.

Conclusion

La mise en place de l'algorithme Q-learning a permis de donner des résultats de minimisation des couts plus performants grace à l'approche hors politique sur laquelle il repose.

Conclusion et perspectives

Le rapport présenté souligne des techniques de minimisation des couts financiers par l'apprentissage par renforcement lors de la l'allocation dynamique des machines virtuelles dans le cloud.

Tout d'abord, nous avons commencé par appliquer l'algorithme de Bellman. Cet algorithme tient compte de la valeur maximale q des états futurs en fonction de la probabilité de transition d'un état s à un état s' suite à une action a spécifique et tient compte aussi de la récompense obtenue lors de la prise de l'action a. Malgré sa convergence rapide, cet alorithme n'a pas réussi à réduire les coûts car il nécessite des modifications dans la façon de prendre des décisions notamment en ce qui concerne l'équilibre entre l'exploration et l'exploitation.

En effet, cet algorithme donne une version récursive juste pour calculer les q-valeurs et pour donner une politique de prise de décision. Mais il n'est pas un vrai algorithme d'apprentissage. Donc, il est nécessaire de l'adapter et résoudre l'équation de Bellman (de la mise à jour de Q) en l'améliorant afin d'obtenir des résultats plus performants.

Converger vers l'utilisation de Sarsa : nous avons ajouté la fonction epsilon greedy à l'algorithme de Sarsa et intégré le taux d'apprentissage utilisé dans le calcul de Q. De plus, la mise à jour de Q de l'action et de l'état actuels dépend non seulement de la récompense recue et de la Q estimée du nouvel état, mais aussi de l'action choisie dans ce nouvel état, ce qui a amélioré les performances.

Enfin, nous avons implémenté Q-learning, qui repose sur une autre approche en évaluant la meilleure action future en supposant une politique optimale future sans tenir compte de l'action prise.

Comme prespectives, des améliorations peuvent etre faites au niveau de choix de taux d'apprentissage (utiliser une autre méthode pour la variation de alpha),l'intégration de l' optimiseur Adam ...