Projet Cassiopée 2019 - Allocation de ressources par Machine Learning

Nessim OUSSEDIK Gabriel GUEZ Lucas BLANCHARD
Encadrant : Andrea ARALDO

7 juin 2019

Remerciements

Nous tenons particulièrement à remercier Andrea Araldo pour nous avoir si bien accompagnés durant la mise en oeuvre de notre projet Cassiopée. Sa bienveillance, sa gentillesse ainsi que ses précieux conseils ont fait de notre travail une très agréable expérience. Merci pour tout !

Nous souhaitons également adresser de chaleureux remerciements à l'endroit de Télécom SudParis ainsi que de toutes les équipes ayant rendu possible la formidable expérience qu'est Cassiopée, au premier rang desquelles Joséphine Kohlenberg. Nous saluons sa bienveillance ainsi que sa forte implication dans la bonne gestion des projets Cassiopée. Merci beaucoup !

Table des matières

Introduction

Ce document, produit avec LaTeX, présente notre projet Cassiopée, dont le titre est "Allocation de ressources par Machine Learning".

L'objectif de ce projet est d'appliquer de façon critique des techniques de machine learning pour les problèmes d'allocation des ressources, c'est à dire la distribution d'une ressource limitée parmi plusieurs agents. Nous étudions ici le cas d'un opérateur de réseau qui a installé un espace de stockage cache dans son réseau d'accès et le répartit parmi plusieurs fournisseurs de contenus, l'objectif étant de réduire le trafic vers son réseau.

Une illustration de ces caractéristiques est présentée à la figure 1 ci-dessous.

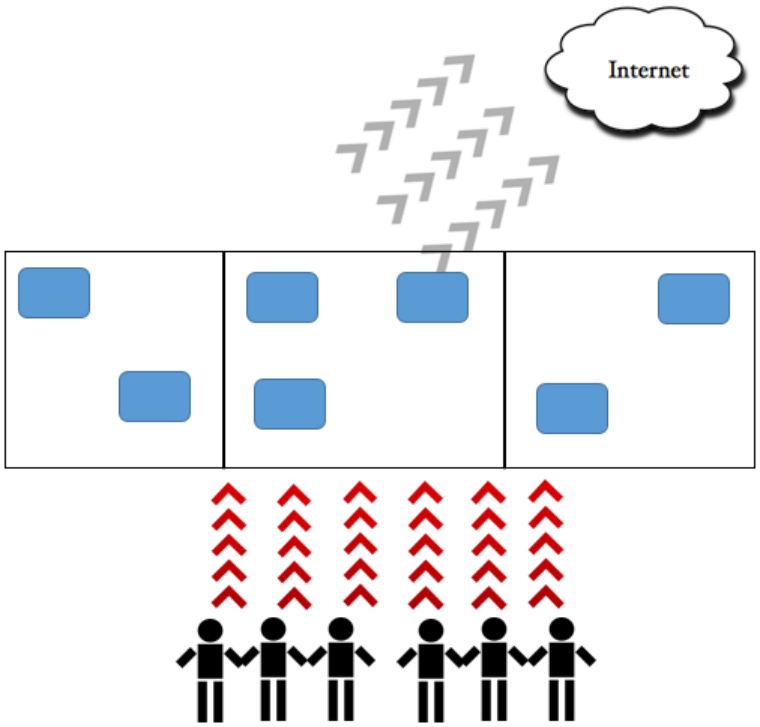


FIGURE 1 – Fonctionnement et partitionnement du cache.

Les utilisateurs émettent des requêtes pour accéder aux contenus (rectangles bleus) proposés par les différents fournisseurs. Le débit utilisateurs qui en résulte est représenté par les flèches rouges au bas de la figure. Le but pour l'opérateur est de minimiser le débit Internet (représenté par les flèches grisées sur la figure), afin que les demandes des utilisateurs soient satisfaites au maximum par le cache.

La question qui est au coeur de notre travail est la suivante : *Quelle est la partition permettant de maximiser le cache hit rate de notre espace de stockage ?* Pour y répondre, nous explorerons une technique de Reinforcement Learning, que nous confronterons à des algorithmes classiques d'optimisation.

Partie 1

Modélisation du problème

Dans cette partie nous formalisons notre étude en termes de Reinforcement Learning. Précisons tout d'abord quelques principes généraux.

Dans le cadre du Reinforcement Learning, un agent capable d'apprentissage interagit avec son environnement. A chaque instant t, il perçoit une partie s_t de l'état de l'environnement. Nous noterons $\mathscr{S}$ l'ensemble de tous les états observables. L'agent peut agir sur son environnement par une action a_t à l'instant t. On note $\mathscr{A}$ l'ensemble des actions possibles. En outre, l'agent reçoit une récompense r_{t+1} et perçoit un nouvel état s_{t+1} comme conséquence de l'action a_t. Cette récompense est notée $\mathscr{R}$.

L'objectif de l'agent est alors, à chaque instant, de maximiser l'espérance du montant total futur de ses récompenses en choisissant l'action la plus adéquate.

1.1 Analyse et formalisation

Dans notre cas, l'agent correspond à l'opérateur de réseau et l'environnement est la mémoire cache qu'il met en place pour les fournisseurs de contenus. Ces derniers envisagent leurs contenus par rapport aux courbes de popularité, que nous introduisons ici. Par exemple, dans le cas de vidéos, ces courbes traduisent la popularité vis à vis des utilisateurs de chaque vidéo. Un exemple de telle courbe est présenté ci-dessous.

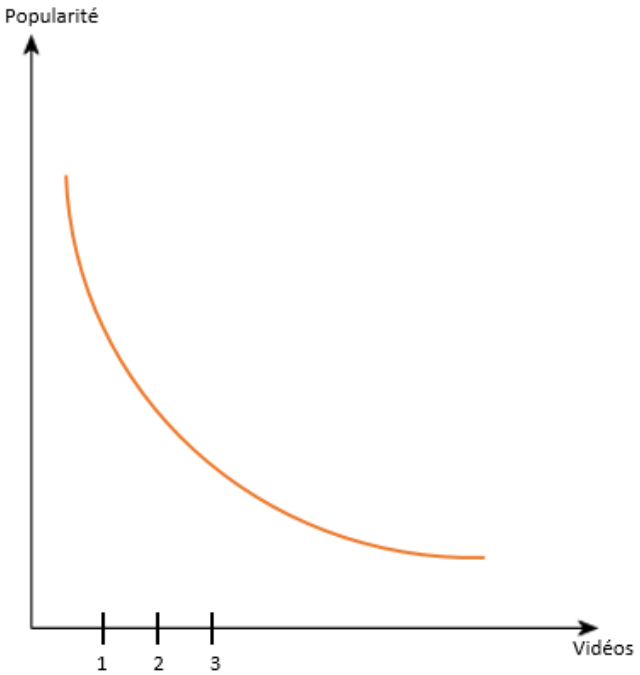


FIGURE 1.1 – Courbe de popularité de vidéos d'un fournisseur de contenus.

Ainsi, si l'on allouait 3 slots du cache au fournisseur Netflix, ce dernier choisirait de mettre ses 3 vidéos les plus populaires . La connaissance des courbes de popularité est donc cruciale. Toutefois, l'opérateur n'a a priori aucune connaissance de ces dernières. C'est précisément là que repose tout l'aspect de *learning*.

L'allocation que l'agent met en place doit s'adapter aux requêtes des utilisateurs. Cependant, en tant qu'opérateur, nous n'avons pas directement accès à ces requêtes, mais plutôt aux débits D_I et D_u - respectivement les débits Internet et utilisateurs que nous avons présentés dans l'introduction.

Par ailleurs, une partition du cache étant caractérisée par l'allocation d'un certain volume mémoire aux fournisseurs de contenus, nous noterons θ_i l'espace mémoire alloué pour le fournisseur de contenus i. Définissons alors le vecteur $\theta \in \mathbb{R}^P$ représentant l'allocation globale pour P fournisseurs de contenus au niveau du cache de capacité C :

$$\theta = \left\{ \theta_i / \sum_{i=1}^{P} \theta_i \leq C \right\} \tag{1.1}$$

Cette formule exprime le fait que la somme des espaces alloués au fournisseur doit être inférieur (ou dans un meilleur cas égale) à la taille du cache.

Finalement, un état correspond aux informations données par le vecteur θ ainsi que par les débits D_I et D_u, ce que l'on note :

$$\mathscr{S} = \{(\theta, D_I, D_u)\} \tag{1.2}$$

L'opérateur ne pouvant effectuer des actions que sur l'allocation en elle-même, l'ensemble des actions est simplement noté :

$$\mathscr{A} = \{\theta\} \tag{1.3}$$

Enfin, de façon cohérente avec nos objectifs, nous représentons la récompense comme la différence entre les débits utilisateurs et Internet. Ainsi, maximiser ce reward reviendra à minimiser le débit Internet.

$$\mathscr{R} = D_u - D_I \tag{1.4}$$

1.2 Requêtes et allocation

Notre problème étant désormais modélisé, nous souhaitons pouvoir générer un *input* pour notre simulation, c'est à dire une séquence de requêtes de vidéos. Dans un premier temps, nous supposerons que les courbes de popularité ne changent pas et que nous disposons d'un catalogue de 1000 vidéos. De plus, on commencera avec un petit nombre de fournisseurs de contenus (2-3).

1.2.1 Génération des données de requêtes

Pour pouvoir générer nos données, il nous faut savoir quelles sont les vidéos i qui sont demandées. On se place dans le cas d'une distribution de Zipf, dans laquelle la probabilité d'apparition de la vidéo i est la suivante :

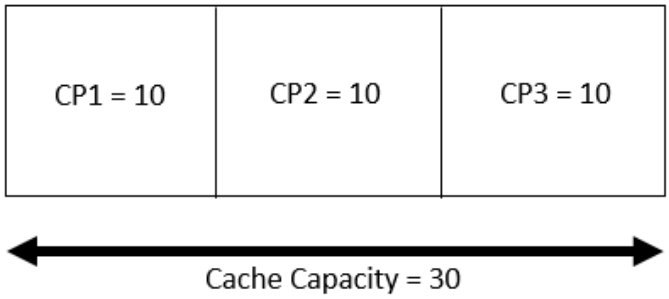
$$p_i = \frac{1}{i^\alpha} \times \frac{1}{\sum_{i=1}^{nbVidos} \frac{1}{i^\alpha}} \tag{1.5}$$

L'idée consiste alors à générer un nombre aléatoire r afin de savoir pour quelle vidéo la requête est générée. Pour ce faire, le critère de choix que nous avons adopté est le suivant : si r appartient à l'unique segment de [0,1] de longueur p_j, alors la requête est destinée à la vidéo j. Un processus similaire est utilisé pour déterminer le Content Provider concerné par la requête.

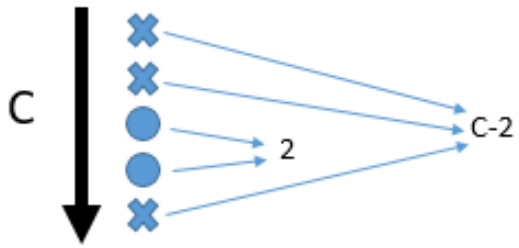
1.2.2 Étude de l'allocation

Une fois les requêtes générées, un autre aspect de notre travail doit être de pouvoir réaliser une allocation du cache parmi les différents CPs. A posteriori, nous devons également pouvoir en déterminer les performances pour choisir telle répartition plutôt qu'une autre.

En première approche, on peut considérer une répartition *naïve* du cache parmi les Content Providers. Il s'agit là d'attribuer à chacun des CPs le même nombre de slots dans le cache. Cette approche n'offre toutefois que peu de perspectives d'utilisation efficace. Une approche plus optimale que nous avons mise en oeuvre consiste à prendre, dans le cadre de notre distribution Zipf, les C vidéos les plus populaires (C étant la capacité du cache) et de compter à chaque fois le CP concerné. Ceci donne alors l'allocation optimale. Une illustration de ces deux types d'allocation est présentée à la figure ci-dessous.



(a)



(b)

FIGURE 1.2 – Principe de l'algorithme d'allocation : (a) Allocation naïve. (b) Allocation optimale.

Pour une allocation donnée, il convient alors d'étudier ses performances, et en particulier d'évaluer son coût. Le principe est de générer des requêtes et d'observer si elles sont satisfaites par l'allocation en question ou non.

Dans la suite nous présentons l'algorithme de Reinforcement Learning que nous utilisons : l'algorithme SARSA. Nous explicitons également les implications et les particularités qui sont propres à notre travail.

1.3 L'algorithme SARSA

SARSA signifie État-action-récompense-État-Action. En SARSA, l'agent commence à l'état 1, effectue l'action 1, et obtient une récompense (récompense 1). Maintenant, il est dans l'état 2 et effectue une autre action (action 2) et obtient la récompense de cet état (récompense 2) avant qu'il ne remonte et mette à jour la valeur de l'action 1, effectuée dans l'état 1.

Nous allons expérimenter cette approche (détaillée en Sec. 6.4 de [4]). Pour pouvoir l'utiliser, il faut utiliser des ensembles d'actions $\mathscr{A}$ et des états $\mathscr{S}$ plutôt petits. Dans ce cadre, une action A est un vecteur de P éléments, où un seul élément est $+\Delta$ et un seul autre élément est $-\Delta$. De plus, Δ fait partie d'un ensemble discret $\mathcal{D}$ de valeurs entières qui inclut 0, par exemple $\mathcal{D} = \{0, 1, 10, 100\}$. Notons par ailleurs que P représente le nombre de Content Providers. Ainsi, pour deux CPs, l'ensemble des actions est le suivant :

$$\mathscr{A} = \{\Delta(+1, -1), \Delta(-1, +1), \Delta(0, 0)/\Delta = 1, 10, 100\} \tag{1.6}$$

En d'autres termes, une action consiste à enlever Δ slots à un CP pour rajouter le même nombre de slots à l'autre CP. Le principe est le même pour le cas général à P Content Providers, et l'on peut montrer que le nombre d'actions possible est alors :

$$|\mathscr{A}| = P \cdot (P - 1) \cdot (|\mathcal{D}| - 1) + 1 \tag{1.7}$$

Ainsi, en considérant qu'on ne retire ou n'ajoute qu'un seul slot au plus par action (ce qui est caractérisé par $\mathcal{D} = \{0, 1\}$), on a 3 actions possibles avec 2 CPs et 7 avec 3 CPs.

L'état est pour l'instant simplement l'allocation θ. Il faut remarquer que cette modélisation est raisonnable seulement si la popularité des contenus est stationnaire : comme elle ne change pas, ce n'est pas nécessaire de l'introduire dans le vecteur d'état (si on le faisait, elle serait une constante et donc ne contribuerait à l'évolution de toute politique de Reinforcement Learning). Pour maintenir l'espace des états petit, on va considérer seulement des scénarios où la capacité C et le nombre de fournisseurs de contenu P sont petits, par exemple $C = 100$ et $P = 3$.

Le nombre total d'états possibles s'écrit alors en fonction du Cache Capacity et du nombre de Content Providers :

$$|\mathscr{S}| = \binom{C + P - 1}{P - 1} \tag{1.8}$$

L'algorithme SARSA repose également sur une table Q(s,a) qui associe une valeur à chaque état et chaque action. Cette table, initialisée à 0, est constamment remise à jour. L'objectif est, rappelons-le, de maximiser le reward $\mathscr{R}$, qui est défini comme la différence entre le nombre de requêtes reçues et le coût.

Il nous faut également une stratégie de sélection de la prochaine action à tester. Nous utilisons une politique ϵ-greedy, qui est un exemple de politique souple. Une politique souple est une politique pour laquelle, pour chaque état, toutes les actions possible ont une probabilité non nulles d'être choisies. Dans le cadre d'une politique ϵ-greedy, une action a est sélectionnée avec une probabilité ϵ si elle ne donne pas un retour maximal. L'action qui donne un retour maximale est sélectionnée avec une probabilité 1-ϵ. On expérimente avec différentes valeurs de ϵ, par exemple $0.01, 0.1, 0.5$.

On suppose que l'allocation peut changer à chaque intervalle T, par exemple $T = 10s$. Ainsi, à chaque intervalle T on génère une séquence de requêtes qui suit les distributions de probabilités qui décrivent la popularité des contenus. On calcule la bande économisée, ce qui correspond à la récompense.

La figure 1.3 présente l'algorithme SARSA que nous mettons en oeuvre.

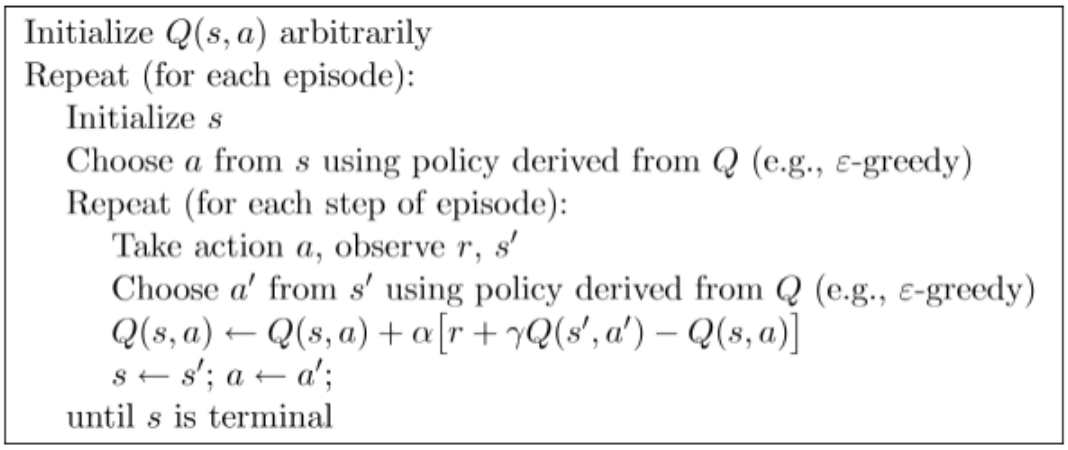


FIGURE 1.3 – Principe de l'algorithme SARSA mis en oeuvre dans notre travail.

En particulier, la formule de mise à jour de la table Q(s,a) que nous utilisons met en jeu des paramètres α et γ. Il s'agit de paramètres qui permettent d'assurer la convergence.

Partie 2

Élaboration du code

Dans toute cette partie nous utiliserons les conventions suivantes :

- CP : Content Provider. Ce sont les fournisseurs de contenus (Youtube, Netflix etc). Nous avons commencé à écrire notre algorithme pour 2 CPs puis pour k CPs ($k \in \mathbb{R}$). Au final, nous avons fixé k=3 pour simplifier notre démarche.
- Nous fixons le Cache Capacity à 100 vidéos (taille du cache).

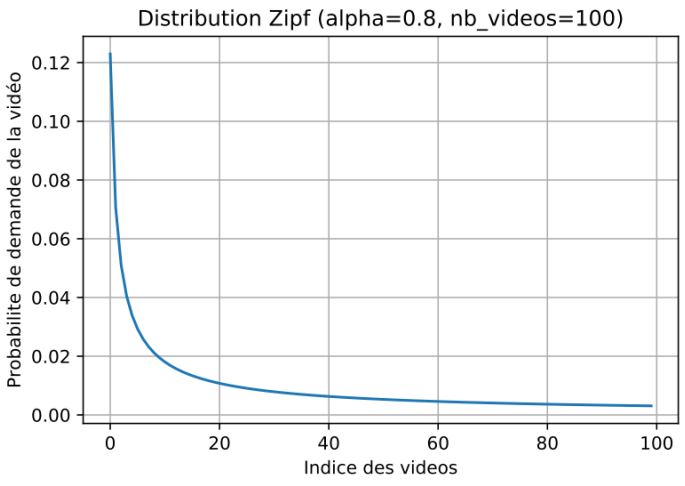
Pour plus d'information concernant le code et/ou les variables, on se reportera aux annexes A et B.

2.1 Codage de la modélisation

Avant de commencer tout type de test, nous avions besoin de modéliser notre système informatiquement. Pour cela, nous avons utilisé le langage Python (IDE : Spider) sur Windows 10. Comme nous l'avons présenté dans la 1ère partie, chaque CP dispose d'une courbe décrivant la demande pour chacune de ses vidéos proposées, correspondant à une probabilité. Cette courbe suit une loi de Zipf que nous représentons dans notre fonction zipf_distribution(alpha, nb_videos, norme). Alpha est défini dans (1.5) et la variable nb_videos représente le nombre de vidéos dans le catalogue du CP. La variable norme représente le terme, ie la partie de droite de (1.5). Nous expliquerons son utilité par la suite. La figure ci-dessous présente le retour de la fonction zipf_distribution.

```
In [15]: zipf_distribution(0.8, 30, 5.466973620168242)
Out[15]:
[0.1829165584979036,
 0.1050579749241304,
 0.07595493876294337,
 0.060339961487334164,
 0.0504750802663926,
 0.043624656605446795,
 0.038563335406063715,
 0.034656207250542595,
 0.03153980574453351 5,
 0.02899032083517424,
 0.02686203270826806,
 0.025055785639993635,
 0.023501649223157 03,
 0.022148819972072152,
 0.020959456389140634,
 0.0199047641295673,
 0.018962425372205043,
 0.018111486148783584,
 0.017348029058954804,
 0.016650566927100457,
 0.016013180016701364,
 0.015428186391932037,
 0.014889179061386074,
 0.014390769873909508,
 0.013928393081635501,
 0.013498152901178912,
 0.013096703948476209,
 0.012721156533522378,
 0.012369000964579432,
 0.012038046537918972]
```

(a)



(b)

FIGURE 2.1 – Retour de la fonction zipf_distribution avec 30 CPs et $\alpha = 0.8$: (a) Retour en ligne de commande des probabilités de chaque vidéo. (b) Représentation graphique.

Comme nous pouvons l'espérer, la somme des probabilités vaut 1. Par ailleurs, bien qu'on parle de probabilités, une fois que l'on a fixé les paramètres alpha et nb_videos, la fonction zipf_distribution nous retourne toujours la même liste et donc la "variable" norme est calculable pour chaque couple (alpha, nb_videos).

Une autre partie du système essentielle pour la modélisation est la création de requêtes. Un utilisateur génère une requête qui est adressée à un CP et concernant une vidéo précise. Ainsi, la fonction request_creation renvoie un couple [CP, numéro_de_la_vidéo]. L'utilisation de cette fonction est illustré à la figure 2.2 ci-dessous.

```
In [47]: request_creation()
Out[47]: [1, 67]
```

FIGURE 2.2 – Retour de la fonction request_creation. Cette fonction renvoie le numéro du CP suivi du numéro de la vidéo qui fait l'objet de la requête.

Nous nous sommes ensuite demandés s'il était vraiment utile de faire tout ce travail. Ne pouvait-on pas simplement distribuer équitablement le cache entre les 3 CPs ?

Pour répondre à ces questions, nous avons codé les fonctions decide_naive_alloc et decide_opt_alloc. Ces deux fonctions retournent des allocations, i.e une partition de la taille du cache entre les 3 CPs. Les retours de ces deux fonctions sont présentés en figure 2.3.

```
In [49]: decide_naive_alloc()
Out[49]: [10.0, 10.0, 10.0]
```

(a)

```
In [53]: decide_opt_alloc()
Out[53]: [23, 7, 1]
```

(b)

FIGURE 2.3 – Retour des fonctions de décision d'allocation : (a) Allocation naive. (b) Allocation optimale.

Contrairement à la fonction naive, la fonction optimale prend en compte la probabilité que la requête soit pour tel ou tel CP (par exemple Netflix plus populaire que Dailymotion) mais aussi les probabilités de Zipf calculées précédemment. Par ailleurs, en raison du seed (que nous expliciterons plus bas), chaque appel de la fonction optimale renvoie la même allocation, ce qui est commode pour la réalisation de tests.

Une dernière fonction primordiale à la modélisation est evaluate_cout. Cette fonction retourne le coût associé à une allocation avec un nombre de requêtes. L'objectif de notre projet est de minimiser ce coût. La figure ci-dessous montre l'utilisation de cette fonction.

```
In [56]: evaluate_cout2(decide_opt_alloc(), 1000)
Out[56]: 408

In [57]: evaluate_cout2(decide_opt_alloc(), 1000)
Out[57]: 451
```

FIGURE 2.4 – Retour de la fonction evaluate_cout pour 1000 requêtes et pour l'allocation optimale. Cette fonction renvoie le coût associé.

Comme nous pouvons le constater, le coût n'est pas constant. En effet, même si l'allocation retournée par decide_opt_alloc est toujours la même, la génération de requête a un côté aléatoire qui rend le résultat pseudo-aléatoire (on reste dans le même ordre de grandeur néanmoins). Si l'on prenait un nombre infini de requêtes, on obtiendrait un résultat identique à chaque fois grâce à la loi des grands nombres.

2.2 Codage de SARSA

Comme expliqué précédemment, nous avons commencé par écrire le code pour 2 CPs. En prenant du recul, cela nous a permis de faire un gain de temps considérable. En effet, bien que nous ayons pris un certain temps à écrire la fonction sarsa_pour_2 (dans Code2CP.py), cet exercice nous a donné un ensemble d'informations utiles pour écrire sarsa_pour_3 , et notamment qu'il nous serait impossible d'expliciter tous les cas possibles d'égalités avec 7 actions (d'après 1.7) ; nous l'avions fait pour les 3 actions dans sarsa_pour_2. De plus, l'écriture de sarsa_pour_2 nous a permis de nous rendre compte d'une mauvaise compréhension de l'algorithme SARSA.

Rappelons que la table Q qui est modifiée dans l'algorithme SARSA est constitué des ensembles explicites des actions en ligne et des états (allocations) en colonne.

Lors de la rédaction de la fonction sarsa_pour_3, plusieurs problèmes se sont confrontés à nous. Tout d'abord, contrairement au cas avec 2 CPs où les 101 états ([0, 100], [1, 99], ... , [100, 0]) étaient simple à implémenter dans l'algorithme avec une boucle, décrire les 5151 états pour 3 CPs (formule 1.8) est beaucoup plus complexe et fastidieux. De plus, avec plus de 2 CPs, on ne peut plus classer dans un ordre relativement utilisable les 5151 allocations ; dans le cas avec 2 CPs il suffisait de dire que l'allocation [i, 100-i] (avec i $\in [\![0; 100]\!]$) correspondait au i-ème élément de la liste.

Afin de résoudre ces problèmes, nous avons crée les fonctions states_2CP, states_3CP et position_etat. Les fonctions states_2CP et states_3CP renvoient respectivement la liste des états existants dans les cas avec 2CPs et 3CPs. Il est intéressant de noter que la fonction states_3CP appelle dans sa boucle la fonction states_2CP. On pourrait ainsi construire une fonction récursive states_kCP qui retournerait l'ensemble des états possibles pour k CPs. Cependant, nous n'avons pas jugé utile de prendre le temps de la faire sachant que nous allons nous limiter à 3 CPs.

```
In [60]: states_2CP()
Out[60]:
[[0, 30],
 [1, 29],
 [2, 28],
 [3, 27],
 [4, 26],
 [5, 25],
 [6, 24],
 [7, 23],
 [8, 22],
 [9, 21],
 [10, 20],
 [11, 19],
 [12, 18],
 [13, 17],
 [14, 16],
 [15, 15],
 [16, 14],
 [17, 13],
 [18, 12],
 [19, 11],
 [20, 10],
 [21, 9],
 [22, 8],
 [23, 7],
 [24, 6],
 [25, 5],
 [26, 4],
 [27, 3],
 [28, 2],
 [29, 1],
 [30, 0]]
```

FIGURE 2.5 – Retour de la fonction states_2CP pour un Cache Capacity de 30.

```
In [63]: states_3CP()
Out[63]:
[[0, 0, 30],
 [0, 1, 29],
 [0, 2, 28],
 [0, 3, 27],
 [0, 4, 26],
 [0, 5, 25],
 [0, 6, 24],
 [0, 7, 23],
 [0, 8, 22],
 [0, 9, 21],
 [0, 10, 20],
 [0, 11, 19],
 [0, 12, 18],
 [0, 13, 17],
 [0, 14, 16],
 [0, 15, 15],
 [0, 16, 14],
 [0, 17, 13],
 [0, 18, 12],
 [0, 19, 11],
 [0, 20, 10],
 [0, 21, 9],
 [0, 22, 8],
 [0, 23, 7],
 [0, 24, 6],
 [0, 25, 5],
 [0, 26, 4],
 [0, 27, 3],
 [0, 28, 2],
 [0, 29, 1],
 [0, 30, 0],
 [1, 0, 29],
 [1, 1, 28],
 [1, 2, 27],
 [1, 3, 26],
 [1, 4, 25],
 [1, 5, 24],
 [1, 6, 23],
 [1, 7, 22],
 [1, 8, 21],
 [1, 9, 20],
 [1, 10, 19],
 [1, 11, 18],
```

FIGURE 2.6 – Retour de la fonction states_3CP pour un Cache Capacity de 30 (résultats partiels).

La fonction position_etat(allocation) retourne la position de l'allocation donnée dans la liste des états. Cela est essentiel afin de pouvoir nous positionner dans la table Q et ainsi résoudre le problème de l'absence d'ordre. Un exemple de son utilisation est montré en figure 2.7.

```
 [27, 3, 0],
 [28, 0, 2],
 [28, 1, 1],
 [28, 2, 0],
 [29, 0, 1],
 [29, 1, 0],
 [30, 0, 0]]

In [77]: position_etat([30, 0, 0])
Out[77]: 495

In [78]: position_etat([29, 1, 0])
Out[78]: 494
```

FIGURE 2.7 – Retour de la fonction position_etat(allocation).

2.3 Tests de validation

Comme nous l'avons vu, la formule caractéristique de l'algorithme SARSA dépend de 2 paramètres α et γ. De plus, avec notre politique ϵ-greedy, nous avons un troisième paramètre ϵ qu'il est intéressant d'observer.

Dans cette partie, nous allons expliquer les motivations de ces tests et les challenges qu'ils ont apportés. L'interprétation des tests sera faite en partie 3.

A ce stade du développement, notre code global était assez brut : il n'y avait qu'un seul fichier 'CodeCasiopee.py' qui contenait toutes les fonctions, chaque fonction appelant un nombre trop important de paramètres etc. Afin de rendre le code plus lisible, nous avons fait le choix de séparer le code en différents fichiers. Pour les présenter succinctement, le fichier 'FonctionsAuxiliaires.py' contient des fonctions non essentielles pour la compréhension du projet (calcul de la somme des éléments d'une liste par exemple). Le fichier nommé 'des_tests_basiques.py' nous a servi à mettre en évidence certains aspects de l'algorithme. Par exemple la comparaison des fonctions decide_naive_alloc et decide_opt_alloc. Avec le fichier 'Code2CP.py', 'des_tests_basiques.py' contient du code correspondant à une situation avec 2CPs. Ces lignes de code compilent et renvoient des résultats. Cependant, il est fortement probable qu'elles ne soient pas optimisées. En effet, nous avons préféré nous focaliser sur le modèle avec 3 CPs qui semblait être plus riche et plus complexe. Le code était alors devenu plus lisible. Cependant, un problème persistait : la complexité d'effectuer les tests. Nous avions deux problèmes à résoudre : gérer les variables et contrôler le temps de calcul.

Le premier problème a été résolu en 3 étapes : la création d'une fonction init() qui déclare des variables globales utilisables partout dans le code, la création d'un fichier 'generator.py' qui est capable de générer des listes de alphas de zipf (différents ou égaux), des listes correspondantes aux catalogues des CPs (nombre de vidéos proposé par chaque CP) ou enfin des listes illustrant la popularité de chaque CP. Enfin, le 3ème outil utilisé est seed(n), où $n \in \mathbb{N}$, du module random de python. La fonction seed renvoie une suite de nombres pseudo-aléatoires qui sont stockés dans le module random. De plus, à chaque fois que cette fonction est appelée avec le même paramètre n, la même suite de nombres pseudo-aléatoires sera stockée dans le module random. L'appel de cette fonction dans notre fonction init() permet de pouvoir faire des tests répétitifs avec les mêmes variables pseudo-aléatoires, et ainsi de pouvoir comparer nos résultats.

L'autre problème des temps de calcul était plus complexe à gérer. En effet, nous n'avions quasiment jamais eu ce type de problème concrètement dans notre vie de développeur. L'aide de notre encadrant Mr. Araldo nous fut précieuse. En effet, il sut détecter les sources de surcharge de calculs (comme par exemple le calcul répétitif et inutile de la constante de normalisation dans la fonction zipf_distribution).

Cependant, même optimisé, il fallait laisser l'algorithme tourner un minimum de 3h pour observer la convergence du coût. Nous nous sommes alors demandés quelle pouvait être la cause de ce temps si élève. Pour cela, nous avons introduit la fonction sarsa_pour_3_bis. Comme vous pouvez le voir sur le code dans l'annexe A, cette fonction diffère en quelques endroits de la fonction sarsa_pour_3. La fonction sarsa_pour_3_bis est en fait l'ancienne version, non optimisée de la fonction finale sarsa_pour_3. Nous avons décidé de la laisser telle qu'elle car c'est avec cette fonction que nous avons réussi à localiser les lignes de code gourmandes en calcul. A notre surprise, il s'est avéré que la ligne correspondant à la modification de la table Q prenait autant de temps qu'une boucle de l'algorithme ; d'habitude avec Python les modifications de matrices se font quasi instantanément. Dès lors, il ne nous semblait plus possible d'optimiser le code.

Partie 3

Présentation des résultats

Dans cette section nous présentons les principaux résultats de notre travail, ainsi que les interprétations qui en découlent. En particulier, nous nous intéressons à l'impact des paramètres ϵ, α et γ sur l'évolution du coût.

3.1 Paramètres et équations

Nous nous basons ici sur les développements de la section 1.3. Les 3 paramètres en question sont tous des réels compris entre 0 et 1.

3.1.1 Paramètre α (learning rate)

$\alpha = \mathbf{0}$
Notons que si $\alpha = 0$, on retrouve que Q(s,a)=Q(s,a), ce qui signifie que la table n'est jamais modifiée. En outre, si $\gamma = 0$, on ne prend jamais en compte le nouvel état et rien n'est modifié.

$\alpha = \mathbf{1}$
Par ailleurs, on a :

$$\alpha = 1 \Rightarrow Q(s_{i-1}, a_{i-1}) \longleftarrow R_i + \gamma Q(s_i, a_i)$$

C'est à dire :

$$Q(s_{i-1}, a_{i-1}) \longleftarrow R_i + \gamma R_i + 1 + \gamma^2 Q(s_{i+1}, a_{i+1})$$

Et donc par une récurrence immédiate on obtient la formule suivante :

$$Q(s_{i-1}, a_{i-1}) \longleftarrow \sum_{k=0}^{n<etapesSARSA} \gamma^k R_{i+k} + \gamma^{n+1} Q(s_{i+n-1}, a_{i+n-1}) \tag{3.1}$$

3.1.2 Paramètre γ (discount factor)

$\gamma = \mathbf{0}$
Si $\gamma = 0$, alors :

$$Q(s, a) \longleftarrow Q(s, a) + \alpha(R - Q(s, a)) \tag{3.2}$$

On ne prend donc pas en compte Q(s',a') : on considère seulement la récompense actuelle.

$\gamma = \mathbf{1}$
On a :

$$Q(s,a) \longleftarrow Q(s,a) + \alpha(R + Q(s',a') - Q(s,a)) \tag{3.3}$$

On prend donc autant en compte Q(s',a') et Q(s,a).
Si en outre on a $\alpha = 1$:

$$Q(s,a) \longleftarrow R + Q(s',a') \tag{3.4}$$

Dans ce cas, on prend seulement en compte le reward à long terme. Notons qu'une valeur de γ trop proche de 1 introduit une possibilité pour que Q diverge.

3.1.3 Paramètre ϵ

Ce paramètre correspond à celui de la politique ϵ-greedy. A la limite, le comportement est totalement aléatoire si sa valeur est 1. A l'inverse, une valeur égale à 0 traduit le fait que l'on explore toute la table Q, et que rien n'est laissé au hasard.

3.2 Cas théorique d'un nombre de requêtes infini

Notre motivation est de d'abord observer la convergence avant de se lancer dans des calculs assez lourds en termes de temps. Typiquement, on considère un nombre de l'ordre de 10^{200} requêtes. La loi des grands nombres nous assure que la probabilité de requête d'une vidéo se confond avec la probabilité Zipf de cette même vidéo.

Dans ce cadre, le calcul du coût est réalisé en sommant,pour chaque CP, les probabilités des vidéos qui sont dans le cache. On retranche cette somme à 1 et on multiplie le résultat par le nombre de requêtes concernant le CP en question. Notons nb(i) le nombre de requêtes concernant le CP i, et al(i) le nombre de spots alloués au CP i. Le coût est donc :

$$Cout = \sum_i nb(i) \left(1 - \sum_{n=0}^{al(i)} p_n\right) \tag{3.5}$$

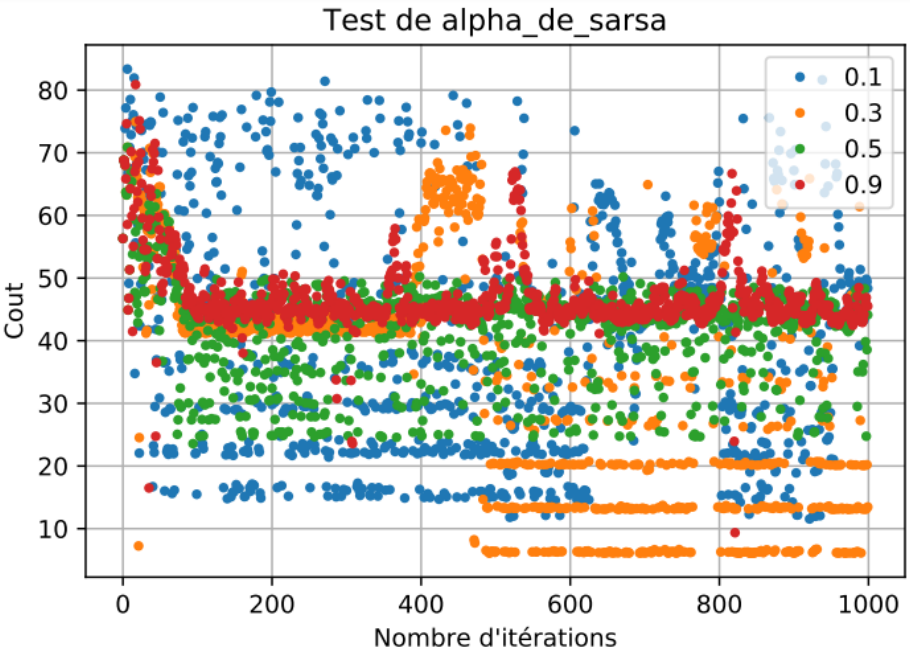


FIGURE 3.1 – Influence du paramètre α sur l'évolution du coût (cas infini).

La figure 3.1 ci-dessus présente l'évolution du coût en fonction de α. Nous remarquons que α petit détermine un changement de la table Q très lent (chaque mise à jour est très petite). On ne voit donc pas de convergence. On pourrait voir la convergence si on utilisait 10^6 intervalles. A l'inverse, plus α est proche de 1, plus on observe une convergence du coût (vers 45 ici).

La figure 3.2 montre le test d'impact du paramètre γ sur le coût.

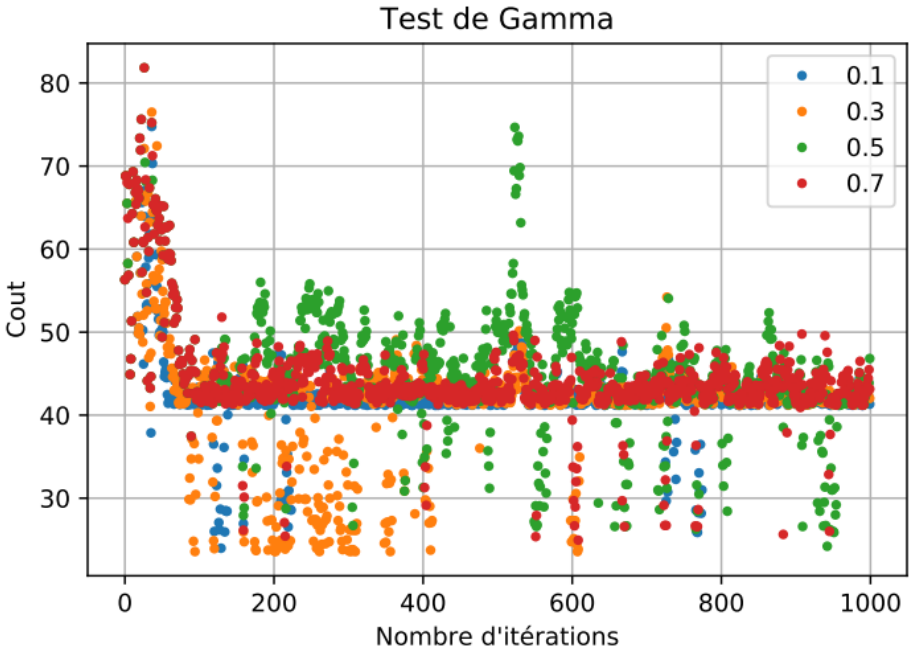


FIGURE 3.2 – Influence du paramètre γ sur l'évolution du coût (cas infini).

Nous voyons ici que la valeur de γ la plus stable semble être 0.7. Pour les autres valeurs testées, la convergence apparaît plus chaotique.

Enfin, l'impact du paramètre ϵ est illustré ci-après.

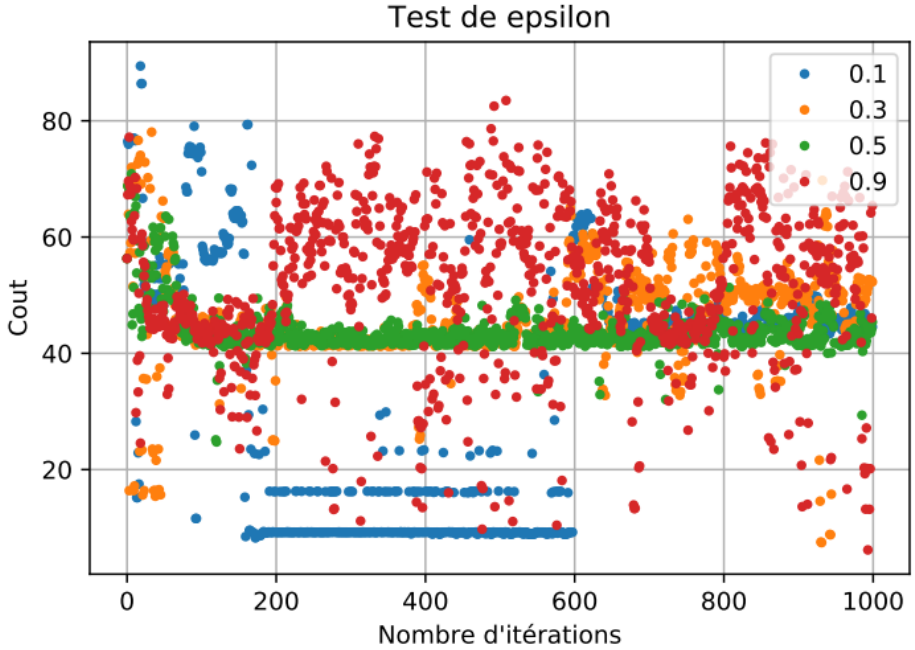


FIGURE 3.3 – Influence du paramètre ϵ sur l'évolution du coût (cas infini).

Nous remarquons que plus ϵ augmente, plus les choix se font aléatoirement et donc moins on converge. La valeur la plus stable correspond au compromis moitié exploration moitié exploitation ($\epsilon = 0.5$).

3.3 Cas réel d'un nombre de requêtes fini

Dans le cas réel, les requêtes sont générées par les différents utilisateurs. A présent, on évalue le coût par la fonction evaluate_cout.

A chaque intervalle de SARSA, on génère taille_intervalle*request_rate = nb_iterations requêtes. Ceci a pour conséquence une très forte augmentation du temps de calcul.

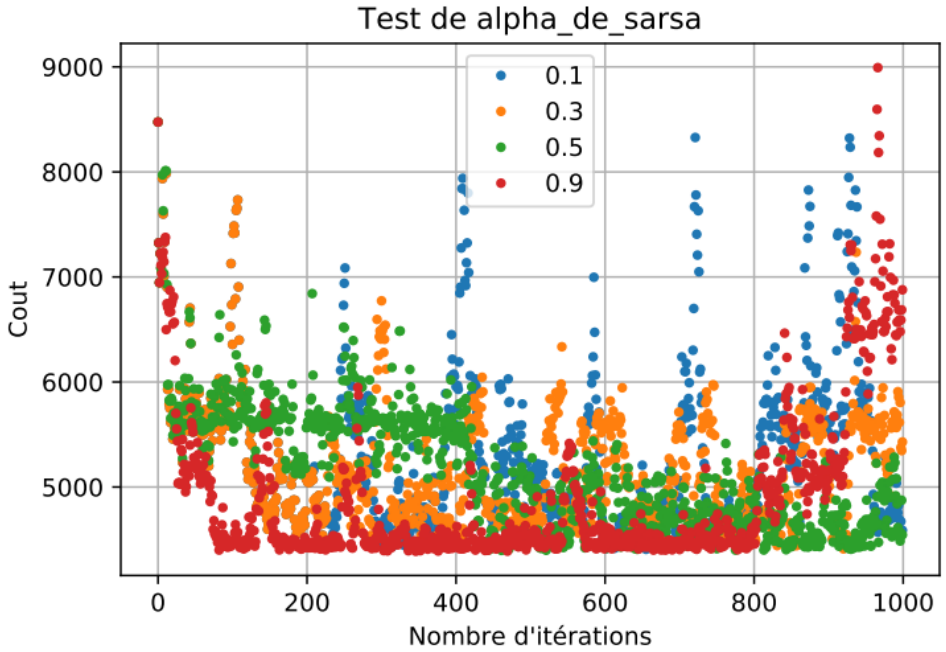


FIGURE 3.4 – Influence du paramètre α sur l'évolution du coût (cas fini).

La figure 3.4 ci-dessus présente l'évolution du coût en fonction de α. Nous remarquons que α petit détermine encore un changement de la table Q très lent (chaque mise à jour est très petite). On ne voit donc pas de convergence. La valeur la plus stable semble être autour de 0.5.

La figure 3.5 montre le test d'impact du paramètre γ sur le coût.

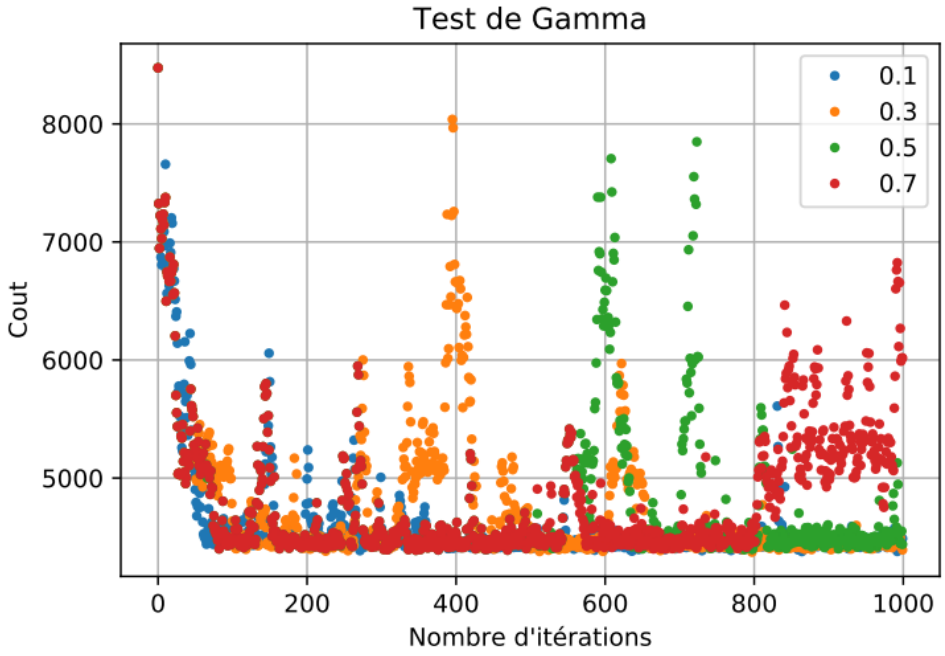


FIGURE 3.5 – Influence du paramètre γ sur l'évolution du coût (cas fini).

Nous voyons là que peu de réelles interprétations peuvent être tirées. Les valeurs autour de 0.5 semblent les meilleures.

Enfin, l'impact du paramètre ϵ est illustré ci-après.

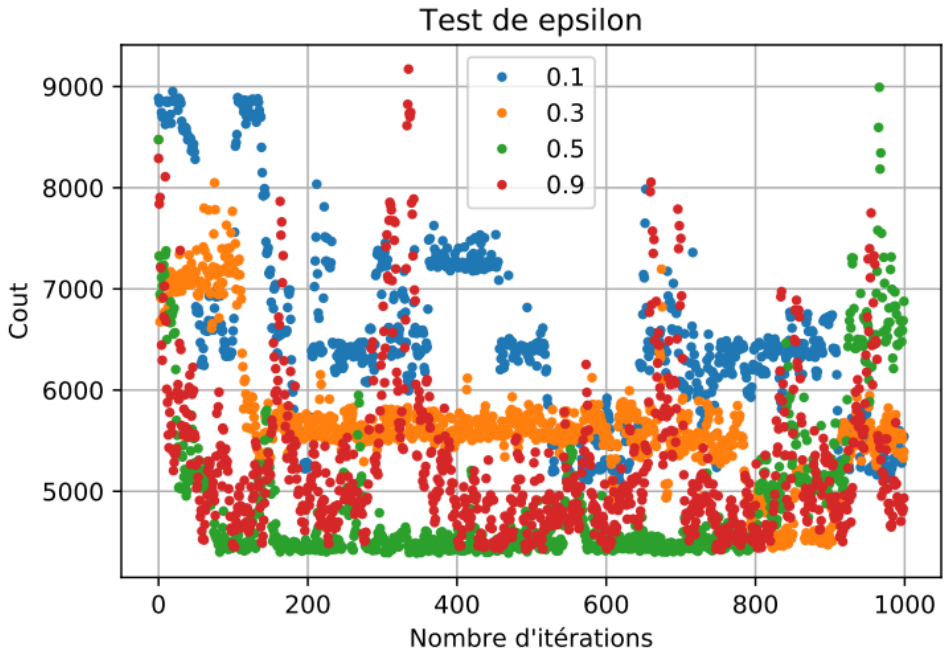


FIGURE 3.6 – Influence du paramètre ϵ sur l'évolution du coût (cas fini).

Contrairement au cas infini, le compromis moitié moitié n'est pas optimal. Dans le cas de requêtes finies, on observe une meilleure convergence du coût pour ϵ inférieur à 0.5.

Partie 4

Conclusion

Notre travail nous a permis de tirer de nombreux enseignements et d'exercer un regard critique sur les modèles que nous manipulons.

Comme présenté dans la partie 3.3, l'algorithme SARSA converge vers une valeur autour de 4360. Toutefois, le coût associé à l'allocation optimal avec 100*100 requêtes se trouve autour de 4296 (cf. figure 4.1).

```
In [29]: evaluate_cout(decide_opt_alloc(), 10000)
Out[29]: 4296
```

FIGURE 4.1 – Coût optimal obtenu par la fonction evaluate_cout.

Ainsi, l'algorithme SARSA trouve l'allocation optimale avec environ 1% d'erreur relative. L'algorithme est extrêmement performant dans notre étude. Cependant, nous pouvons nous demander quelles sont les limites de cet algorithme. En effet, comme nous l'avons vu précédemment, le temps de calcul a tendance à exploser, notamment en utilisant Python qui n'est pas forcement le langage le plus optimisé.

Par manque de temps, nous n'avons pas pu finalisé la généralisation de l'ensemble du code à k CPs – notamment la fonction states_kCP qui décrirait l'ensemble des états pour k CP. Cependant, l'utilité de développer une telle fonction serait moindre. En effet, grâce à (1.8), on se rend compte que pour 6 CPs le nombre d'états possibles est de 96.560.646 et que pour 7 CPs il y a en 1.705.904.746. Or, sur Python 32 bits, la taille des listes est limitée à environ 500 millions d'éléments. On ne pourrait même pas stocker l'ensemble des états sur une seule liste ; sans mentionner la matrice Q avec le nombre d'états correspondant à son nombre de lignes.

Cette méthode est donc efficace pour des petites valeurs de CPs. Une méthode plus appropriée pour un grand nombre de CPs serait l'approximation de fonction linéaire, présenté en annexe C.

D'un point de vue plus personnel, ce projet nous a permis de développer nos compétences d'une part technique (développement informatique avec Python, l'efficacité de la modélisation ou encore l'importance d'avoir un code clair et bien séparé) et d'une part managériale (équipe qui s'autogère, motivation de l'équipe lors d'une baisse d'activité).

Annexe A

Code

Listing A.1 – Code2CP.py

```
#Fonctions utilisees pour uniquement 2 CPs
import CodeCassiopee as cc
import random as rd
import numpy as np

# Cette fonction complete la creation d'un input
# Elle permet de creer une requete portant sur une video i d un CP (YT ou Nf)
def Request_creation(proba_yt, alpha_yt, alpha_nf, nb_videos_yt, nb_videos_nf):
    Content_Provider = 'a␣determiner'
    distribution = []
    choix_CP = rd.random()
    if choix_CP <= proba_yt:
        Content_Provider = 'youtube'
        distribution = cc.zipf_distribution(alpha_yt, nb_videos_yt)
        choix_video_yt = rd.random()
        compteur_choix = 0
        for i in range(1, nb_videos_yt +1):
            compteur_choix += distribution[i-1]
            if compteur_choix >= choix_video_yt:
                video_choisie = i
                break
    else:
        Content_Provider = 'netflix'
        distribution = cc.zipf_distribution(alpha_nf, nb_videos_nf)
        choix_video_nf = rd.random()
        compteur_choix = 0
        for i in range(1, nb_videos_nf +1):
            compteur_choix += distribution[i-1]
            if compteur_choix >= choix_video_nf:
                video_choisie = i
                break

    return [Content_Provider, video_choisie]


#Permet d'evaluer le cout pour 2 CPs
def evaluate_cout_2(allocation, proba_yt, alpha_yt, alpha_nf,
nb_videos_yt, nb_videos_nf, nb_requetes):
    cout = 0
    requete = Request_creation(proba_yt, alpha_yt, alpha_nf,
```

```
    nb_videos_yt, nb_videos_nf)
    if requete[0] == 'youtube':
        if requete[1] > allocation[0]:
            cout += 1
    else:
        if requete[1] > allocation[1]:
            cout += 1
    return cout

def sarsa_pour_2(intervalle, request_rate, proba_yt, alpha_yt,
alpha_nf, nb_videos_yt, nb_videos_nf, cache_capacity):
    nb_iterations = intervalle * request_rate
    allocation = [cache_capacity/10.0 , 9*cache_capacity/10.0]
    index = allocation[0]
    rewards = np.zeros((3, 101))
    epsilon = rd.random()
    gain_init = nb_iterations - evaluate_cout_2(allocation, proba_yt,
    alpha_yt, alpha_nf, nb_videos_yt, nb_videos_nf, nb_iterations)
    for i in range (nb_iterations):
        alea = rd.random()
        if alea <= epsilon:
            action = rd.randint(-1,1)
            index += action
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
                position = 0
            elif action == -1:
                allocation[0] -= 1
                allocation[1] += 1
                position = 1
            elif action == 0:
                position = 2
        elif rewards[0, index] == rewards[1, index] and
        rewards[0, index] == rewards[2, index]:
            action = rd.randint(-1,1)
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
                position = 0
                index +=1
            elif action == -1:
                allocation[0] -= 1
                allocation[1] += 1
                position = 1
                index -= 1
            else:
                position = 2
        elif rewards[0, index] == rewards[1, index]:
            if rewards[0, index] > rewards[2, index]:
                action = rd.randint(0,1)
                if action == 0:
                    allocation[0] += 1
                    allocation[1] -= 1
                    index += 1
```

```
                position = 0
            else:
                allocation[0] -= 1
                allocation[1] += 1
                index -= 1
                position = 1
        else:
            action = 0
            position = 2
    elif rewards[0, index] == rewards[2, index]:
        if rewards[0, index] > rewards[1, index]:
            action = rd.randint(0,1)
            if action == 0:
                allocation[0] += 1
                allocation[1] -= 1
                index += 1
                position = 0
            else:
                allocation[0] -= 0
                allocation[1] += 0
                position = 2
        else:
            allocation[0] -= 1
            allocation[1] += 1
            index -= 1
            position = 1
    elif rewards[1, index] == rewards[2, index]:
        if rewards[1, index] > rewards[0, index]:
            action = rd.randint(0,1)
            if action == 0:
                allocation[0] -= 1
                allocation[1] += 1
                index -= 1
                position = 1
            else:
                allocation[0] -= 0
                allocation[1] += 0
                position = 2
        else:
            allocation[0] += 1
            allocation[1] -= 1
            index += 1
            position = 0
    else:
        max = rewards[:,index].max()
        if rewards[0, index] == max:
            position = 0
            index += 1
        if rewards[1, index] == max:
            position = 1
            index -=1
        if rewards[2, index] == max:
            position = 2
    nouv_gain = nb_iterations - evaluate_cout_2(allocation,
    proba_yt, alpha_yt, alpha_nf, nb_videos_yt, nb_videos_nf, nb_iterations)
```

```
            delta_gain = nouv_gain - gain_init
            gain_init = nouv_gain
            nouv_Q = fa.trouver_max_col(rewards, index)
            rewards[position][index] = delta_gain + nouv_Q[0]
        return allocation
```

Listing A.2 – CodeCassiopee.py

```
import matplotlib.pyplot as plt # necessaire pour tracer les plot
import random as rd
import numpy as np
import generator as gn
import FonctionsAuxiliaires as fa #Pour alleger le code principal
import time #Permet d'etudier les couts de SARSA
from copy import deepcopy #permet de faire des copies de liste
independante de l'originale

#Initialise les variables globales
def init():
    global liste_alpha #alpha correspondant aux fonctions de zipf pour chaque CP
    global liste_proba #Popularite de chaque CP
    global liste_nb_video #Nombre de videos propose par chaque CP
    global cons_zipf_1 #alpha=0.8, 100 videos
    global cons_zipf_2 #alpha=1, 100 videos
    global cons_zipf_3 #alpha=1.2, 100 videos
    global conss_zipf #liste contenant les cons_zipf ci dessus
    global k #Nombre de CPs
    global cache_capacity #Taille du cache memoire
    global alpha #alpha de zipf utilise pour faire des tests
    global nb_videos #nb video pour chaque CP
    global gamma #parametre de la l'equation pour calculer le nouveau
    Q dans SARSA
    global epsilon #Politique epsilon-greedy
    global alpha_de_sarsa #parametre de la l'equation pour calculer
    le nouveau Q dans SARSA
    rd.seed(5)
    liste_alpha=[0.8, 1.0, 1.2]
    liste_proba=[0.7, 0.25, 0.05] #test pour essayer de trouver la convergence
    liste_nb_video=gn.liste_100_videos(k) #100 videos pour chaque CP
    cons_zipf_1 = 8.13443642804101
    cons_zipf_2 = 5.187377517639621
    cons_zipf_3 = 3.6030331432380347
    conss_zipf=[cons_zipf_1, cons_zipf_2, cons_zipf_3]
    k=3
    cache_capacity = 30
    alpha=0.8
    nb_videos = 100
    gamma = 0.8
    epsilon = 0.5
    alpha_de_sarsa = 0.9

# Cette fonction permet de creer un input sur les videos d'un CP
# Elle retourne le graphe des probabilites pi de la vid o i en fonction de i
# nb_videos est le nombre de films du catalogue du CP
# alpha est le parametre present dans la loi de distribution de zipf
```

```
def zipf_distribution(alpha, nb_videos, norme):
    "indices_videos = range(1,nb_videos+1)" # necessaire pour tracer
    le plot (decocher si besoin)
    probabilites_pi = [0] * (nb_videos)
    for i in range(1, nb_videos+1):
        pi = (1.0/i**alpha) * (1.0/norme)
        probabilites_pi[i-1] = pi
    """
    liste_abscisse=[k for k in range(nb_videos)]
    plt.plot(liste_abscisse, probabilites_pi, "-")
    plt.title('Distribution Zipf (alpha=0.8, nb_videos=100)')
    plt.xlabel('Indice des videos')
    plt.ylabel('Probabilite de demande de la video')
    plt.grid('on')
    plt.savefig('zipf_distribution.pdf')
    plt.close()
    plt.show()
    """
    return probabilites_pi

# Cette fonction complete la creation d'un input
# Elle permet de creer une requete portant sur une vid o i d'un CP
def request_creation(): #k
    somme=0
    i=0
    choix_CP = rd.random()
    while(1==1):
        somme=somme+liste_proba[i]
        if(choix_CP <= somme):
            break
        else:
            i +=1
    CP=i
    distribution = zipf_distribution(liste_alpha[i], liste_nb_video[i],
    conss_zipf[i]) #conss_zipf permet de soulager les calculs des
    constantes de normalisation de zipf
    choix_video = rd.random()
    compteur_choix = 0
    for j in range(1, liste_nb_video[i]+1):
        compteur_choix += distribution[j-1]
        if compteur_choix >= choix_video:
            video_choisie = j
            break
    return [CP, video_choisie]


# Cette fonction realise une allocation naive du cache entre les CP
def decide_naive_alloc(): # cache_capacity, k
    liste_nb_video=gn.liste_nb_de_video(k)
    liste_allocation=[0]*k
    nb_video_total=fa.somme_liste(liste_nb_video)
    for i in range(k):
        liste_allocation[i]=((1.0*liste_nb_video[i])/(nb_video_total))
        *cache_capacity
    return liste_allocation
```

```
# Cette fonction realise une allocation optimale du cache entre les CP
# Le cache_capacity doit etre inferieur au nombre total de video
(sinon pas de cassiopee)
def decide_opt_alloc(): # cache_capacity, k
    distribution=[0]*k
    popularite=[0]*k
    allocation=[0]*k
    pointeurs_max=[0]*k
    if cache_capacity > fa.somme_liste(liste_nb_video):
        return 'Erreur: Le cache est trop grand par rapport au
        nombre total de videos'
    for i in range(k):
        distribution[i]=zipf_distribution(liste_alpha[i],
        liste_nb_video[i], conss_zipf[i])
        popularite[i] = [piyt * liste_proba[i] for piyt in distribution[i]]
        #liste que l'on va comparer
    for j in range (cache_capacity + 1):
        max_temp=0 #popularite[0] par defaut
        for m in range(k-1):
            if popularite[m+1][pointeurs_max[m+1]]>
            popularite[m][pointeurs_max[m]]:
                max_temp=m+1
        allocation[max_temp]=allocation[max_temp] + 1
        pointeurs_max[max_temp] = pointeurs_max[max_temp] + 1 ;
    return allocation

# Cette fonction permet d'evaluer a posteriori le cout d'une allocation donnee
# La variable allocation est du type liste
def evaluate_cout(allocation, nb_requetes): # k
    cout = 0
    for r in range(1, nb_requetes +1):
        requete = request_creation()
        if allocation[requete[0]]<requete[1]:
            cout +=1
    return cout



#Creation des 101 etats possibles pour 2 CPs
def states_2CP(cache_capacity):
    liste=[]
    for i in range(cache_capacity+1):
        liste.append([i, cache_capacity-i])
    return liste


#Creation des 5151 etats possibles pour  3 CPs
def states_3CP(cache_capacity):
    liste=[]
    for m in range (cache_capacity + 1): #le premier CP sur les 3
        for i in range(cache_capacity - m+1):
            liste.append([m, states_2CP(cache_capacity - m)[i][0],
            states_2CP(cache_capacity - m)[i][1]])
```

```
    return liste


#Recheche la position d'un etat pour 3 CP
def position_etat(alloc): #la liste est en fait une allocation a 3 CP
    #la generalisation a k CP n'est pas faite ici par manque
    de la fonction states_kCP
    cache_capacity=alloc[0]+alloc[1]+alloc[2]
    compteur=-1
    for k in states_3CP(cache_capacity):
        compteur +=1 ;
        if alloc == k:
            return compteur

#Utile uniquement pour tester la convergence
def sarsa_pour_3(request_rate, nb_intervalle, taille_intervalle): #intervalle,
request_rate, gamma, epsilon, cache_capacity, alpha
    init()
    liste_cout=[]
    ### CAS THEORIQUE : NOMBRE DE REQUETES INFINI ####
    if request_rate == -1:
        nb_iterations = 100
    else:
        nb_iterations = taille_intervalle * request_rate
        #Nombre de requete a chaque intervalle
    allocation = [int(0*cache_capacity/10) ,
    int(0*cache_capacity/10), int(10*cache_capacity/10)]
    index = 0
    Q = np.zeros((7, 5151))
    ###### ACTION ######
    for j in range(nb_intervalle):
        alea = rd.random()
        old_allocation=deepcopy(allocation)
        #copie sur un autre pointeur
        if alea <= epsilon: #politique epsilon-greedy
            action = rd.randint(0,6) #random entre 0 et 6 inclus
            --> 7 actions possibles
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
```

```
        else: #on cherche le max
            action=fa.recherche_max(Q[:, index])
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
        if -1 in allocation:
            allocation = old_allocation
            Q[action][index] = 0
        #### CALCUL DU GAIN #####
        if request_rate == -1:
            cout_1 = 0
            for cp in range(2):
                requetes_vers_le_cp = nb_iterations*liste_proba[cp]
                hit_ratio = fa.somme_liste(zipf_distribution(liste_alpha[cp],
                nb_videos, conss_zipf[cp])[0 : (allocation[cp]-1)])
                cout_1 += requetes_vers_le_cp * (1- hit_ratio)
        else :
            cout_1 = evaluate_cout(allocation, nb_iterations)
        nouv_gain = nb_iterations - cout_1  # R dans la formule
        ## MISE A JOUR DE LA TABLE###
        index_prime = position_etat(allocation)
        # index du nouvel etat
        Q[action][index] = Q[action][index] + alpha_de_sarsa*(nouv_gain +
        gamma*Q[action][index_prime] - Q[action][index])
        index = index_prime
        liste_cout.append(cout_1)
    liste_cout_moyen=[]
    k=0
    while (k<nb_intervalle): #tous 100 intervalles
        liste_cout_moyen.append
        ((fa.somme_liste(liste_cout[k : k+100]) / 100.0))
        k += 100
    plt.plot(range(len(liste_cout_moyen)), liste_cout_moyen, ".")
    #plt.xlim(4000, 5000)
    plt.title('Cout en fonction du nombre d\'it ration' )
    plt.xlabel('Nombre d\'it rations')
    plt.ylabel('Cout')
    plt.grid('on')
    #plt.rcParams["figure.figsize"] = [16, 9]
    plt.savefig('fig8.pdf')
```

```
    plt.close()
    plt.show()
    return allocation


#Utile uniquement pour chercher les couts (temps de calcul)
de differentes parties de l'algo
def sarsa_pour_3_bis(request_rate, intervalle):#intervalle,
request_rate, gamma, epsilon, cache_capacity, alpha
    debut_algo=time.time()
    nb_iterations = intervalle * request_rate
    allocation = [int(10*cache_capacity/10),
    int(0*cache_capacity/10), int(0*cache_capacity/10)]
    index = 0
    rewards = np.zeros((7, 5151))
    gain_init = nb_iterations - evaluate_cout(allocation,
    nb_iterations)
    for i in range (0, nb_iterations):  #nb de requete
        alea = rd.random()
        old_allocation=deepcopy(allocation)
        #copie sur un autre pointeur
        if alea <= epsilon: #politique epsilon-greedy
            action = rd.randint(0,6) #random entre 0 et 6 inclus
            -->7 actions possibles
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
        else: #on cherche le max
            action=fa.recherche_max(rewards[:, index])
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
```

```
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
        if -1 in allocation:
            allocation = old_allocation
            rewards[action][index] = -150000000.0
        cout_local = evaluate_cout(allocation, nb_iterations)
        #juste utilise pour le print pour les tests
        nouv_gain = nb_iterations - cout_local
        delta_gain = nouv_gain - gain_init #R dans la formule
        gain_init = nouv_gain
        avant_Q = time.time() - debut_algo
        print('avant_Q : ', avant_Q)
        rewards[action][index] = rewards[action][index] + alpha_de_sarsa*
        (delta_gain +
        gamma*rewards[action][position_etat(allocation)] -
        rewards[action][index])
        apres_Q=time.time() - debut_algo
        print('apres changement Q : ', apres_Q)
        index = position_etat(allocation)
    fin=time.time() - debut_algo
    print('Temps total SARSA : ', fin)
    return allocation

#Utile uniquement pour tester les epsilon et gamma diff rents
def tests_sarsa_pour_3(request_rate, nb_intervalle,
taille_intervalle, gama, epsi, alfa):
#intervalle, request_rate, gamma, epsilon, cache_capacity, alpha
    ### REQUEST RATE = 100 ###
    init()
    liste_cout=[]
    ### CAS THEORIQUE : NOMBRE DE REQUETES INFINI ####
    if request_rate == -1:
        nb_iterations = 100
    else:
        nb_iterations = taille_intervalle * request_rate
        #Nombre de requ te   chaque intervalle
    allocation = [int(0*cache_capacity/10),
    int(0*cache_capacity/10), int(10*cache_capacity/10)]
    index = 0
    Q = np.zeros((7, 5151))
    ###### ACTION %%%%%
    for j in range(nb_intervalle):
        alea = rd.random()
        old_allocation=deepcopy(allocation)
        #copie sur un autre pointeur
        if alea <= epsi: #politique epsilon-greedy
            action = rd.randint(0,6) #random entre 0 et 6 inclus
            --> 7 actions possibles
            #position=action
            if action == 1:
```

```
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
        else: #on cherche le max
            action=fa.recherche_max(Q[:, index])
            if action == 1:
                allocation[0] += 1
                allocation[1] -= 1
            if action == 2:
                allocation[0] -= 1
                allocation[1] += 1
            if action == 3:
                allocation[0] += 1
                allocation[2] -= 1
            if action == 4:
                allocation[0] -= 1
                allocation[2] += 1
            if action == 5:
                allocation[1] += 1
                allocation[2] -= 1
            if action == 6:
                allocation[1] -= 1
                allocation[2] += 1
        if -1 in allocation:
            allocation = old_allocation
            Q[action][index] = 0
        ##### CALCUL DU GAIN ######
        if request_rate == -1:
            cout_1 = 0
            for cp in range(2):
                requetes_vers_le_cp = nb_iterations*liste_proba[cp]
                hit_ratio = fa.somme_liste(zipf_distribution(liste_alpha[cp],
                nb_videos, conss_zipf[cp])[0 : (allocation[cp]-1)])
                cout_1 += requetes_vers_le_cp *(1- hit_ratio)
        else :
            cout_1 = evaluate_cout(allocation, nb_iterations)
        nouv_gain = nb_iterations - cout_1  # R dans la formule
        ## MISE A JOUR DE LA TABLE###
        index_prime = position_etat(allocation)
        # index du nouvel etat
        Q[action][index] = Q[action][index] + alfa*
```

```
        (nouv_gain + gama*Q[action][index_prime] - Q[action][index])
        index = index_prime
        liste_cout.append(cout_1)
    return liste_cout


def tests_de_gamma(request_rate, nb_intervalle, taille_intervalle):
    liste_gamma = [0.1, 0.3, 0.5, 0.7]
    for k in liste_gamma:
        cout_du_sarsa=tests_sarsa_pour_3(request_rate, nb_intervalle, taille_inte
        k, epsilon, alpha_de_sarsa)
        ## On fait une moyenne tous les 10 points intervalles
        liste_cout_moyen=[]
        i=0
        while (i<len(cout_du_sarsa)): #tous 10 intervalles
            liste_cout_moyen.append((fa.somme_liste
            (cout_du_sarsa[i : i+10]) / 10.0))
            i += 10
        plt.plot(range(len(liste_cout_moyen)), liste_cout_moyen,
        ".", label = str(k))
    plt.title('Test de Gamma' )
    plt.xlabel('Nombre d\'it rations')
    plt.ylabel('Cout')
    plt.grid('on')
    plt.legend(loc = "best")
    plt.savefig('fig2.pdf')
    plt.close()
    plt.show()


def tests_de_epsilon(request_rate, nb_intervalle, taille_intervalle):
    liste_epsilon = [0.1, 0.3, 0.5, 0.9]
    for k in liste_epsilon:
        cout_du_sarsa=tests_sarsa_pour_3(request_rate,
        nb_intervalle, taille_intervalle, gamma, k, alpha_de_sarsa)
        liste_cout_moyen=[]
        i=0
        while (i<len(cout_du_sarsa)): #tous 10 intervalles
            liste_cout_moyen.append(
            (fa.somme_liste(cout_du_sarsa[i : i+10]) / 10.0))
            i += 10
        plt.plot(range(len(liste_cout_moyen)), liste_cout_moyen,
        ".", label = str(k))
    plt.title('Test de epsilon' )
    plt.xlabel('Nombre d\'it rations')
    plt.ylabel('Cout')
    plt.grid('on')
    plt.legend(loc = "best")
    plt.savefig('fig8.pdf')
    plt.close()
    plt.show()


def tests_de_alpha(request_rate, nb_intervalle, taille_intervalle):
    liste_alpha = [0.1, 0.3, 0.5, 0.9]
```

```
for k in liste_alpha:
    cout_du_sarsa=tests_sarsa_pour_3(request_rate,
    nb_intervalle, taille_intervalle, gamma, epsilon, k)
    liste_cout_moyen=[]
    i=0
    while (i<len(cout_du_sarsa)): #tous 10 intervalles
        liste_cout_moyen.append((
        fa.somme_liste(cout_du_sarsa[i : i+10]) / 10.0))
        i += 10
    plt.plot(range(len(liste_cout_moyen)), liste_cout_moyen,
    ".", label = str(k))
plt.title('Test de alpha de sarsa' )
plt.xlabel('Nombre d\'iterations')
plt.ylabel('Cout')
plt.grid('on')
plt.legend(loc = "best")
plt.savefig('fig1.pdf')
plt.close()
plt.show()
```

Annexe B

Analyse du code

B.1 CodeCassiopee

init : Variables globales du codes utilisées dans les autres fonctions.

zipf_distribution (alpha, nb_videos, norme) : Elle retourne le tableau des probabilités d'avoir une vidéo donnée pour un CP fixé, en suivant la loi de distribution de Zipf qui est la loi la plus adaptée à la simulation d'une répartition standard des vidéos sur un CP donné. Dans le tableau, les vidéos sont triées par ordre décroissant de popularité suivant une loi de Zipf avec un alpha passé en paramètre. Les probabilités sont obtenues divisant par la norme qui est entrée en paramètre pour alléger la charge de calcul.

request_creation () : Cette fonction permet de simuler la génération d'une requête aléatoire : elle utilise pour cela la fonction random qui renvoie une probabilité (P) comprise entre 0 et 1. Or, la somme des probabilités (P(i)) dans la Liste des probabilités de choisir un CP donné étant égale à 1, on parcourt ainsi la liste en additionnant la probabilité cumulée jusqu'à atteindre le nombre pioché au hasard entre 0 et 1. Le CP auquel on a dépassé P est donc le CP que la requête a sélectionné. On effectue de même pour sélectionner une vidéo dans un CP donné mais cette fois suivant la loi de Zipf avec le alpha et le catalogue du CP correspondant. Cet algorithme permet de garantir que la probabilité de choisir une vidéo donnée est la même que sa probabilité d'apparition selon la distribution de Zipf.

decide_naive_alloc () : Allocation naïve de la mémoire cache entre les CP : allocation de mémoire cache proportionnelle au nombre de vidéos proposées du CP.

decide_opt_alloc () : Allocation de la mémoire cache qui tient compte de la distribution de Zipf, des probabilités de sélection de chaque CP et du fait que, d'un CP à l'autre, alpha a une valeur différente. On compare les vidéos les plus populaires de chaque CP entre elles, on sélectionne celle qui a la probabilité la plus élevée et on déplace le curseur du CP choisie vers nouvelle vidéo la plus populaire (la suivante) et ainsi de suite.

evaluate_cout (allocation, nb_requetes) : A partir d'une allocation et d'un nombre de requête, cette fonction va générer des requêtes une à une et tester si la vidéo demandée est dans le cache. Si elle n'y est pas, le coût augmente de 1.

states_2CP(cache_capacity) : Renvoie la liste des 101 états possibles pour 2CPs grâce à une simple boucle for.

states_3CP(cache_capacity) : Renvoie la liste des 5151 états possibles pour 3CPs. Grâce à une double boucle, on peut appeler states_2CP en ''retirant" une quantité de l'allocation donnée aux 2CPs et la ''donnant" au 3éme CP.

position_etat(alloc) : Renvoie la position correspondant à une allocation dans la liste des allocations possibles des 3 CPs.

sarsa_pour_3(request_rate, nb_intervalles, taille_intervalle) : Cette fonction est la plus importante de tout notre code. Nous allons donc la présenter un peu plus détail. En réalité, cette fonction a besoin d'autres paramètres notamment alpha et gamma dans la formule de SARSA ou encore epsilon pour la politique epsilon-greedy. Cependant, nous avons déclaré ces variables comme des variables globales pour alléger le code et simplifier les tests.

sarsa_pour_3_bis(request_rate, intervalle) : Cette fonction est seulement utile pour montrer notre utilisation du module time de python pour déterminer les lignes de calcul gourmande en temps

tests_sarsa_pour_3(_rate, nb_intervalle, taille_intervalle, gama, epsi, alfa) : Fonction identique à sarsa_pour_3 sauf qu'on a les 3 variables que l'on va modifier en tant que paramètres. Les noms de ces variables sont intentionnellement mal orthographiés pour bien les différencier des variables globales et ainsi éviter les erreur.

tests_de_gamma(request_rate, nb_intervalle, taille_intervalle), tests_de_epsilon(request_rate, nb_intervalle, taille_intervalle), tests_de_alpha(request_rate, nb_intervalle, taille_intervalle) : Ces 3 fonctions permettent de faire les tests utilisés dans la partie 3.2 et 3.3 de notre étude. Lorsqu'on étudie un paramètre, disons alpha pour l'exemple, les deux autres variables epsilon et gamma sont fixées en les appelant par leur nom de variable globale.

B.2 Generator

liste_des_proba(k) : Pour k CP cette fonction leur associe des probabilités d'être choisi en prenant des nombres aléatoires entre 0 et 1, puis on calcule la somme de ces nombres ce qui permet de normaliser les valeurs afin qu'on obtienne bien une somme totale des probabilités égale à 1. La fonction renvoie le tableau de longueur k dont chaque valeur correspond à la probabilité normalisée pour chaque CP.

List_proba_uniforme(k) : Idem, mais cette fois les probabilités sont choisies suivant une loi uniforme et non plus aléatoirement.

List_des_alphas(k) : Renvoie le tableau de longueur k de la valeur de alpha pour chaque CP (fixée à 0.8).

List_alpha_seed1/2/3(k) : Idem, mais cette fois alpha est choisi aléatoirement entre 3 valeurs (0.8, 1.0 et 1.2) suivant une loi uniforme pour les k CP.

List_nombre_de_video(k) : Renvoie le tableau de longueur k du nombre de vidéos pour chaque CP (fixé à 1000).

List_100/1000/10000_videos(k) : Idem, avec des nombres de vidéos fixés respectivement à 100, 1000 et 10 000.

B.3 Fonctions auxiliaires

somme_liste(liste) : Calcule la somme de l'ensemble des valeurs contenues dans la liste.

trouver_max_col (reward, index) : Cette fonction permet, avec tous les cas d'égalité explicités, de trouver la maximum d'une colonne (ici du vecteur reward[index]) pour 2 CPs, et de retourner sa valeur ainsi que l'allocation associé.

recherche_max(vec) : Retourne l'indice de la valeur maximale d'un vecteur qui représente une ligne dans la matrice de récompense : on cherche la décision qui maximise la récompense.

B.4 Tests basiques

zipf_Distribution (alpha, nb_videos) : Renvoie le tableau des probabilités de choisir une vidéo donnée dans un CP donné (qui est caractérisé par un nombre de vidéos et un alpha). Chaque case d'indice i du tableau prend la valeur de la i-ème vidéo la plus populaire (qui a la probabilité la plus élevée d'être sélectionné).

Request_Creation (proba_yt, alpha_yt, alpha_nf, nb_videos_yt, nb$_videos\_nf$) :
Cette fonction centrale permet de simuler un choix aléatoire de visionnage d'une vidéo par un internaute en tenant compte des popularités de chaque vidéo. 2 fonctions random permettent de choisir le CP puis de choisir une vidéo parmi les vidéos du CP choisi. La variable proba_yt détermine si une requête est dans YouTube (si random () $<$ proba_yt) ou Netflix (si random () $>$ proba_yt). Dans les 2 cas, on utilise le tableau des probabilités défini par la fonction Zipf_Distribution (alpha, nb_videos) : on parcourt le tableau jusqu'à ce que la somme des probabilités cumulée dépasse la valeur renvoyée par random (). On quitte alors la boucle, et l'indice auquel on s'est arrêté (le plus grand i tel que somme $<$ random ()) correspond à la vidéo sélectionnée. Le résultat en sortie de la fonction consiste en un tableau de longueur 2 avec l'indice du CP et l'indice de la vidéo dans le CP.

Decide_naive_alloc (nb_videos_yt, nb_videos_nf, cache_capacity) : Cette fonction permet de répartir la mémoire cache entre les différents CP. Celle-ci fait une affectation dite « naïve », elle alloue le cache de manière proportionnelle au nombre de vidéos de chaque CP. Elle retourne un tableau de longueur 2 qui représente l'allocation de mémoire pour chacun des 2 CP (YouTube et Netflix).

Decide_Opt_Alloc (proba_yt, alpha_yt, alpha_nf, nb_videos_yt, nb_videos_nf, cache_capacity) : Idem, mais désormais on reprend les distributions de Zipf pour YouTube et Netflix pour déterminer quelles seront les vidéos contenues dans la mémoire cache. Les probabilités sont pondérées par les probabilités de sélectionner le CP de la vidéo (Netflix ou YouTube ici, pour k = 2). Puis on parcourt les 2 listes en comparant à chaque tour de boucle les probabilités des 2 premiers indices : on choisit toujours celui avec la probabilité la plus élevée et enfin on le retire de la liste... on réitère l'opération jusqu'à occupation totale de la mémoire cache.

Evaluate_Cout (allocation, proba_yt, alpha_yt, alpha_nf, nb_videos_yt, nb_videos_nf, nb_requetes) : Fonction permettant la synthèse des 2 fonctions précédentes : on calcule, pour une allocation de mémoire cache donnée (répartie entre les CP), le coût total d'une série de requêtes (nombre de requêtes défini en entrée), ce qui permet d'en déduire le coût moyen par requête et donc d'évaluer l'efficacité d'une allocation. A chaque requête générée aléatoirement par la fonction Request_Creation, on ajoute +1 au coût si la vidéo demandée n'est pas contenue dans la mémoire cache, mais rien dans le cas contraire. Plus l'allocation fait souvent appel à la Bande Passante (BP) à la place de la Mémoire Cache (MC), moins elle est optimale.

Main_tests () : Tests graphiques de différentes allocations en faisant varier respectivement alpha, puis le nombre de requêtes et les probabilités de chaque CP. La modélisation graphique sert à suivre le courbe d'évolution du coût en fonction de ces paramètres.

Annexe C

Value Approximator

L'idée de cette approche est bien expliquée par David Silver.

À la place d'utiliser une table pour stocker toutes les valeurs de $q(\mathbf{a}, \mathbf{s})$, on va approximer la valeur de $q(\mathbf{a}, \mathbf{s})$ pour chaque paire $(\mathbf{a}, \mathbf{s})$ avec une fonction $q_{\mathbf{w}}(\mathbf{a}, \mathbf{s})$ paramétrée par le vecteur $\mathbf{w}$. Pour l'instant on va choisir une approximation linéaire :

$$q_{\mathbf{w}}(\mathbf{a}, \mathbf{s}) = \mathbf{w} \cdot (\mathbf{a} + \mathbf{s})$$

où $\mathbf{w}$ est un vecteur de la même dimension de $\mathbf{a}$ et $\mathbf{s}$ (cette dimension est le nombre de fournisseurs de contenu).

L'algorithme de reinforcement learning est très proche de ce qui a été expliqué avant. Sauf que l'on ne met pas à jour une table, mais on met à jour le paramètre $\mathbf{w}$, et ce faisant on met à jour la fonction $q_{\mathbf{w}}$:

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

où

$$\Delta\mathbf{w} = \alpha \cdot (U(\mathbf{a}_t, \mathbf{s}_t) - q_{\mathbf{w}}(\mathbf{a}_t, \mathbf{s}_t)) \cdot \mathbf{w}$$

où $U(\mathbf{a}_t, \mathbf{s}_t$ est l'estimation mise à jour de la récompense totale (jusqu'à la fin du monde) qu'on obtient si on se trouve à l'état $\mathbf{s}_t$ et on prend l'action $\mathbf{a}_t$. Cette estimation est $U(\mathbf{a}_t, \mathbf{s}_t) = r_{t+1} + q_{\mathbf{w}}(\mathbf{a}_{t+1}, \mathbf{s}_{t+1}))$, où $\mathbf{s}_{t+1}$ est l'état ou on se retrouve après avoir pris l'action, r_{t+1} est la récompense instantanée qu'on a obtenu après qu'on a pris l'action et $\mathbf{a}_{t+1}$ et l'action que la politique nous fait prendre quand on se retrouve à l'état s_{t+1}. On a appelé $U(\mathbf{a}_t, \mathbf{s}_t$ estimation "mise à jour" parce qu'on ne peut la calculer qu'après avoir pris l'action $\mathbf{a}_t$.

Annexe D

Bibliographie

[1] Article de présentation du problème intitulé "*Caching Encrypted Content via Stochastic Cache Partitioning*" - Andrea Araldo.

[2] Cours en ligne de Reinforcement Learning par David Silver - disponible sur **https ://www. youtube.com/playlist ?list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9-** .

[3] Polycopié du module "*Apprentissage, classification automatique, data-mining*" - Telecom SudParis.

[4] R. S. Sutton et A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press,2017.