Introduction_aux_algorithmes_de_bandit__comme_UCB1_et_Thompso

January 10, 2018

1 Table of Contents

2 Introduction aux *algorithmes de bandit* (UCB1, Thompson Sampling)

Ce petit document est un notebook Jupyter, ayant pour but de présenter le concept de problèmes de bandit, comment les simuler et les résoudre, et deux algorithmes conçus dans ce but.

Je ne vais pas donner beaucoup d'explications mathématiques, je conseille plutôt ce petit article, datant de 2017, en français, écrit par Émilie Kaufmann.

Je préfère me focaliser sur une implémentation simple, claire et concise de chaque morceau nécessaire à la simulation de problèmes et d'algorithmes de bandit. J'utilise le langage de programmation Python.

Dans ce but, j'utilise une approche *objet* : chaque morceau sera une classe, et des *instances* (des *objets*) seront utilisées pour toutes les composantes.

```
In [1]: # Dépendances
        import numpy as np
        import random as rd
```

```
In [2]: import matplotlib as mpl
        import matplotlib.pyplot as plt
        import seaborn as sns
        sns.set(context="notebook", style="whitegrid", palette="hls", font="sans-serif", font_s
        from tqdm import tqdm_notebook as tqdm
```

```
In [3]: %load_ext watermark
        %watermark -v -m -a "Lilian Besson (Naereen)" -p scipy,numpy,matplotlib,sympy,seaborn -
```

```
Lilian Besson (Naereen)

CPython 3.6.3
IPython 6.2.1

scipy 1.0.0
numpy 1.14.0
matplotlib 2.1.1
sympy 1.1.1
seaborn 0.8.1

compiler   : GCC 7.2.0
system     : Linux
release    : 4.13.0-21-generic
machine    : x86_64
processor  : x86_64
CPU cores  : 4
interpreter: 64bit
Git hash   :
```

2.1 Problèmes de bandit

Je ne rentrerai pas trop dans les détails ici. Pour plus d'explications, cette page Wikipédia (en français) est très bien faite.

> "Dans le problème dit du bandit manchot, un utilisateur fait face à $K \geq 2$ machines à sou. Chacune donnant une récompense moyenne que l'utilisateur ne connait pas à priori. A chacune de ces actions, il va donc sélectionner une machine permettant de maximiser son gain."

2.1.1 Problèmes stochastiques

On se focalise ici sur des problèmes dits stochastiques : pour chaque bras, $k \in \{1, \dots, K\}$, une distribution de probabilité ν_k est supposée générer les récompense à chaque sélection. Les récompenses $r_k(t)$ venant du $k$ième bras sont donc "i.i.d." : indépendentes (les unes des autres) et identiquement distribuées, selon ν_k :

$$\forall t \geq 1, \quad r_k(t) \sim \nu_k.$$

Un des exemples les plus simples sera de considérer des lois de Bernoulli, avec des *récompenses binaires*, par exemple pour un traitement médical $r_k(t) = 0$ signifie que le traitement a échoué, et $r_k(t) = 1$ que le traitement a réussi à guérir telle maladie :

$$\forall t \geq 1, r_k(t) \in \{0, 1\}, \text{ et } r_k(t) \sim \mathrm{B}(\mu_k).$$

2.1.2 Radio intelligente ?

Dans le contexte de la radio intelligente, le vocabulaire change un peu :

- les bras sont des *canaux radio*,
- les joueurs sont des *objets communiquants*,
- et des *récompenses binaires* sont obtenues soit avant la communication (avec une écoute du spectre, "sensing", pour l'Accès Opportuniste au Spectre, "OSA"), soit après la communication (avec un "acknowledgement" reçu depuis une station de base),
- ou bien des *récompenses réelles*, reflétant par exemple le SNR de la communication, le débit, la puissance reçue etc.

2.2 Simulation de problèmes de bandit

Une simple *fonction* suffira ici, pour initialiser un algorithme, le simuler durant T étapes, et stocker les récompenses et les bras tirés.

Il est important de tout stocker pour ensuite pouvoir afficher différentes statistiques sur l'expérience, permettant d'évaluer l'efficacité des différentes algorithmes.

Je préfère donner directement cette fonction afin de fixer les *signatures* des différentes classes qu'on va écrire ensuite :

- Les *bras* ont besoin d'une seule méthode, `tire()` qui donne $r_k(t) \sim \nu_k$ à l'instant t pour la distribution ν_k du $k^{\text{ième}}$ bras, noté `r_k_t`.
- Les *algorithmes* ont besoin de trois méthodes :
  - `commence()` pour initialiser l'algorithme, une fois,
  - `A_t = choix()` pour choisir un bras, à chaque instant t, noté $A(t) \in \{1, \dots, K\}$,
  - `recompense(A_t, r_k_t)` pour donner la récompense `r_k_t` tirée du bras `A_t`.

```
In [4]: def simulation(bras, algorithme, horizon):
            """ Simule l'algorithme donné sur ces bras, durant horizon étapes."""
            choix, recompenses = np.zeros(horizon), np.zeros(horizon)
            # 1. Initialise l'algorithme
```

```
    algorithme.commence()
    # 2. Boucle en temps, pour t = 0 à horizon - 1
    for t in range(horizon):
        # 2.a. L'algorithme choisi son bras à essayer
        A_t = algorithme.choix()
        # 2.b. Le bras k donne une récompense
        r_k_t = bras[A_t].tire()
        # 2.c. La récompense est donnée à l'algorithme
        algorithme.recompense(A_t, r_k_t)
        # 2.d. On stocke les deux
        choix[t] = A_t
        recompenses[t] = r_k_t
    # 3. On termine en renvoyant ces deux vecteurs
    return recompenses, choix
```

2.3 Bras stochastiques, de Bernoulli

Les récompenses de tels bras, notées $r_k(t)$ pour le bras k à l'instant t, sont tirées de façons identiquement distribuées et indépendantes, selon une loi de Bernoulli :

$$\forall t \in \mathbb{N}, \forall k \in \{1, \dots, K\}, r_k(t) \in \{0, 1\}, \text{ et } r_k(t) \sim \mathrm{B}(\mu_k).$$

```
In [5]: class Bernoulli():
            """ Bras distribués selon une loi de Bernoulli."""

            def __init__(self, probabilite):
                assert 0 <= probabilite <= 1, "Erreur, probabilite doit être entre 0 et 1 pour
                self.probabilite = probabilite

            def tire(self):
                """ Tire une récompense aléatoire."""
                return float(rd.random() <= self.probabilite)
```

Par exemple, on peut considérer le problème à trois bras ($K = 3$), caractérisé par ces paramètres $\boldsymbol{\mu} = [\mu_1, \dots, \mu_K] = [0.1, 0.5, 0.9]$:

```
In [6]: mus = [ 0.1, 0.5, 0.9 ]
        bras = [ Bernoulli(mu) for mu in mus ]
```

On peut prendre 10 échantillons de chaque bras, et vérifier leurs moyennes :

```
In [7]: rd.seed(10000)  # pour être reproductible.
        T = 10
        exemples_echantillons = [ [ bras_k.tire() for _ in range(T) ] for bras_k in bras ]
        exemples_echantillons

Out[7]: [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
         [1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0],
         [0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]]
```

```
In [8]: np.mean(exemples_echantillons, axis=1)

Out[8]: array([0.1, 0.5, 0.9])
```

C'est assez proche de $\boldsymbol{\mu} = [\mu_1, \dots, \mu_K] = [0.1, 0.5, 0.9]$...

2.4 Présentation des algorithmes de bandit

Comme on l'a dit plus haut, les *algorithmes* ont besoin de trois méthodes :

- `commence()` pour initialiser l'algorithme, une fois. Généralement, il s'agit de remettre à zero les vecteurs de mémoires internes de l'algorithme, et de mettre $t = 0$.
- `A_t = choix()` pour choisir un bras, à chaque instant t, noté $A(t) \in \{1, \dots, K\}$. C'est la partie "intelligente" qui doit être conçue avec soin.
- `recompense(A_t, r_k_t)` pour donner la récompense `r_k_t` tirée du bras `A_t`. Souvent, il suffit de mettre à jour les deux ou trois vecteurs internes.

En fait, il faut aussi une méthode pour *créer* l'instance de la classe, i.e., une méthode `__init__(K)`, qui demande de simplement connaître K, le nombre de bras.

> Bien-sûr, les algorithmes ne doivent pas connaître $\boldsymbol{\mu} = [\mu_1, \dots, \mu_K]$ les paramètres du problème... Sinon l'apprentissage n'a aucun intérêt : il suffit de viser $k^* = \arg\max_k \mu_k$...

- Lisez ce paragraphe sur la page Wikipédia précédemment citée, il est assez instructif.

2.5 Deux algorithmes naïfs

On va commencer par donner deux exemples naïfs :

1. Un algorithme "stupide" qui choisi un bras de façon complètement uniforme, $A^1(t) \sim U(1, \dots, K), \forall t$, à chaque instant $t \in \mathbb{N}$, via la classe `ChoixUniforme`.
2. Un algorithme moins stupide, mais assez naïf, qui utilise un *estimateur empirique* $\widehat{\mu_k}(t) = \frac{X_k(t)}{N_k(t)}$ de la *moyenne* de chaque bras, et tire $A^2(t) \in \arg\max_k \widehat{\mu_k}(t)$ à chaque instant $t \in \mathbb{N}$. Ici, $X_k(t) = \sum_{\tau=0}^{t} \mathbb{1}(A(\tau) = k) r_k(\tau)$ compte les récompenses *accumulées* en tirant le bras k, sur les instants $t = 0, \dots, \tau$. Et $N_k(t) = \sum_{\tau=0}^{t} \mathbb{1}(A(\tau) = k)$ compte le nombre de sélections de ce bras k. Via la classe `MoyenneEmpirique`.

2.5.1 `ChoixUniforme`

C'est l'algorithme le plus naïf : à chaque instant, le bras choisi est tiré uniformément dans $\{0, \dots, K-1\}$ (notation Python, on indice à partir de 0).

```
In [9]: class ChoixUniforme(object):
            """Algorithme stupide, choix uniforme."""

            def __init__(self, K):
                """Crée l'instance de l'algorithme."""
                self.K = K

            def commence(self):
                """Initialise l'algorithme : rien à faire ici."""
                pass

            def choix(self):
                """Choix uniforme d'un indice A(t) ~ U(1...K)."""
                return rd.randint(0, self.K - 1)

            def recompense(self, k, r):
                """Donne une récompense r tirée sur le bras k à l'algorithme : rien à faire ic
                pass
```

2.5.2 MoyenneEmpirique

Voilà qui donne une bonne idée de la structure que vont devoir suivre les différentes algorithmes.

L'algorithme suivant est un peu plus complexe. Il est aussi appelé "Follow the Leader" (FTL).

```
In [10]: class MoyenneEmpirique(object):
             """Algorithme naïf, qui utilise la moyenne empirique."""

             def __init__(self, K):
                 """Crée l'instance de l'algorithme."""
                 self.K = K
                 # Il nous faut de la mémoire interne
                 self.recompenses = np.zeros(K)  # X_k(t) pour chaque k
                 self.tirages = np.zeros(K)      # N_k(t) pour chaque k
                 self.t = 0                      # Temps t interne

             def commence(self):
                 """Initialise l'algorithme : remet à zeros chaque X_k et N_k, et t = 0."""
                 self.recompenses.fill(0)
                 self.tirages.fill(0)
                 self.t = 0

             def choix(self):
                 """Si on a vu tous les bras, on prend celui de moyenne empirique la plus gran
                 # 1er cas : il y a encore des bras qu'on a jamais vu
                 if np.min(self.tirages) == 0:
                     k = np.min(np.where(self.tirages == 0)[0])
                 # 2nd cas : tous les bras ont été essayé
                 else:
```

```
            # Notez qu'on aurait pu ne stocker que ce vecteur moyennes_empiriques
            moyennes_empiriques = self.recompenses / self.tirages
            k = np.argmax(moyennes_empiriques)
        self.t += 1       # Inutile ici
        return k

    def recompense(self, k, r):
        """Donne une récompense r tirée sur le bras k à l'algorithme : met à jour les
        self.recompenses[k] += r
        self.tirages[k] += 1
```

2.6 Approche fréquentiste, UCB1, "Upper Confidence Bound"

Il s'agit d'une amélioration de l'algorithme précédent, où on utilise un autre *indice*.

Au lieu d'utiliser la moyenne empirique $g_k(t) = \widehat{\mu}_k(t) = \frac{X_k(t)}{N_k(t)}$ et $A(t) = \arg\max_k g_k(t)$, on utilise une borne supérieure d'un intervalle de confiance autour de cette moyenne :

$$g'_k(t) = \widehat{\mu}_k(t) + \sqrt{\alpha \frac{\log t}{N_k(t)}}.$$

Et cet *indice* est toujours utilisé pour décider le bras à essayer à chaque instant :

$$A^{\text{UCB1}}(t) = \arg\max_k g'_k(t).$$

Il faut une constante $\alpha \geq 0$, qu'on choisira $\alpha \geq \frac{1}{2}$ pour avoir des performances raisonnables. α contrôle le compromis entre *exploitation* et *exploration*, et ne doit pas être trop grand. $\alpha = 1$ est un bon choix par défaut.

> On va gagner du temps en *héritant* de la classe `MoyenneEmpirique` précédente. Ça permet de ne pas réécrire les méthodes qui sont déjà bien écrites.

```
In [11]: class UCB1(MoyenneEmpirique):
             """Algorithme UCB1."""

             def __init__(self, K, alpha=1):
                 """Crée l'instance de l'algorithme. Par défaut, alpha=1."""
                 super(UCB1, self).__init__(K)  # On laisse la classe mère faire le travaille
                 assert alpha >= 0, "Erreur : alpha doit etre >= 0."
                 self.alpha = alpha

             def choix(self):
                 """Si on a vu tous les bras, on prend celui d'indice moyenne empirique + UCB
                 self.t += 1       # Nécessaire ici
                 # 1er cas : il y a encore des bras qu'on a jamais vu
                 if np.min(self.tirages) == 0:
                     k = np.min(np.where(self.tirages == 0)[0])
```

```
        # 2nd cas : tous les bras ont été essayé
        else:
            moyennes_empiriques = self.recompenses / self.tirages
            ucb = np.sqrt(self.alpha * np.log(self.t) / self.tirages)
            indices = moyennes_empiriques + ucb
            k = np.argmax(indices)
        return k
```

2.7 Variantes d'UCB : UCB-V et UCB-H

On peut rapidement implémenter deux variantes d'UCB, UCB-V qui utilise les carrés des récompenses pour estimer la variance de chaque bras et calculer un meilleur intervalle de confiance, et UCB-H qui utilise la connaissance de l'horizon T de l'expérience pour utiliser $\sqrt{\log(T)/N_k(t)}$ à la place de $\sqrt{\log(t)/N_k(t)}$ et obtenir de meilleures performances.

```
In [33]: class UCBV(MoyenneEmpirique):
             """Algorithme UCBV."""

             def __init__(self, K, alpha=1):
                 """Crée l'instance de l'algorithme. Par défaut, alpha=1."""
                 super(UCBV, self).__init__(K)  # On laisse la classe mère faire le travaille
                 assert alpha >= 0, "Erreur : alpha doit etre >= 0."
                 self.alpha = alpha
                 self.recompensesCarrees = np.zeros(K)  # somme des r_k(t)^2 pour chaque k

             def recompense(self, k, r):
                 """Donne une récompense r tirée sur le bras k à l'algorithme : met à jour les
                 self.recompenses[k] += r
                 self.recompensesCarrees[k] += r ** 2
                 self.tirages[k] += 1

             def choix(self):
                 """Si on a vu tous les bras, on prend celui d'indice moyenne empirique + UCB
                 self.t += 1       # Nécessaire ici
                 # 1er cas : il y a encore des bras qu'on a jamais vu
                 if np.min(self.tirages) == 0:
                     k = np.min(np.where(self.tirages == 0)[0])
                 # 2nd cas : tous les bras ont été essayé
                 else:
                     moyennes_empiriques = self.recompenses / self.tirages
                     variance = (self.recompensesCarrees / self.tirages) - moyennes_empiriques

                     ucb = np.sqrt(self.alpha * np.log(self.t) * variance / self.tirages) + 3.(

                     indices = moyennes_empiriques + ucb
                     k = np.argmax(indices)
                 return k
```

Et pour UCBH :

```
In [13]: class UCBH(MoyenneEmpirique):
             """Algorithme UCBH."""

             def __init__(self, K, horizon, alpha=1):
                 """Crée l'instance de l'algorithme. Par défaut, alpha=1."""
                 super(UCBH, self).__init__(K)  # On laisse la classe mère faire le travaille
                 self.horizon = int(horizon)
                 assert alpha >= 0, "Erreur : alpha doit etre >= 0."
                 self.alpha = alpha

             def choix(self):
                 """Si on a vu tous les bras, on prend celui d'indice moyenne empirique + UCB
                 self.t += 1      # Nécessaire ici
                 # 1er cas : il y a encore des bras qu'on a jamais vu
                 if np.min(self.tirages) == 0:
                     k = np.min(np.where(self.tirages == 0)[0])
                 # 2nd cas : tous les bras ont été essayé
                 else:
                     moyennes_empiriques = self.recompenses / self.tirages
                     ucb = np.sqrt(self.alpha * np.log(self.horizon) / self.tirages)
                     indices = moyennes_empiriques + ucb
                     k = np.argmax(indices)
                 return k
```

2.8 Approche fréquentiste optimale, KL-UCB, "Kullback-Leibler UCB"

2.8.1 KL binaire

Pour $x, y \in \{0,1\}$, $x, y \neq 0, 1$, $\mathrm{kl}(x,y)$, on définit la divergence de Kullback-Leibler binaire de x et y comme la KL de deux lois de Bernoulli de moyennes x et y, ce qui est défini comme :

$$\mathrm{kl}(x,y) := x \log\left(\frac{x}{y}\right) + (1-x)\log\left(\frac{1-x}{1-y}\right).$$

Pour vous facilier la tâche, la fonction kl est déjà implémentée, ainsi qu'une fonction pour résoudre (de façon approchée) le probl_me d'optimisation contrainte qui définit l'indice $g''_k(t)$.

```
In [14]: # Just forcing the ?? in Jupyter to be in the main document (to be saved) and not a f
         # Thanks to https://nbviewer.jupyter.org/gist/minrk/7715212
         from __future__ import print_function
         from IPython.core import page
         def myprint(s):
             try:
                 print(s['text/plain'])
             except (KeyError, TypeError):
                 print(s)
         page.page = myprint
```

```
In [15]: from kullback import klBern
         klBern?

Signature: klBern(x, y)
Docstring:
Kullback-Leibler divergence for Bernoulli distributions. https://en.wikipedia.org/wiki/Bernoull

>>> klBern(0.5, 0.5)
0.0
>>> klBern(0.1, 0.9)  # doctest: +ELLIPSIS
1.757779...
>>> klBern(0.9, 0.1)  # And this KL is symetric  # doctest: +ELLIPSIS
1.757779...
>>> klBern(0.4, 0.5)  # doctest: +ELLIPSIS
0.020135...
>>> klBern(0.01, 0.99)  # doctest: +ELLIPSIS
4.503217...

- Special values:

>>> klBern(0, 1)  # Should be +inf, but 0 --> eps, 1 --> 1 - eps  # doctest: +ELLIPSIS
34.539575...
File:      ~/Bureau/Python_code_TP_Odalric/kullback.py
Type:      function
```

2.8.2 KLUCB

(Garivier & Cappé - COLT, 2011)

Il s'agit d'une autre amélioration de l'algorithme précédent, où on utilise un autre *indice*.

Au lieu d'utiliser une borne supérieure d'un intervalle de confiance autour de cette moyenne, on utilise la pseudo-distance de Kullback-Leibler afin d'obtenir l'intervale de confiance optimal :

$$g''_k(t) = \sup_{q \in [0,1]} \left\{ q : \mathrm{kl}(\hat{\mu}_k(t), q) \leq \frac{f(t)}{N_k(t)} \right\}.$$

Et cet *indice* est toujours utilisé pour décider le bras à essayer à chaque instant :

$$A^{\mathrm{KLUCB}}(t) = \arg\max_k g''_k(t).$$

On utilise

$$f(t) := \log(t) + c \log\log(t).$$

Il faut une constante $c \geq 0$, qu'on doit choisir $c > 0$ pour simplifier les preuves théoriques. En pratique, $c = 0$ est un bon choix par défaut.

```
In [16]: from kullback import klucb
         klucb?
         # do klucb?? to see the code
```

```
Signature: klucb(x, d, kl, upperbound, lowerbound=-inf, precision=1e-06)
Docstring:
The generic KL-UCB index computation.

- x: value of the cum reward,
- d: upper bound on the divergence,
- kl: the KL divergence to be used (klBern, klGauss, etc),
- upperbound, lowerbound=float('-inf'): the known bound of the values x,
- precision=1e-6: the threshold from where to stop the research,


.. note:: It uses a bisection search.
File:      ~/Bureau/Python_code_TP_Odalric/kullback.py
Type:      function
```

```
In [17]: from kullback import klucbBern
         klucbBern?
         # do klucbBern?? to see the code

Signature: klucbBern(x, d, precision=1e-06)
Docstring:
KL-UCB index computation for Bernoulli distributions, using :func:`klucb`.

- Influence of x:

>>> klucbBern(0.1, 0.2)  # doctest: +ELLIPSIS
0.378391...
>>> klucbBern(0.5, 0.2)  # doctest: +ELLIPSIS
0.787088...
>>> klucbBern(0.9, 0.2)  # doctest: +ELLIPSIS
0.994489...

- Influence of d:

>>> klucbBern(0.1, 0.4)  # doctest: +ELLIPSIS
0.519475...
>>> klucbBern(0.1, 0.9)  # doctest: +ELLIPSIS
0.734714...

>>> klucbBern(0.5, 0.4)  # doctest: +ELLIPSIS
0.871035...
>>> klucbBern(0.5, 0.9)  # doctest: +ELLIPSIS
0.956809...

>>> klucbBern(0.9, 0.4)  # doctest: +ELLIPSIS
0.999285...
```

```
>>> klucbBern(0.9, 0.9)  # doctest: +ELLIPSIS
0.999995...
File:      ~/Bureau/Python_code_TP_Odalric/kullback.py
Type:      function
```

C'est assez facile avec tout ça :

```
In [19]: def f(t, c=0):
             return np.log(t) + c * np.log(np.maximum(0, np.log(t)))

         class KLUCB(MoyenneEmpirique):
             """Algorithme KLUCB."""

             def __init__(self, K, c=0, tolerance=1e-4):
                 """Crée l'instance de l'algorithme. Par défaut, c=0."""
                 super(KLUCB, self).__init__(K)  # On laisse la classe mère faire le travaille
                 assert c >= 0, "Erreur : c doit etre >= 0."
                 self.c = c
                 self.tolerance = tolerance
                 # Version vectorisée
                 self.klucb = np.vectorize(klucbBern)

             def choix(self):
                 """Si on a vu tous les bras, on prend celui d'indice KLUCB le plus grand."""
                 self.t += 1      # Nécessaire ici
                 # 1er cas : il y a encore des bras qu'on a jamais vu
                 if np.min(self.tirages) == 0:
                     k = np.min(np.where(self.tirages == 0)[0])
                 # 2nd cas : tous les bras ont été essayé
                 else:
                     indices = self.klucb(self.recompenses / self.tirages, f(self.t, self.c) /
                     k = np.argmax(indices)
                 return k
```

2.9 Approche bayésienne, Thompson Sampling

Ce petit article explique très bien l'approche bayésienne.

On a besoin de savoir manipuler des posteriors, qui seront les posteriors conjugués des distributions des bras.

Pour des bras de Bernoulli, le posterior conjugué associé est une loi Beta, notée $\mathrm{Beta}(\alpha, \beta)$ pour deux paramètres $\alpha, \beta > 0$.

- Les posteriors sont initialisés à $\mathrm{Beta}(1,1) = U([0,1])$, c'est-à-dire qu'on met un a priori uniforme sur les μ_k, comme on ne connaît que $\mu_k \in [0,1]$.
- Comme les *observations* sont binaires, $r_k(t) \in \{0,1\}$, les paramètres α,β restent entiers.

```
In [20]: from numpy.random import beta

         class Beta():
             """Posteriors d'expériences de Bernoulli."""

             def __init__(self):
                 self.N = [1, 1]

             def reinitialise(self):
                 self.N = [1, 1]

             def echantillon(self):
                 """Un échantillon aléatoire de ce posterior Beta."""
                 return beta(self.N[1], self.N[0])

             def observe(self, obs):
                 """Ajoute une nouvelle observation. Si 'obs'=1, augmente alpha, sinon si 'obs
                 self.N[int(obs)] += 1
```

Dès qu'on sait manipuler ces postériors Beta, on peut implémenter rapidement le dernier algorithme, Thompson Sampling.

Les paramètres du posterior sur μ_k, i.e., $\alpha_k(t)$,$\beta_k(t)$ seront mis à jour à chaque étape pour compter le nombre d'observations réussies et échouées :

$$\alpha_k(t) = 1 + X_k(t)\beta_k(t) = 1 + N_k(t) - X_k(t).$$

La moyenne empirique estimant μ_k sera, à l'instant t,

$$\widetilde{\mu_k}(t) = \frac{\alpha_k(t)}{\alpha_k(t) + \beta_k(t)} = \frac{1 + X_k(t)}{2 + N_k(t)} \simeq \frac{X_k(t)}{N_k(t)}.$$

La différence avec UCB1 est que la prise de décision de Thompson Sampling se fait sur un indice, tiré aléatoirement selon les posteriors. C'est une *politique d'indice randomisée*.

D'un point de vue bayésien, un *modèle* est tiré selon les posteriors, puis on joue selon le meilleur modèle :

$$g_k'''(t) \sim \mathrm{Beta}(\alpha_k(t), \beta_k(t)) A^{\mathrm{TS}}(t) = \arg\max_k g_k'''(t).$$

```
In [21]: class ThompsonSampling(MoyenneEmpirique):
             """Algorithme Thompson Sampling."""

             def __init__(self, K, posterior=Beta):
                 """Crée l'instance de l'algorithme. Par défaut, alpha=1."""
                 self.K = K
                 # On créé K posteriors
                 self.posteriors = [posterior() for k in range(K)]

             def commence(self):
                 """Réinitialise les K posteriors."""
                 for posterior in self.posteriors:
```

```
            posterior.reinitialise()

    def choix(self):
        """On tire K modèles depuis les posteriors, et on joue dans le meilleur."""
        moyennes_estimees = [posterior.echantillon() for posterior in self.posteriors]
        k = np.argmax(moyennes_estimees)
        return k

    def recompense(self, k, r):
        """Observe cette récompense r sur le bras k en mettant à jour le kième poster
        self.posteriors[k].observe(r)
```

2.10 Exemples de simulations

On va comparer, sur deux problèmes, les 4 algorithmes définis plus haut.

Les problèmes sont caractérisés par les moyennes des bras de Bernoulli, $\boldsymbol{\mu} = [\mu_1, \dots, \mu_K]$, et on les suppose ordonnées par ordre décroissant : $\mu_1 > \mu_2 \geq \cdots \geq \mu_K$.

On affichera plusieurs choses, dans des graphiques au cours du temps $t = 0, \dots, T$ pour un horizon $T = 1000$ ou $T = 5000$ étapes :

1. leurs *taux de sélection* du meilleur bras k^*, (qui sera toujours μ_1 le premier bras), i.e., $N_k(t)/t$ en %, pour chaque algorithme,
2. leurs *récompenses accumulées*, i.e., $R(t) = \sum_{\tau=0}^{t} \sum_{k=1}^{K} X_k(\tau) \mathbb{1}(A(t) = k)$, pour chaque algorithme,
3. les *récompenses moyennes*, i.e., $R(t)/t$, qui devrait converger vers $\mu^* = \mu_{k^*} = \mu_1$,
4. et enfin leurs *regret*. Cette notion est moins triviale, mais pour notre problème simple il se définit comme la perte, en récompenses accumulées, entre la meilleure stratégie (toujours sélectionner le meilleur bras $k^* = 1$) et la performance de l'algorithme :

$$\mathcal{R}(t) = \mu^* t - R(t)$$

On souhaite maximiser $R(t)$, donc minimiser $\mathcal{R}(t)$. Les algorithmes "efficaces" ont typiquement un regret *logarithmique*, i.e., $\mathcal{R}(T) = \mathcal{O}(\log T)$ *asymptotiquement*, ce qu'on souhaiterait vérifier.

2.10.1 Fonctions pour l'affichage

On définit 4 fonctions d'affichage pour ces quantités.

```
In [22]: mpl.rcParams['figure.figsize'] = (15, 8)
```

```
In [23]: def affiche_selections(choix, noms, kstar=0):
             plt.figure()
             for i, c in enumerate(choix):
                 selection_kstar = 1.0 * (c == kstar)
                 selection_moyenne = np.cumsum(selection_kstar) / np.cumsum(np.ones_like(c))
                 plt.plot(selection_moyenne, label=noms[i])
```

```
            plt.legend()
            plt.xlabel("Temps discret, $t = 1, ..., T = {}$".format(len(choix[0])))
            plt.ylabel("Taux de sélection")
            plt.title("Sélection du meilleur bras #{} pour différents algorithmes".format(1 +
            plt.show()
```

```
In [24]: def affiche_recompenses(recompenses, noms):
            plt.figure()
            for i, r in enumerate(recompenses):
                recompense_accumulee = np.cumsum(r)
                plt.plot(recompense_accumulee, label=noms[i])
            plt.legend()
            plt.xlabel("Temps discret, $t = 1, ..., T = {}$".format(len(recompenses[0])))
            plt.ylabel("Récompenses accumulées")
            plt.title("Récompenses accumulées pour différents algorithmes")
            plt.show()
```

```
In [25]: def affiche_recompenses_moyennes(recompenses, noms):
            plt.figure()
            for i, r in enumerate(recompenses):
                recompense_moyenne = np.cumsum(r) / np.cumsum(np.ones_like(r))
                plt.plot(recompense_moyenne, label=noms[i])
            plt.legend()
            plt.xlabel("Temps discret, $t = 1, ..., T = {}$".format(len(recompenses[0])))
            plt.ylabel(r"Récompenses moyennes $\in [0, 1]$")
            plt.title("Récompenses moyennes pour différents algorithmes")
            plt.show()
```

```
In [26]: def affiche_regret(recompenses, noms, mustar=1):
            plt.figure()
            for i, r in enumerate(recompenses):
                recompense_accumulee = np.cumsum(r)
                regret = mustar * np.cumsum(np.ones_like(r)) - recompense_accumulee
                plt.plot(regret, label=noms[i])
            plt.legend()
            plt.xlabel("Temps discret, $t = 1, ..., T = {}$".format(len(recompenses[0])))
            plt.ylabel("Regret")
            plt.title("Regret accumulé pour différents algorithmes")
            plt.show()
```

Pour afficher un histogramme, c'est moins évident, mais voici :

```
In [93]: def nrows_ncols(N):
            """(nrows, ncols) pour créer un subplots de N figures avec les bonnes dimensions.
            nrows = int(np.ceil(np.sqrt(N)))
            ncols = N // nrows
            while N > nrows * ncols:
                ncols += 1
            nrows, ncols = max(nrows, ncols), min(nrows, ncols)
            return nrows, ncols
```

```
In [94]: def affiche_hist_regret(recompenses, noms, horizon, mustar=1):
             nrows, ncols = nrows_ncols(len(noms))
             fig, axes = plt.subplots(nrows, ncols, sharex=False, sharey=False)
             fig.suptitle("Histogramme du regret à $t = T = {}$ pour différents algorithmes".fo

             # XXX See https://stackoverflow.com/a/36542971/
             ax0 = fig.add_subplot(111, frame_on=False)  # add a big axes, hide frame
             ax0.grid(False)  # hide grid
             ax0.tick_params(labelcolor='none', top='off', bottom='off', left='off', right='off
             # Add only once the ylabel, xlabel, in the middle
             ax0.set_ylabel("Distribution")
             ax0.set_xlabel("Regret")

             for i, r in enumerate(recompenses):
                 x, y = i % nrows, i // nrows
                 ax = axes[x, y] if ncols > 1 else axes[x]
                 regret = mustar * horizon - r
                 ax.hist(regret, normed=True, bins=25)
                 ax.set_title(noms[i])
             plt.show()
```

2.10.2 Premier problème, à 3 bras

On reprend le problème donné plus haut :

```
In [28]: horizon = 1000
         mus = [0.9, 0.5, 0.1]
         bras = [ Bernoulli(mu) for mu in mus ]
         K = len(mus)
         kstar = np.argmax(mus)  # = 0

In [29]: horizon, mus, bras, K, kstar

Out[29]: (1000,
          [0.9, 0.5, 0.1],
          [<__main__.Bernoulli at 0x7f1814c6ba90>,
           <__main__.Bernoulli at 0x7f1814c6b320>,
           <__main__.Bernoulli at 0x7f1814c6b1d0>],
          3,
          0)

In [34]: algorithmes = [ChoixUniforme(K), MoyenneEmpirique(K),
                        UCB1(K, alpha=1), UCBV(K, alpha=1), UCBH(K, horizon, alpha=1),
                        KLUCB(K), ThompsonSampling(K)]
         algorithmes

Out[34]: [<__main__.ChoixUniforme at 0x7f1814bfa390>,
          <__main__.MoyenneEmpirique at 0x7f1814bfa3c8>,
          <__main__.UCB1 at 0x7f1814bfa400>,
```

```
 <__main__.UCBV at 0x7f1814bfa438>,
 <__main__.UCBH at 0x7f1814bfa470>,
 <__main__.KLUCB at 0x7f1814bfa4a8>,
 <__main__.ThompsonSampling at 0x7f1814bfa518>]
```

Pour les légendes, on a besoin des noms des algorithmes :

```
In [35]: noms = ["ChoixUniforme", "MoyenneEmpirique",
                 "UCB1(alpha=1)", "UCBV(alpha=1)", "UCBH(alpha=1)",
                 "KLUCB", "ThompsonSampling"]
```

On peut commencer la simulation, pour chaque algorithme.

```
In [36]: %%time
         N = len(algorithmes)
         recompenses, choix = np.zeros((N, horizon)), np.zeros((N, horizon))

         for i, alg in tqdm(enumerate(algorithmes), desc="Algorithmes"):
             rec, ch = simulation(bras, alg, horizon)
             recompenses[i] = rec
             choix[i] = ch
```

```
HBox(children=(IntProgress(value=1, bar_style='info', description='Algorithmes', max=1), HTML(v
```

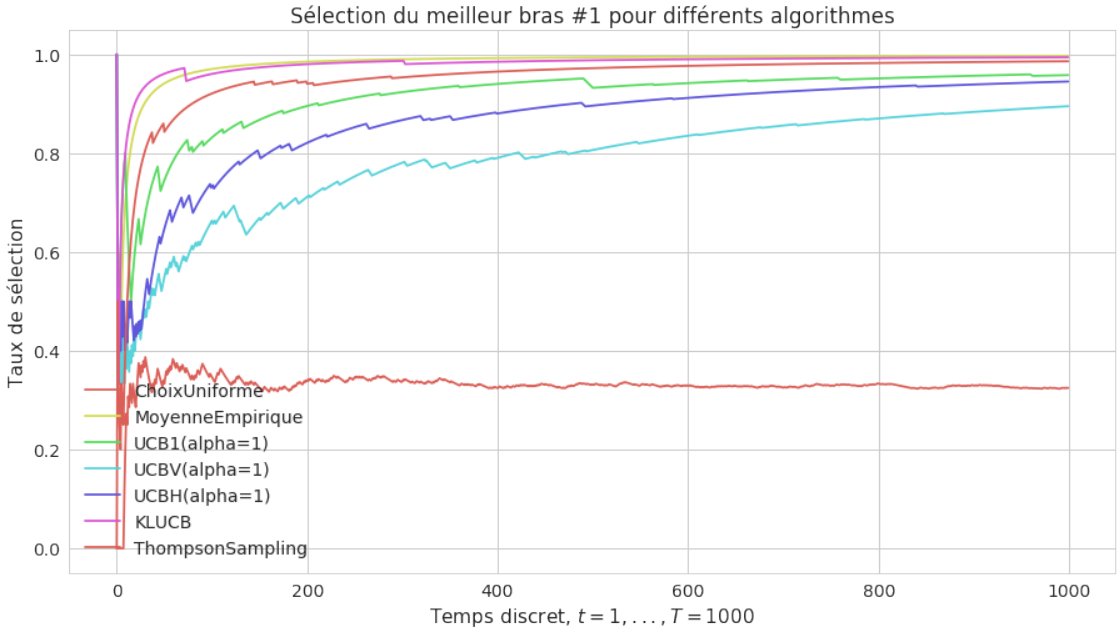
```
CPU times: user 335 ms, sys: 0 ns, total: 335 ms
Wall time: 335 ms
```

```
In [37]: recompenses, choix
```

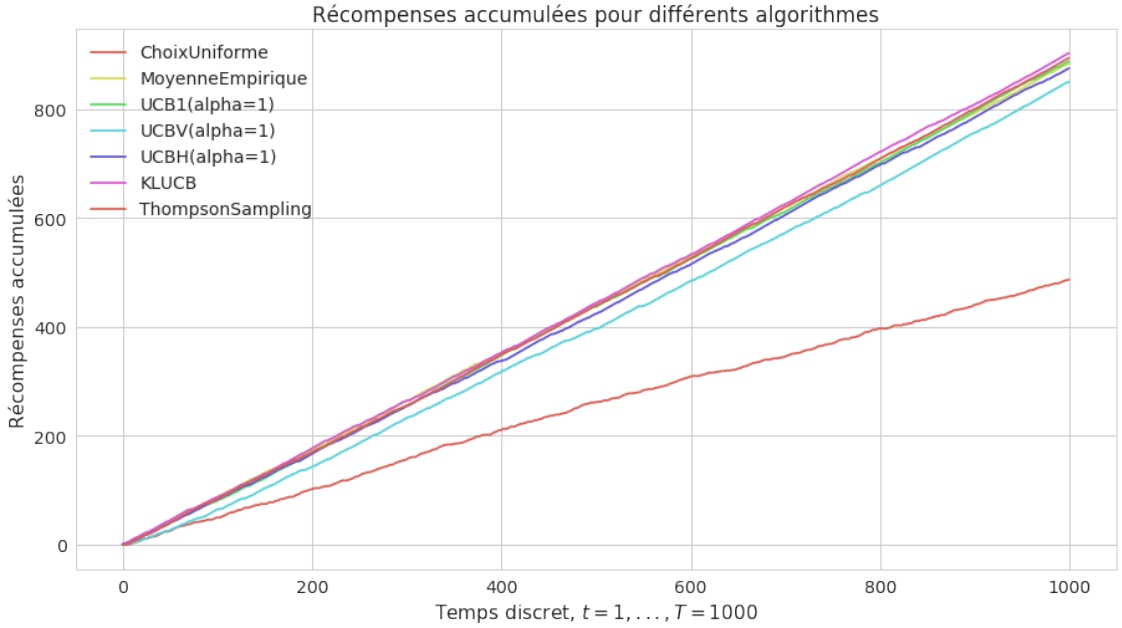
```
Out[37]: (array([[0., 0., 1., ..., 0., 0., 1.],
                 [1., 1., 0., ..., 1., 1., 1.],
                 [1., 0., 0., ..., 1., 1., 1.],
                 ...,
                 [1., 1., 0., ..., 1., 1., 1.],
                 [1., 0., 0., ..., 1., 1., 1.],
                 [0., 0., 0., ..., 1., 1., 1.]]), array([[2., 2., 1., ..., 1., 2., 0.],
                 [0., 1., 2., ..., 0., 0., 0.],
                 [0., 1., 2., ..., 0., 0., 0.],
                 ...,
                 [0., 1., 2., ..., 0., 0., 0.],
                 [0., 1., 2., ..., 0., 0., 0.],
                 [1., 0., 2., ..., 0., 0., 0.]]))
```

On affiche et vérifie les résultats attendus :

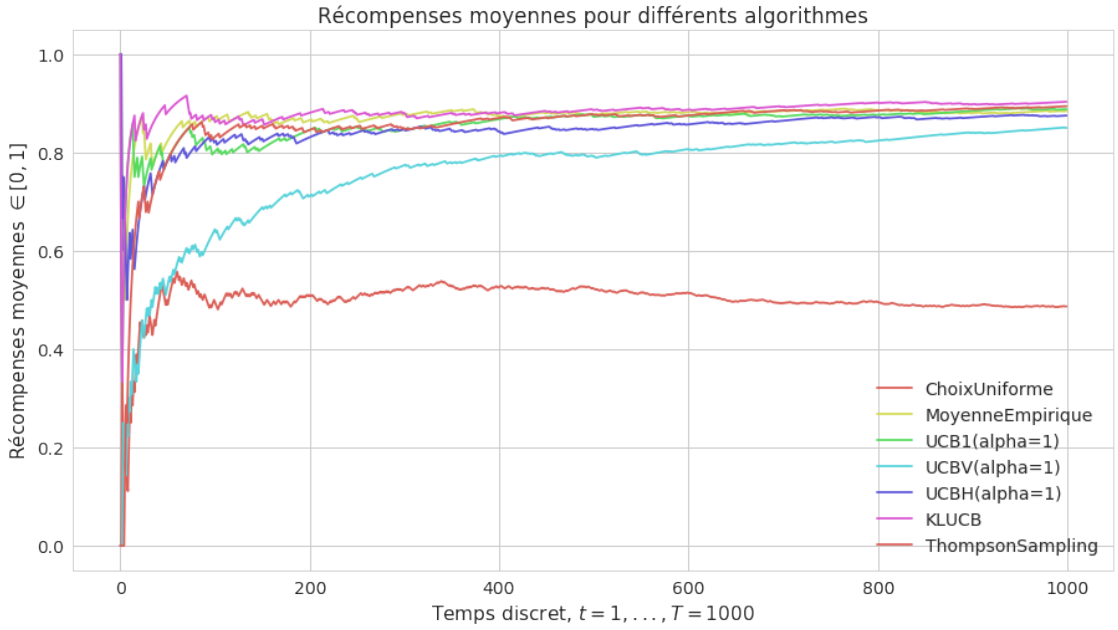
```
In [38]: affiche_selections(choix, noms, kstar)
```



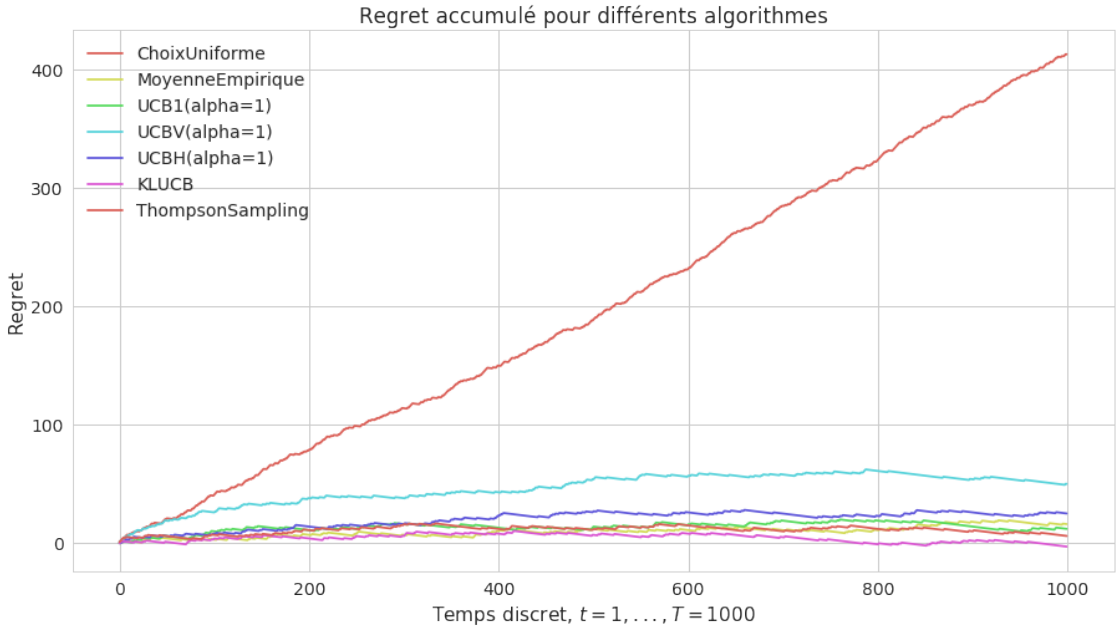
In [39]: `affiche_recompenses(recompenses, noms)`



In [40]: `affiche_recompenses_moyennes(recompenses, noms)`



In [41]: `affiche_regret(recompenses, noms, mustar=mus[kstar])`



$\Longrightarrow$ L'algorithme uniforme est, bien évidemment, très inefficace ! Il empêche de visualiser le regret des trois autres algorithmes.

Et sur ce problème simple à trois bras, avec *une seule simulation* (donc une variance immense), difficile de savoir lequel des trois algorithmes est le plus efficace...

2.10.3 Second problème, à 9 bras

```
In [42]: horizon = 5000
         mus = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
         bras = [ Bernoulli(mu) for mu in mus ]
         K = len(mus)
         kstar = np.argmax(mus)  # = 0

In [43]: horizon, mus, bras, K, kstar

Out[43]: (5000,
          [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1],
          [<__main__.Bernoulli at 0x7f1814a146a0>,
           <__main__.Bernoulli at 0x7f1814a141d0>,
           <__main__.Bernoulli at 0x7f1814a14a20>,
           <__main__.Bernoulli at 0x7f1814a14b38>,
           <__main__.Bernoulli at 0x7f1814a14390>,
           <__main__.Bernoulli at 0x7f1814a14358>,
           <__main__.Bernoulli at 0x7f1814a14dd8>,
           <__main__.Bernoulli at 0x7f1814a148d0>,
           <__main__.Bernoulli at 0x7f1814a14748>],
          9,
          0)
```

On va comparer différents choix de α pour l'algorithme UCB : $\alpha = 4, 1, 0.5, 0.1$.

```
In [44]: algorithmes = [MoyenneEmpirique(K),
                        UCB1(K, alpha=4), UCB1(K, alpha=1), UCB1(K, alpha=0.5), UCB1(K, alpha=
                        UCBV(K, alpha=1), UCBH(K, horizon, alpha=1),
                        KLUCB(K), ThompsonSampling(K)]
         algorithmes

Out[44]: [<__main__.MoyenneEmpirique at 0x7f1814a14630>,
          <__main__.UCB1 at 0x7f1814a14eb8>,
          <__main__.UCB1 at 0x7f1814a14400>,
          <__main__.UCB1 at 0x7f1814a146d8>,
          <__main__.UCB1 at 0x7f1814a14668>,
          <__main__.UCBV at 0x7f1814a14550>,
          <__main__.UCBH at 0x7f1814a14160>,
          <__main__.KLUCB at 0x7f1814a14198>,
          <__main__.ThompsonSampling at 0x7f1814a14208>]
```

Pour les légendes, on a besoin des noms des algorithmes :

```
In [45]: noms = ["MoyenneEmpirique",
                 "UCB1(alpha=4)", "UCB1(alpha=1)", "UCB1(alpha=0.5)", "UCB1(alpha=0.1)",
                 "UCBV(alpha=1)", "UCBH(alpha=1)",
                 "KLUCB", "ThompsonSampling"]
```

On peut commencer la simulation, pour chaque algorithme.

```
In [46]: %%time
         N = len(algorithmes)
         recompenses, choix = np.zeros((N, horizon)), np.zeros((N, horizon))

         for i, alg in tqdm(enumerate(algorithmes), desc="Algorithmes"):
             rec, ch = simulation(bras, alg, horizon)
             recompenses[i] = rec
             choix[i] = ch
```

```
HBox(children=(IntProgress(value=1, bar_style='info', description='Algorithmes', max=1), HTML(v
```

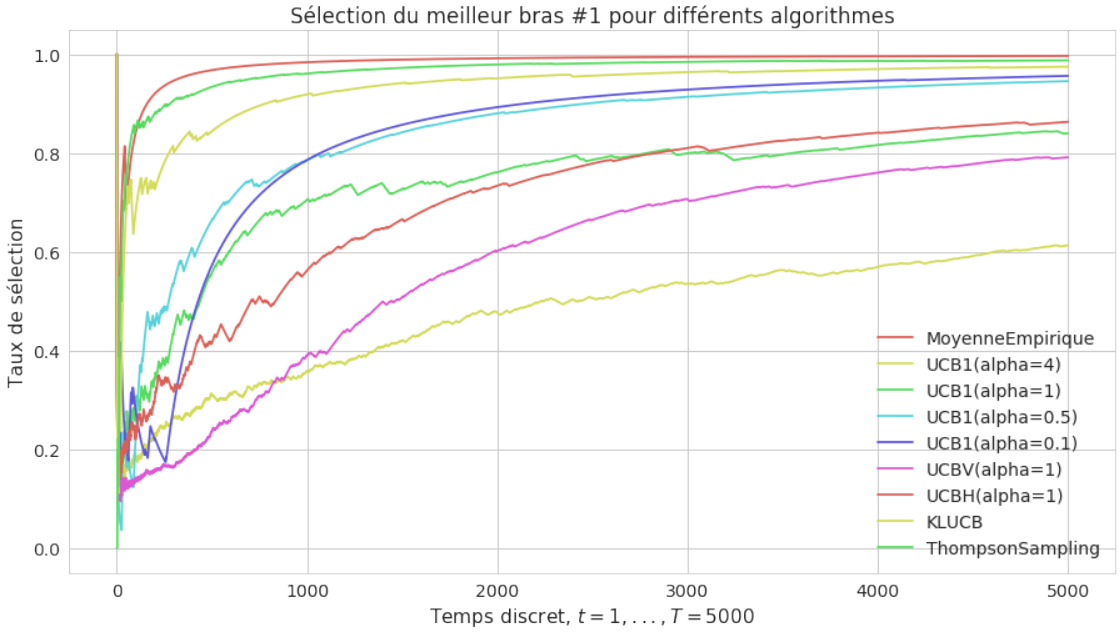
```
CPU times: user 2.75 s, sys: 45 ms, total: 2.79 s
Wall time: 2.76 s
```

```
In [47]: recompenses, choix
```

```
Out[47]: (array([[1., 1., 1., ..., 1., 1., 1.],
                 [1., 1., 1., ..., 1., 1., 1.],
                 [1., 1., 1., ..., 1., 0., 1.],
                 ...,
                 [1., 1., 0., ..., 1., 1., 1.],
                 [1., 1., 1., ..., 1., 1., 1.],
                 [0., 0., 1., ..., 1., 1., 1.]]), array([[0., 1., 2., ..., 0., 0., 0.],
                 [0., 1., 2., ..., 1., 1., 1.],
                 [0., 1., 2., ..., 0., 0., 0.],
                 ...,
                 [0., 1., 2., ..., 0., 0., 0.],
                 [0., 1., 2., ..., 0., 0., 0.],
                 [6., 8., 3., ..., 0., 0., 0.]]))
```

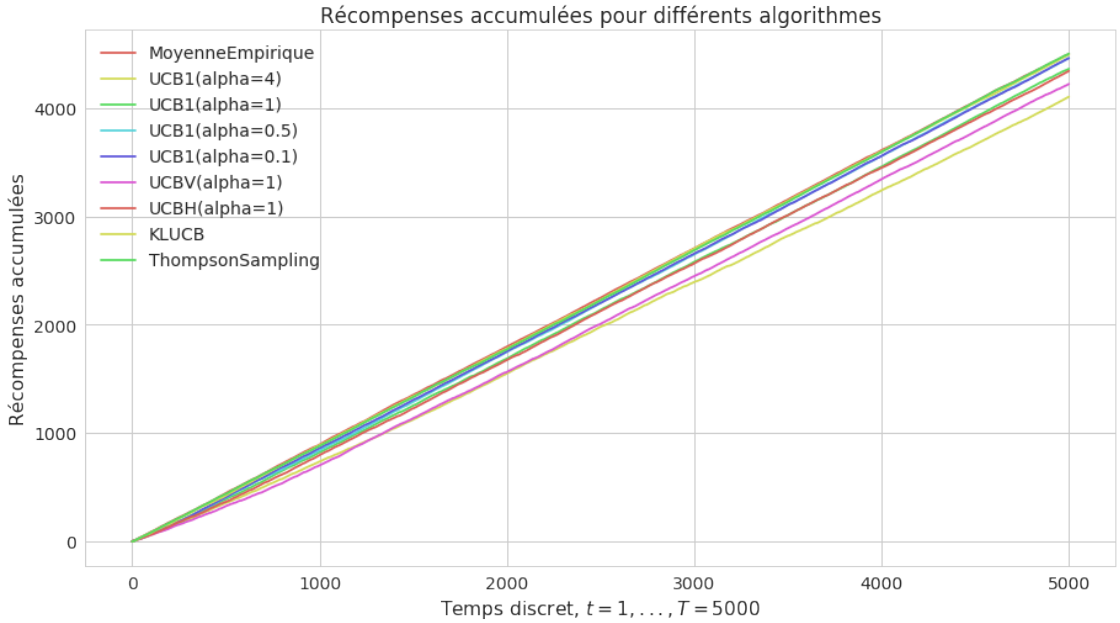
On affiche et vérifie les résultats attendus :

```
In [48]: affiche_selections(choix, noms, kstar)
```



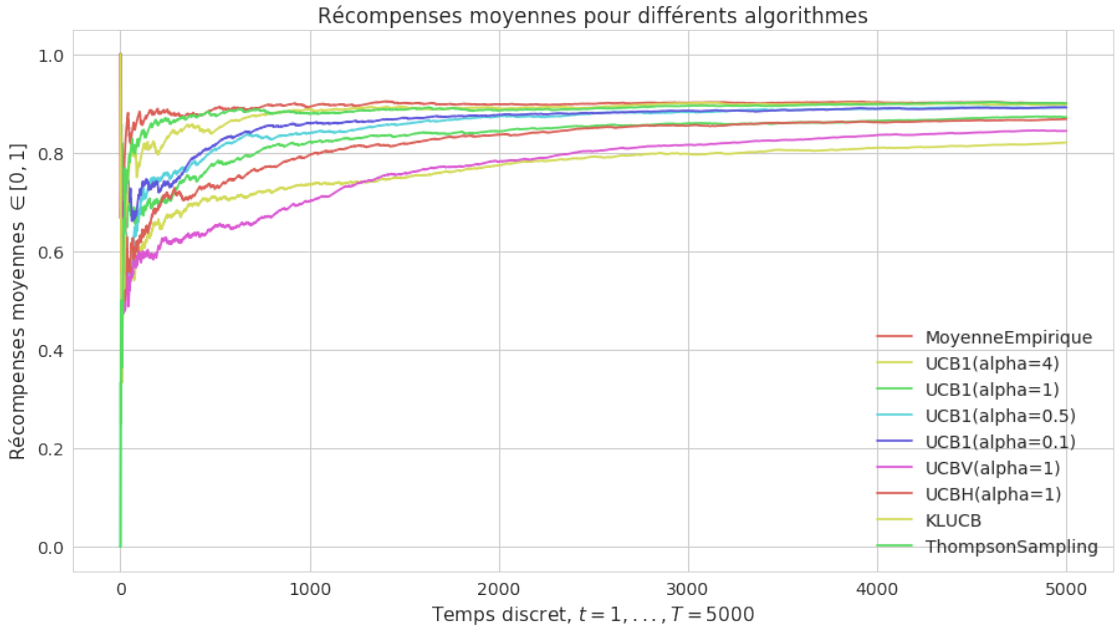
On commence à voir une différence de vitesse de convergence, pour l'identification du meilleur bras.

```
In [49]: affiche_recompenses(recompenses, noms)
```



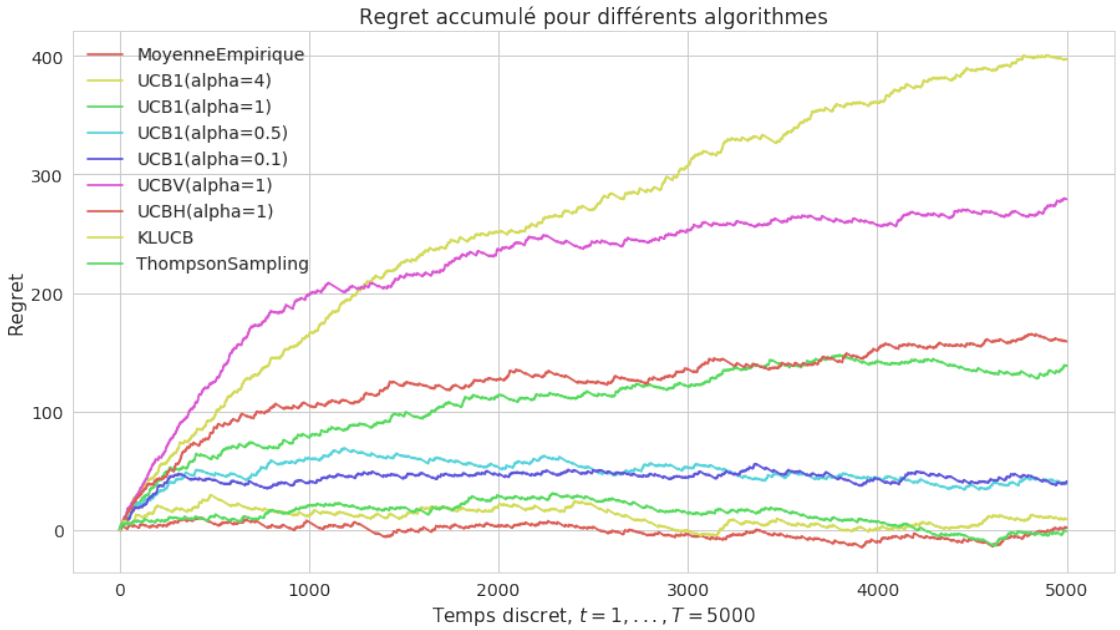
Difficile de différence quel algorithme est le plus efficace, même si clairement les plus grandes valeurs de $\alpha = 4, 1$ pour UCB semblent avoir moins bien fonctionné.

```
In [50]: affiche_recompenses_moyennes(recompenses, noms)
```



En terme de récompenses moyennes au cours du temps, les 4 algorithmes les plus efficaces convergent vers $\mu^* = \mu_1 = 0.9$.

```
In [51]: affiche_regret(recompenses, noms, mustar=mus[kstar])
```



Encore une fois, une seule simulation ne permet pas vraiment de conclure...
Mais on voit quand même :

- KLUCB, UCBH et Thompson Sampling sont tous les trois très efficaces,
- UCBV ne semble pas meilleur que UCB,
- Et FTL (MoyenneEmpirique) est pas si mauvais !

2.10.4 Troisième problème, simulé 100 fois

On s'intéresse au même problème, mais simulé 100 fois et sur un horizon plus long ($T = 10000$).

On s'intéressera ensuite aux statistiques *moyennes* (du regret cumulé) sur ces 100 répétitions pour les graphiques. On doit aussi considérer la distribution du regret (par exemple à $t = T$ fixé), afin de vérifier que *chaque* répétition de l'expérience a donné des résultats similaires (la moyenne efface les valeurs extrêmes si elles sont peu présentes).

```
In [52]: horizon = 10000
         repetitions = 100
         mus = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
         bras = [ Bernoulli(mu) for mu in mus ]
         K = len(mus)
         kstar = np.argmax(mus)  # = 0

In [53]: horizon, repetitions, mus, bras, K, kstar

Out[53]: (10000,
          100,
          [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1],
          [<__main__.Bernoulli at 0x7f1810d1e7f0>,
           <__main__.Bernoulli at 0x7f1810d1e240>,
           <__main__.Bernoulli at 0x7f1810d1ecf8>,
           <__main__.Bernoulli at 0x7f1810d1ec88>,
           <__main__.Bernoulli at 0x7f1810d1e8d0>,
           <__main__.Bernoulli at 0x7f1810d1e860>,
           <__main__.Bernoulli at 0x7f1810d1e470>,
           <__main__.Bernoulli at 0x7f1810d1e400>,
           <__main__.Bernoulli at 0x7f1810d1e2b0>],
          9,
          0)
```

On va comparer différents choix de α pour l'algorithme UCB : $\alpha = 4, 1, 0.5, 0.1$.

```
In [54]: algorithmes = [MoyenneEmpirique(K),
                        UCB1(K, alpha=4), UCB1(K, alpha=1), UCB1(K, alpha=0.5), UCB1(K, alpha=0
                        UCBV(K, alpha=1), UCBH(K, horizon, alpha=1),
                        KLUCB(K), ThompsonSampling(K)]
         algorithmes

Out[54]: [<__main__.MoyenneEmpirique at 0x7f1810d1eef0>,
          <__main__.UCB1 at 0x7f1810d1ecc0>,
```

```
 <__main__.UCB1 at 0x7f1810d1ec50>,
 <__main__.UCB1 at 0x7f1810d1e668>,
 <__main__.UCB1 at 0x7f1810d1ef98>,
 <__main__.UCBV at 0x7f1810d1e710>,
 <__main__.UCBH at 0x7f1810d1eb70>,
 <__main__.KLUCB at 0x7f1810d1e2e8>,
 <__main__.ThompsonSampling at 0x7f1810d1e208>]
```

Pour les légendes, on a besoin des noms des algorithmes :

```
In [55]: noms = ["MoyenneEmpirique",
                 "UCB1(alpha=4)", "UCB1(alpha=1)", "UCB1(alpha=0.5)", "UCB1(alpha=0.1)",
                 "UCBV(alpha=1)", "UCBH(alpha=1)",
                 "KLUCB", "ThompsonSampling"]
```

On peut commencer la simulation, pour chaque algorithme.

```
In [56]: %%time
         N = len(algorithmes)
         recompenses, choix = np.zeros((N, repetitions, horizon)), np.zeros((N, repetitions, ho

         # Pour chaque répétitions
         for rep in tqdm(range(repetitions), desc="Répétitions"):
             for i, alg in enumerate(algorithmes):
                 rec, ch = simulation(bras, alg, horizon)
                 recompenses[i, rep] = rec
                 choix[i, rep] = ch

HBox(children=(IntProgress(value=0, description='Répétitions'), HTML(value='')))



CPU times: user 10 mins, sys: 0 ns, total: 10 mins
Wall time: 10 mins
```

```
In [71]: # On moyenne
         recompenses_moy = np.mean(recompenses, axis=1)
         choix_moy = np.mean(choix, axis=1)
         print("Dimension de 'recompenses_moy' =", np.shape(recompenses_moy))  # juste pour vé

         # On garde la distribution à t=T
         recompenses_fin = np.sum(recompenses, axis=2)
         choix_fin = np.sum(choix, axis=2)
         print("Dimension de 'recompenses_fin' =", np.shape(recompenses_fin))  # juste pour vé

Dimension de 'recompenses_moy' = (9, 10000)
Dimension de 'recompenses_fin' = (9, 100)
```

Cette fois, les statistiques accumulées ne sont plus entières.

```
In [72]: recompenses_moy, choix_moy
```

```
Out[72]: (array([[0.94, 0.77, 0.68, ..., 0.86, 0.88, 0.88],
                 [0.94, 0.82, 0.64, ..., 0.91, 0.9 , 0.92],
                 [0.91, 0.82, 0.66, ..., 0.91, 0.94, 0.87],
                 ...,
                 [0.94, 0.82, 0.69, ..., 0.87, 0.93, 0.79],
                 [0.89, 0.76, 0.75, ..., 0.83, 0.92, 0.86],
                 [0.47, 0.53, 0.45, ..., 0.89, 0.92, 0.9 ]]),
          array([[0.  , 1.  , 2.  , ..., 0.34, 0.34, 0.34],
                 [0.  , 1.  , 2.  , ..., 0.15, 0.16, 0.17],
                 [0.  , 1.  , 2.  , ..., 0.02, 0.02, 0.02],
                 ...,
                 [0.  , 1.  , 2.  , ..., 0.07, 0.06, 0.06],
                 [0.  , 1.  , 2.  , ..., 0.  , 0.  , 0.  ],
                 [3.67, 3.57, 3.8 , ..., 0.  , 0.  , 0.  ]]))
```

```
In [73]: recompenses_fin
```

```
Out[73]: array([[7875., 6986., 8015., 8993., 8968., 9022., 8980., 9066., 9004.,
                 8975., 8998., 8037., 8973., 7034., 9032., 8935., 9021., 8956.,
                 6991., 7948., 7980., 8935., 8995., 7970., 8992., 8997., 9041.,
                 8951., 9030., 8983., 8969., 9024., 8996., 9040., 7957., 8982.,
                 8971., 7017., 9038., 8980., 9029., 9028., 8993., 8017., 8040.,
                 8964., 8991., 8039., 7994., 8986., 8998., 9039., 9021., 8018.,
                 8984., 8988., 8992., 9020., 7990., 8977., 9011., 9038., 8937.,
                 9015., 9004., 8974., 9000., 9040., 9017., 9007., 8982., 8996.,
                 8981., 6013., 7012., 8996., 8990., 7983., 8923., 8001., 9016.,
                 8986., 8917., 7981., 7021., 9013., 8962., 8996., 8976., 8018.,
                 8979., 9035., 9001., 9001., 9016., 8008., 8959., 7960., 9015.,
                 8985.],
                [8464., 8354., 8367., 8447., 8346., 8472., 8440., 8385., 8424.,
                 8416., 8414., 8423., 8487., 8451., 8440., 8444., 8432., 8499.,
                 8417., 8409., 8389., 8400., 8380., 8446., 8441., 8447., 8371.,
                 8465., 8397., 8387., 8418., 8439., 8490., 8357., 8422., 8496.,
                 8426., 8424., 8471., 8462., 8475., 8466., 8460., 8385., 8458.,
                 8428., 8435., 8463., 8439., 8480., 8436., 8413., 8385., 8359.,
                 8460., 8451., 8406., 8373., 8453., 8460., 8428., 8460., 8349.,
                 8457., 8412., 8458., 8370., 8459., 8521., 8380., 8397., 8439.,
                 8383., 8484., 8423., 8440., 8418., 8480., 8421., 8353., 8427.,
                 8490., 8469., 8407., 8474., 8432., 8424., 8403., 8425., 8468.,
                 8378., 8467., 8350., 8398., 8485., 8458., 8389., 8481., 8434.,
                 8409.],
                [8794., 8798., 8841., 8827., 8876., 8833., 8798., 8807., 8814.,
                 8789., 8766., 8796., 8853., 8830., 8798., 8830., 8835., 8809.,
                 8807., 8811., 8840., 8843., 8795., 8802., 8795., 8841., 8804.,
                 8773., 8736., 8850., 8835., 8857., 8818., 8840., 8848., 8833.,
```

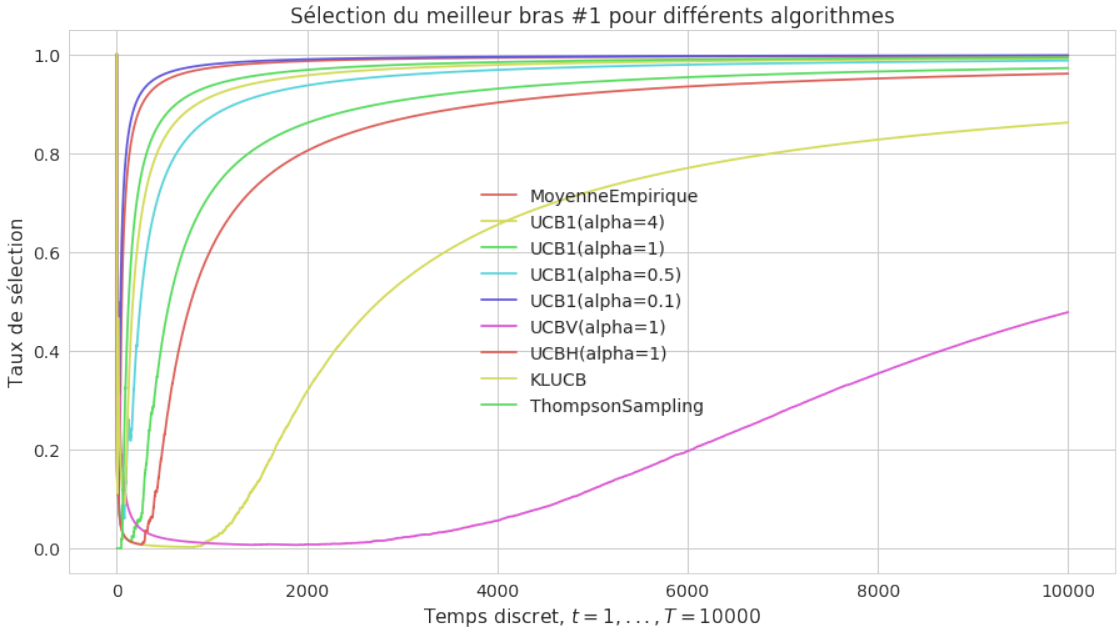
```
 8798., 8763., 8816., 8790., 8812., 8830., 8842., 8794., 8773.,
 8755., 8856., 8884., 8848., 8794., 8837., 8748., 8852., 8816.,
 8796., 8790., 8806., 8797., 8804., 8858., 8882., 8826., 8765.,
 8800., 8788., 8769., 8865., 8787., 8831., 8786., 8804., 8804.,
 8842., 8817., 8801., 8793., 8797., 8820., 8819., 8814., 8855.,
 8840., 8784., 8821., 8826., 8758., 8765., 8862., 8853., 8791.,
 8816., 8835., 8843., 8779., 8799., 8824., 8791., 8786., 8821.,
 8809.],
[8934., 8924., 8938., 8903., 8909., 8862., 8961., 8926., 8893.,
 8907., 8896., 8878., 8887., 8912., 8880., 8956., 8889., 8887.,
 8944., 8914., 8918., 8911., 8905., 8902., 8916., 8935., 8887.,
 8922., 8928., 8942., 8899., 8858., 8912., 8844., 8921., 8942.,
 8912., 8902., 8906., 8874., 8928., 8901., 8901., 8868., 8948.,
 8871., 8921., 8897., 8848., 8838., 8891., 8833., 8903., 8892.,
 8904., 8949., 8926., 8878., 8862., 8906., 8910., 8872., 8934.,
 8879., 8874., 8906., 8852., 8913., 8891., 8915., 8909., 8888.,
 8924., 8886., 8859., 8912., 8914., 8883., 8915., 8836., 8858.,
 8895., 8921., 8869., 8886., 8914., 8953., 8893., 8863., 8905.,
 8940., 8906., 8927., 8865., 8878., 8915., 8860., 8938., 8883.,
 8949.],
[8991., 9023., 8935., 8985., 8927., 8962., 8984., 9023., 8849.,
 8973., 8978., 8966., 8978., 8946., 8951., 9033., 8944., 8937.,
 8905., 9010., 8988., 8971., 8978., 8975., 8981., 8945., 8984.,
 8975., 9002., 8893., 9005., 8941., 8941., 8872., 9005., 8954.,
 8932., 8937., 8972., 8991., 8952., 8970., 9004., 8879., 8960.,
 8910., 8972., 8939., 8981., 8885., 8967., 8998., 9007., 9002.,
 9029., 8977., 8957., 8966., 8966., 8945., 8970., 8989., 8951.,
 9003., 8983., 8980., 8977., 8960., 8930., 8910., 8939., 8978.,
 8887., 8932., 8946., 9001., 8980., 8993., 8974., 9011., 8930.,
 9012., 8957., 8967., 8967., 9012., 8995., 8871., 9019., 8978.,
 8977., 8970., 8928., 9007., 8980., 8972., 8932., 8914., 8984.,
 8975.],
[8669., 8562., 8551., 8505., 8466., 8476., 8389., 8375., 8319.,
 8319., 8318., 8309., 8301., 8176., 8215., 8178., 8188., 8196.,
 8189., 8130., 8162., 8143., 8128., 8064., 8031., 8084., 7979.,
 8085., 8029., 8048., 8043., 7970., 7964., 7984., 8004., 8010.,
 7935., 7914., 7974., 7895., 7923., 7898., 7953., 7927., 7846.,
 7856., 7899., 7890., 7854., 7832., 7817., 7804., 7819., 7757.,
 7888., 7830., 7748., 7823., 7825., 7786., 7753., 7817., 7783.,
 7799., 7730., 7703., 7725., 7796., 7709., 7776., 7680., 7724.,
 7786., 7649., 7700., 7735., 7729., 7689., 7729., 7808., 7673.,
 7700., 7676., 7681., 7734., 7643., 7675., 7692., 7665., 7589.,
 7631., 7667., 7690., 7701., 7660., 7615., 7670., 7681., 7610.,
 7644.],
[8867., 8794., 8813., 8824., 8809., 8802., 8863., 8858., 8772.,
 8793., 8739., 8807., 8818., 8802., 8820., 8873., 8851., 8845.,
 8771., 8787., 8824., 8746., 8850., 8840., 8830., 8830., 8783.,
 8781., 8811., 8871., 8750., 8795., 8840., 8785., 8802., 8863.,
```

```
       8804., 8827., 8808., 8811., 8819., 8889., 8797., 8826., 8822.,
       8772., 8798., 8807., 8827., 8851., 8832., 8821., 8831., 8869.,
       8826., 8828., 8734., 8855., 8799., 8781., 8789., 8861., 8771.,
       8797., 8861., 8829., 8819., 8850., 8847., 8809., 8848., 8781.,
       8783., 8800., 8814., 8772., 8790., 8829., 8789., 8803., 8732.,
       8817., 8827., 8834., 8823., 8808., 8829., 8770., 8849., 8809.,
       8857., 8846., 8813., 8851., 8799., 8824., 8833., 8838., 8791.,
       8804.],
      [8959., 8903., 8914., 8861., 8886., 9010., 8952., 8956., 8982.,
       8905., 8914., 8953., 8918., 8909., 8956., 8945., 8925., 8976.,
       8917., 8960., 8954., 8918., 8924., 8898., 8923., 8910., 8991.,
       8931., 8972., 8911., 8950., 8934., 8932., 8961., 8938., 8954.,
       8966., 8961., 8988., 8923., 8967., 8951., 8925., 8985., 8939.,
       9003., 8936., 8973., 8974., 8925., 8935., 8998., 8921., 8894.,
       8924., 8943., 8952., 8946., 8956., 8962., 8952., 8956., 8925.,
       8934., 8957., 8989., 8944., 8931., 8937., 8977., 8925., 8901.,
       8920., 8990., 8911., 8892., 8962., 8987., 8926., 8992., 8941.,
       8951., 8912., 9007., 8947., 8976., 8954., 8948., 8942., 8924.,
       8932., 8943., 8943., 8922., 8900., 8950., 8979., 8891., 8946.,
       8928.],
      [8951., 8956., 8997., 8922., 9004., 8978., 8947., 8952., 8965.,
       9011., 8979., 9025., 8966., 8958., 9003., 8990., 8987., 8960.,
       8956., 8965., 8932., 8967., 9016., 8984., 8936., 8998., 8968.,
       8983., 8992., 8900., 8988., 8938., 8920., 8987., 8979., 8889.,
       8986., 8946., 8921., 8987., 8914., 8873., 8956., 8956., 8972.,
       8975., 8933., 8955., 8938., 8925., 8932., 8966., 9015., 8929.,
       8963., 8948., 8977., 8983., 8968., 8965., 9005., 8980., 8933.,
       8988., 8983., 8963., 8935., 9004., 8944., 8984., 8943., 8957.,
       8928., 8990., 8974., 9006., 8920., 8941., 8919., 8980., 8938.,
       8941., 8907., 8980., 8988., 8956., 8994., 8963., 8928., 8930.,
       8920., 8992., 8917., 8989., 8976., 8895., 8950., 8958., 8941.,
       8963.]])
```

On affiche et vérifie les résultats attendus :

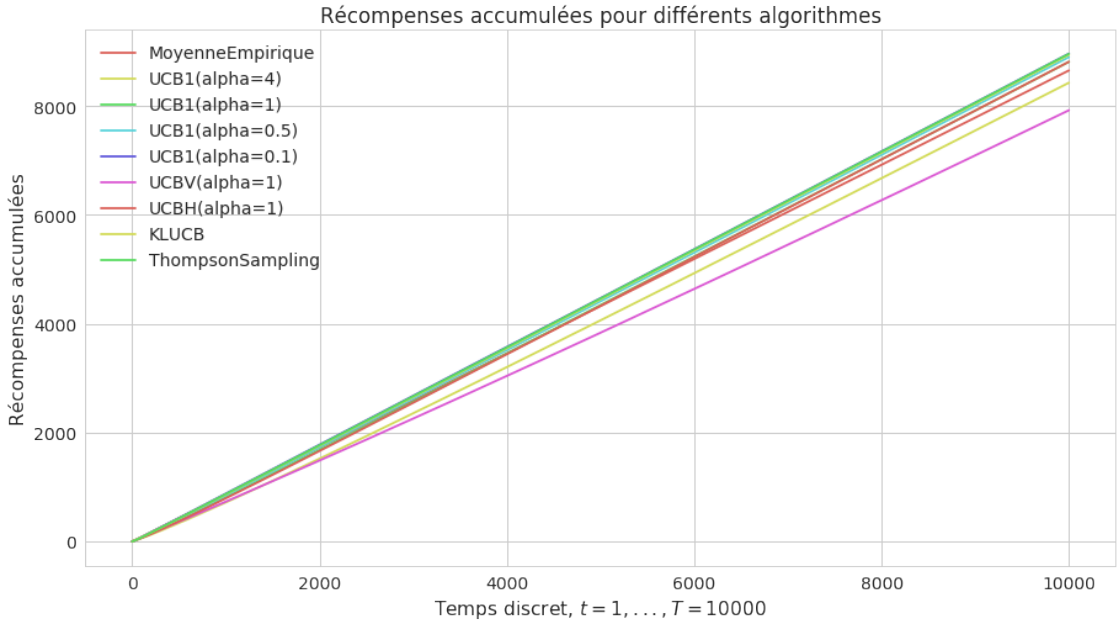
```
In [74]: choix_moy = np.floor(choix_moy)
```

```
In [75]: affiche_selections(choix_moy, noms, kstar)
```



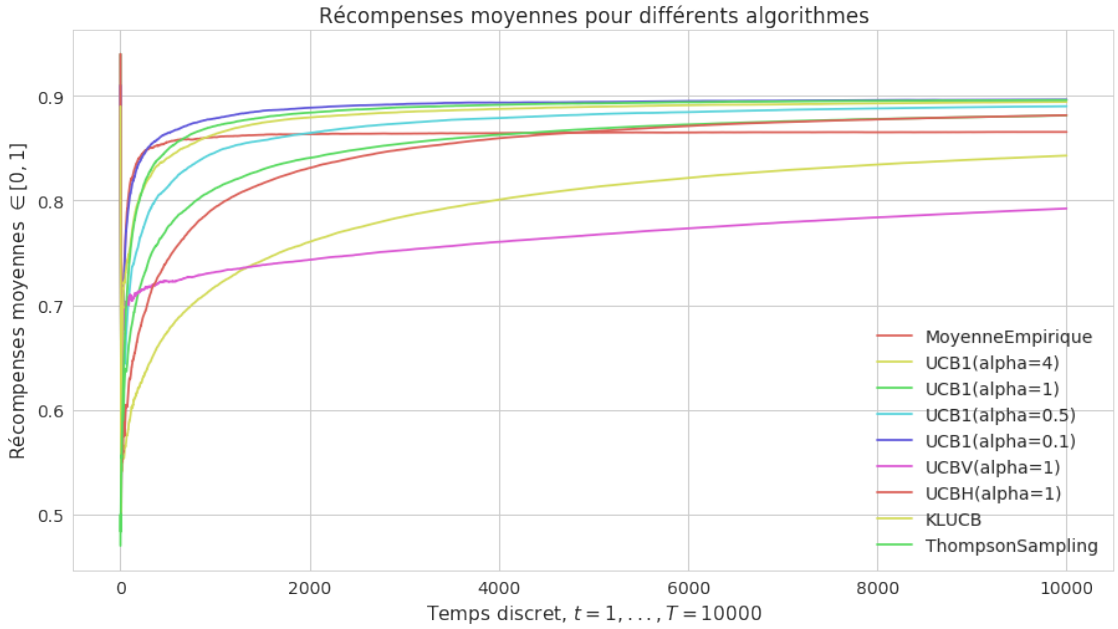
On commence à voir une différence de vitesse de convergence, pour l'identification du meilleur bras.

```
In [76]: affiche_recompenses(recompenses_moy, noms)
```



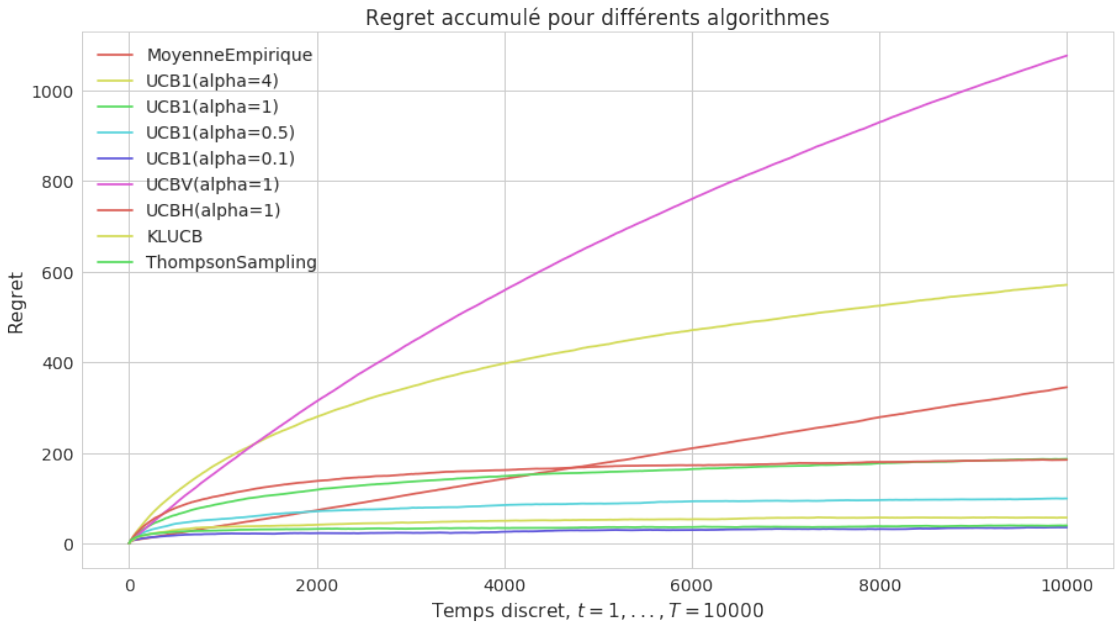
Difficile de différence quel algorithme est le plus efficace, même si clairement les plus grandes valeurs de $\alpha = 4, 1$ pour UCB semblent avoir moins bien fonctionner.

```
In [77]: affiche_recompenses_moyennes(recompenses_moy, noms)
```

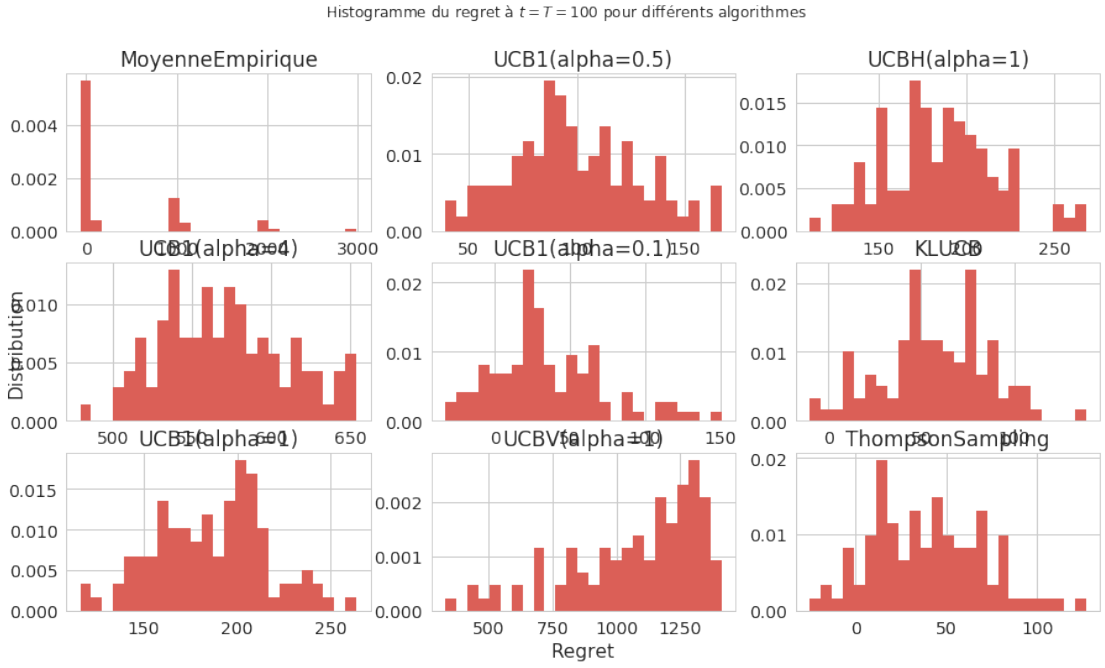


En terme de récompenses moyennes au cours du temps, les 4 algorithmes les plus efficaces convergent vers $\mu^* = \mu_1 = 0.9$.

```
In [89]: affiche_regret(recompenses_moy, noms, mustar=mus[kstar])
```



```
In [95]: affiche_hist_regret(recompenses_fin, noms, horizon, mustar=mus[kstar])
```



Attention, l'axe x est différent pour chaque histogramme !

Cette fois, ces 100 simulations permet de conclure quelques points :

- `MoyenneEmpirique` converge rapidement au début, mais son regret $\mathcal{R}(t)$ ne semble pas avoir un profil logarithmique (il faudrait essayer de plus grands horizons pour le confirmer),
- Les différents `UCB` semblent tous avoir un regret logarithmique, i.e., $\mathcal{R}(t) = \mathcal{O}(\log t)$, et plus le α est petit, plus le regret est bas,
- `ThompsonSampling` et `KLUCB` fonctionnent très bien, sans avoir à choisir de constante α.

2.10.5 Dernier problème

On s'intéresse enfin à un problème bien plus difficile.

```
In [96]: horizon = 10000
         repetitions = 10
         mus = [0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.02, 0.02, 0.02, 0.01, 0.01, 0.005, 0
         bras = [ Bernoulli(mu) for mu in mus ]
         K = len(mus)
         kstar = np.argmax(mus)
```

```
In [97]: horizon, repetitions, mus, bras, K, kstar
```

```
Out[97]: (10000,
          10,
          [0.3,
           0.05,
           0.05,
           0.05,
           0.05,
           0.05,
           0.05,
           0.02,
           0.02,
           0.02,
           0.01,
           0.01,
           0.005,
           0.005,
           0.001,
           0.001],
          [<__main__.Bernoulli at 0x7f180b7dce10>,
           <__main__.Bernoulli at 0x7f180b7e41d0>,
           <__main__.Bernoulli at 0x7f180b7e4c50>,
           <__main__.Bernoulli at 0x7f180b7e4be0>,
           <__main__.Bernoulli at 0x7f180b7ef390>,
           <__main__.Bernoulli at 0x7f180b7ef710>,
           <__main__.Bernoulli at 0x7f180b7efa90>,
           <__main__.Bernoulli at 0x7f180b7efe10>,
           <__main__.Bernoulli at 0x7f180b7fa1d0>,
           <__main__.Bernoulli at 0x7f180b7fa550>,
           <__main__.Bernoulli at 0x7f180b7fa8d0>,
           <__main__.Bernoulli at 0x7f180b7fabe0>,
           <__main__.Bernoulli at 0x7f180b7fac88>,
           <__main__.Bernoulli at 0x7f180b782710>,
           <__main__.Bernoulli at 0x7f180b782a90>,
           <__main__.Bernoulli at 0x7f180b782e10>],
          16,
          0)
```

On va comparer différents choix de α pour l'algorithme UCB : $\alpha = 4, 1, 0.5, 0.1, 0.01, 0.001$.

```
In [98]: algorithmes = [MoyenneEmpirique(K),
                        UCB1(K, alpha=4), UCB1(K, alpha=1), UCB1(K, alpha=0.5),
                        UCB1(K, alpha=0.1), UCB1(K, alpha=0.01), UCB1(K, alpha=0.001),
                        UCBV(K, alpha=1), UCBH(K, horizon, alpha=1),
                        KLUCB(K), ThompsonSampling(K)]
         algorithmes

Out[98]: [<__main__.MoyenneEmpirique at 0x7f180b91d470>,
          <__main__.UCB1 at 0x7f180b91d0f0>,
```

```
          <__main__.UCB1 at 0x7f180b91db70>,
          <__main__.UCB1 at 0x7f180b91def0>,
          <__main__.UCB1 at 0x7f180b91d7f0>,
          <__main__.UCB1 at 0x7f180b9317f0>,
          <__main__.UCB1 at 0x7f180b931ef0>,
          <__main__.UCBV at 0x7f180b931470>,
          <__main__.UCBH at 0x7f180b9310f0>,
          <__main__.KLUCB at 0x7f180b931eb8>,
          <__main__.ThompsonSampling at 0x7f180b8ee160>]
```

Pour les légendes, on a besoin des noms des algorithmes :

```
In [99]: noms = ["MoyenneEmpirique",
                 "UCB1(alpha=4)", "UCB1(alpha=1)", "UCB1(alpha=0.5)",
                 "UCB1(alpha=0.1)", "UCB1(alpha=0.01)", "UCB1(alpha=0.001)",
                 "UCBV(alpha=1)", "UCBH(alpha=1)",
                 "KLUCB", "ThompsonSampling"]
```

On peut commencer la simulation, pour chaque algorithme.

```
In [100]: %%time
          N = len(algorithmes)
          recompenses, choix = np.zeros((N, repetitions, horizon)), np.zeros((N, repetitions, h

          # Pour chaque répétitions
          for rep in tqdm(range(repetitions), desc="Répétitions"):
              for i, alg in enumerate(algorithmes):
                  rec, ch = simulation(bras, alg, horizon)
                  recompenses[i, rep] = rec
                  choix[i, rep] = ch

HBox(children=(IntProgress(value=0, description='Répétitions', max=10), HTML(value='')))



CPU times: user 1min 14s, sys: 625 ms, total: 1min 15s
Wall time: 1min 14s
```

```
In [101]: # On moyenne
          recompenses_moy = np.mean(recompenses, axis=1)
          choix_moy = np.mean(choix, axis=1)
          print("Dimension de 'recompenses_moy' =", np.shape(recompenses_moy))  # juste pour v

          # On garde la distribution à t=T
          recompenses_fin = np.sum(recompenses, axis=2)
          choix_fin = np.sum(choix, axis=2)
          print("Dimension de 'recompenses_fin' =", np.shape(recompenses_fin))  # juste pour v
```

```
Dimension de 'recompenses_moy' = (11, 10000)
Dimension de 'recompenses_fin' = (11, 10)
```

Cette fois, les statistiques accumulées ne sont plus entières.

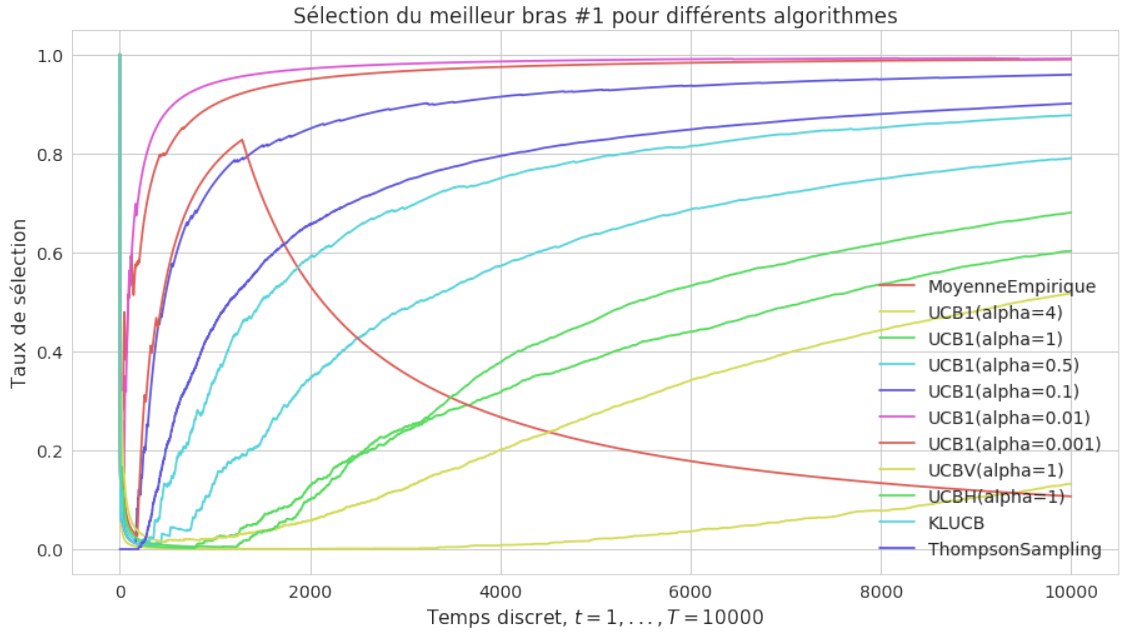
```
In [102]: recompenses_moy, choix_moy
```

```
Out[102]: (array([[0.3, 0.1, 0. , ..., 0.1, 0.4, 0.3],
                  [0.5, 0. , 0. , ..., 0.1, 0.1, 0.2],
                  [0.2, 0. , 0.1, ..., 0.1, 0.3, 0.4],
                  ...,
                  [0.1, 0.1, 0.1, ..., 0.2, 0.4, 0.2],
                  [0.1, 0. , 0.1, ..., 0.1, 0.2, 0.2],
                  [0.1, 0.4, 0. , ..., 0.2, 0.3, 0.3]]),
           array([[0. , 1. , 2. , ..., 0.9, 0.9, 0.9],
                  [0. , 1. , 2. , ..., 3.2, 1.7, 1.2],
                  [0. , 1. , 2. , ..., 0. , 0. , 0. ],
                  ...,
                  [0. , 1. , 2. , ..., 0. , 0. , 0. ],
                  [0. , 1. , 2. , ..., 0. , 0. , 0. ],
                  [4.9, 7.2, 7.3, ..., 0. , 0. , 0. ]]))
```

On affiche et vérifie les résultats attendus :

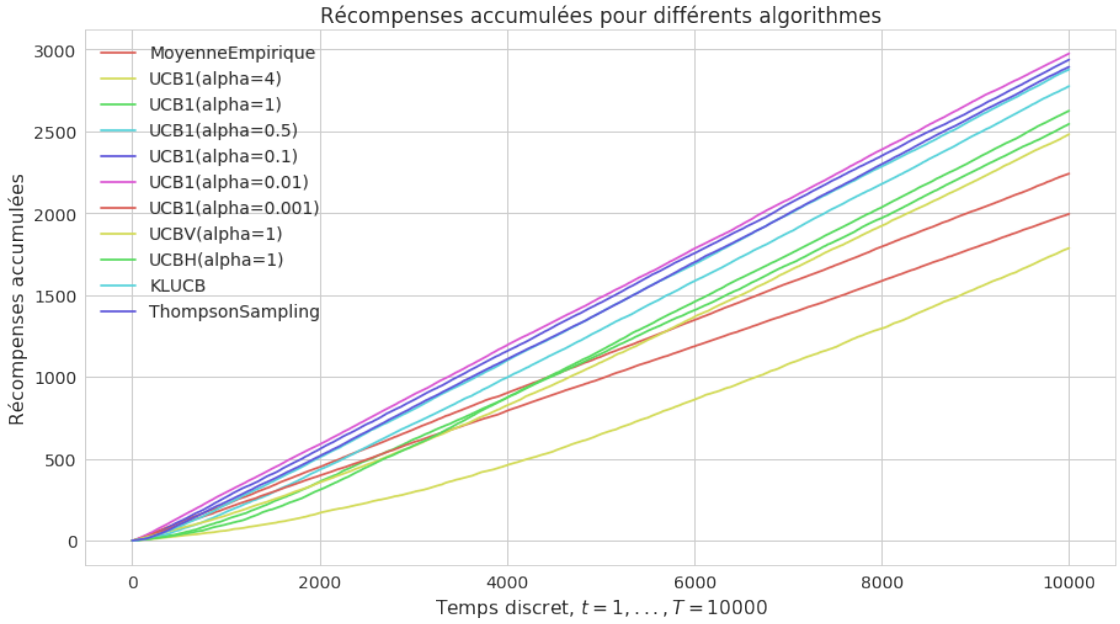
```
In [103]: choix_moy = np.floor(choix_moy)
```

```
In [104]: affiche_selections(choix_moy, noms, kstar)
```



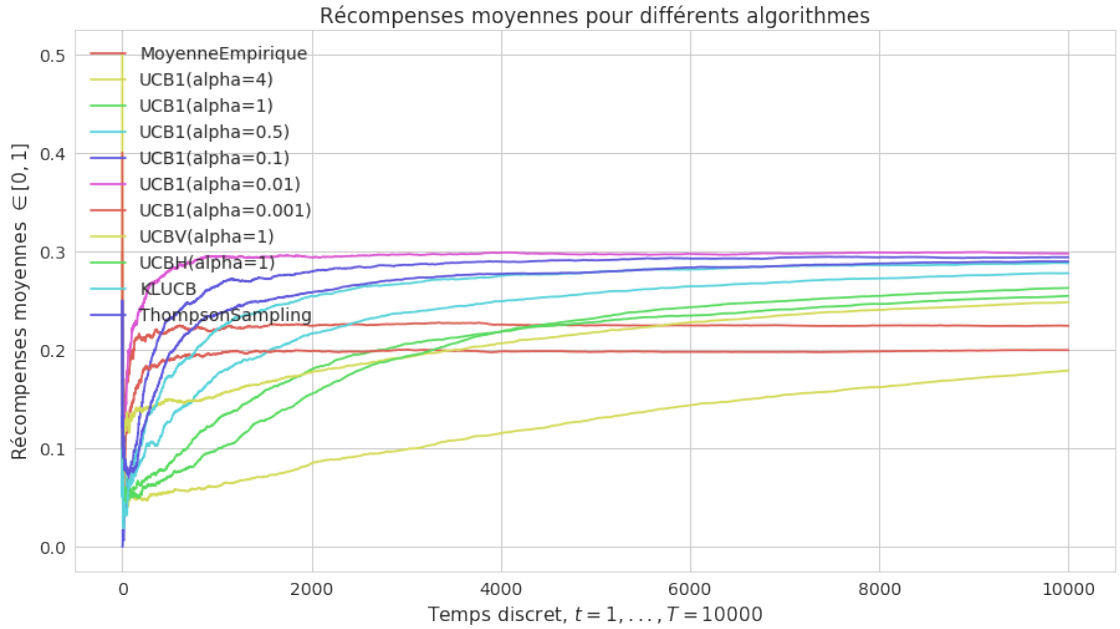
On commence à voir une différence de vitesse de convergence, pour l'identification du meilleur bras.

```
In [105]: affiche_recompenses(recompenses_moy, noms)
```



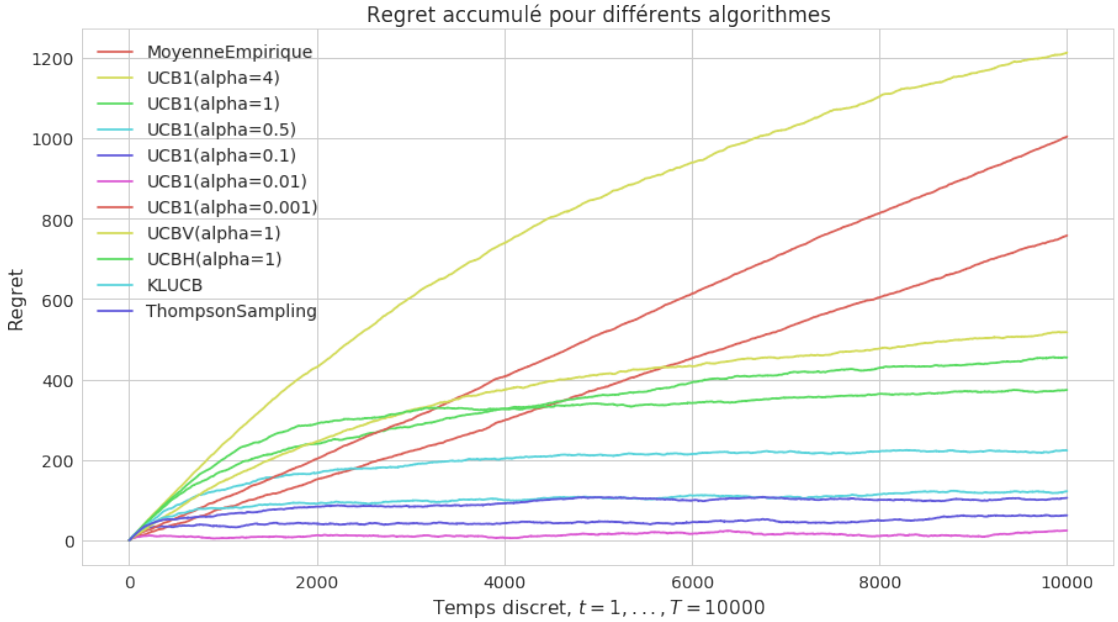
Difficile de différence quel algorithme est le plus efficace, même si clairement les plus grandes valeurs de $\alpha = 4, 1$ pour UCB semblent avoir moins bien fonctionner.

```
In [106]: affiche_recompenses_moyennes(recompenses_moy, noms)
```

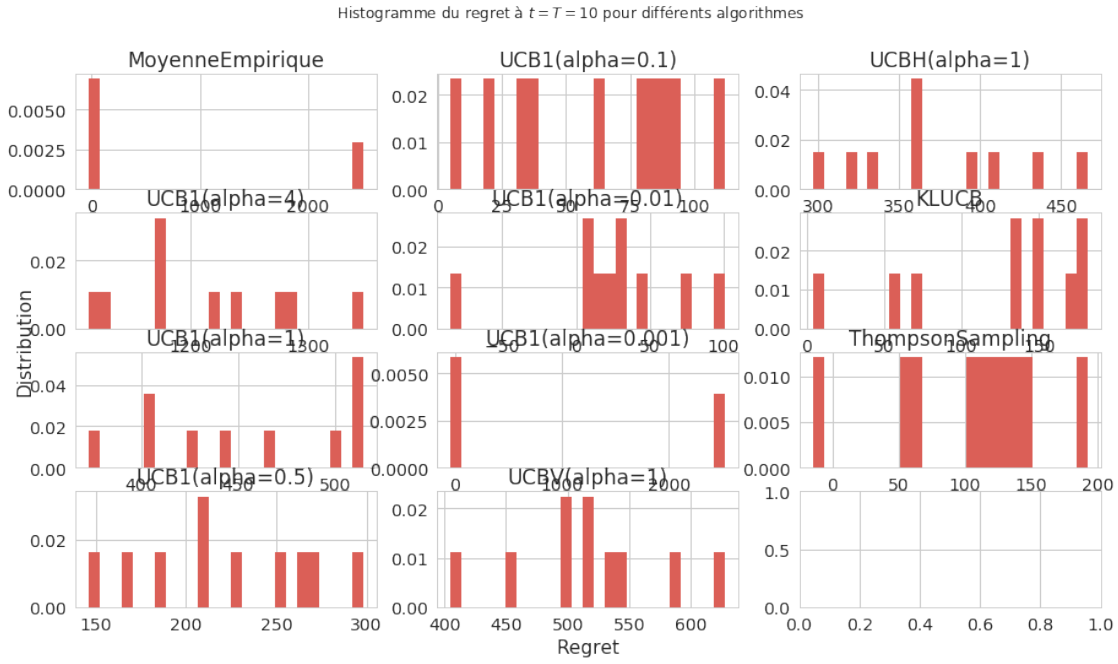


En terme de récompenses moyennes au cours du temps, les 4 algorithmes les plus efficaces convergent vers $\mu^* = \mu_1 = 0.9$.

```
In [107]: affiche_regret(recompenses_moy, noms, mustar=mus[kstar])
```




```
In [108]: affiche_hist_regret(recompenses_fin, noms, horizon, mustar=mus[kstar])
```



Cette fois, cette autre simulation permet de renforcer les conclusions précédentes :

- `MoyenneEmpirique` converge rapidement au début, mais son regret $\mathcal{R}(t)$ ne semble pas avoir un profil logarithmique (il faudrait essayer de plus grands horizons pour le confirmer). De plus, sur l'histogramme on voit que pour certaines expériences ses performances sont très mauvaises.
- Les différents `UCB1` semblent tous avoir un regret logarithmique, i.e., $\mathcal{R}(t) = \mathcal{O}(\log t)$, et plus le α est petit, plus le regret est bas,
- Mais si α est trop petit (e.g., $\alpha = 10^{-3}$), `UCB1(alpha)` n'est pas assez exploiteur, et pour ce problème à $K = 15$ bras, il ne fonctionne plus bien,
- `ThompsonSampling` et `KLUCB` fonctionnent généralement aussi bien qu'un `UCB1` avec "la bonne constante".

2.11 Une autre politique d'indice : ApproximatedFHGittins

Je propose de terminer par l'implémentation d'un dernier algorithme, ressemblant beaucoup à `UCB1`, mais qui utilise des indices un peu plus compliqués à calculer.

Il s'agit de l'algorithme proposé en 2016 par Tor Lattimore, dans cet article (et sa présentation à COLT'16), nommé `ApproximatedFHGittins`, pour *Approximated Finite-Horizon Gittins index*, ou Approximation *des Indices de Gittins à Horizon Fini* en français.

Cet algorithme *exige de connaître l'horizon* T, et calcule l'indice suivant, au temps t et après $N_k(t)$ sélections du bras k :

$$I_k(t) = \frac{X_k(t)}{N_k(t)} + \sqrt{\frac{\alpha}{2N_k(t)} \log\left(\frac{m}{N_k(t) \log^{1/2}\left(\frac{m}{N_k(t)}\right)} \right)}, \text{où } \; m = T - t + 1.$$

- Notez que ce terme $\log^{1/2}(\dots) = \sqrt{(\log(\dots))}$ peut être *indéfini*, dès que $m < N_k(t)$, donc en pratique, $\sqrt{(\max(0, \log(\dots))}$ est préférable, ou alors un horizon *un peu plus grand* peut être utilisé à la place, pour rendre m un peu plus grand (e.g., $T' = 1.1T$).
- Si besoin, voir mon implémentation.

```
In [109]: class ApproximatedFHGittins(MoyenneEmpirique):
              """Algorithme ApproximatedFHGittins de Tor Lattimore."""

              def __init__(self, K, horizon, alpha=1):
                  """Crée l'instance de l'algorithme. Par défaut, alpha=1."""
                  super(ApproximatedFHGittins, self).__init__(K)  # On laisse la classe mère f
                  self.horizon = horizon
                  assert alpha >= 0, "Erreur : alpha doit etre >= 0."
                  self.alpha = alpha

              def choix(self):
                  """Si on a vu tous les bras, on prend celui d'indice le plus grand."""
                  self.t += 1      # Nécessaire ici
                  # 1er cas : il y a encore des bras qu'on a jamais vu
```

```
            if np.min(self.tirages) == 0:
                k = np.min(np.where(self.tirages == 0)[0])
            # 2nd cas : tous les bras ont été essayé
            else:
                # D'abord les moyennes
                moyennes_empiriques = self.recompenses / self.tirages
                # Puis le terme supplémentaire
                m = self.horizon - self.t + 1
                m_sur_Nk = m / self.tirages
                logdemi = np.sqrt(np.maximum(0, np.log(m_sur_Nk)))
                terme_sup = np.sqrt(self.alpha * np.log(m_sur_Nk / logdemi) / (2. * self
                indices = moyennes_empiriques + terme_sup
                k = np.argmax(indices)
            return k
```

2.11.1 Comparaison de `ApproximatedFHGittins` avec les autres algorithmes

Sur le même problème que précédemment, on va vérifier que ce dernier algorithme est plus efficace que les autres.

```
In [110]: horizon = 10000
          repetitions = 10
          mus = [0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.02, 0.02, 0.02, 0.01, 0.01, 0.005,
          bras = [ Bernoulli(mu) for mu in mus ]
          K = len(mus)
          kstar = np.argmax(mus)

In [111]: horizon, repetitions, mus, bras, K, kstar

Out[111]: (10000,
           10,
           [0.3,
            0.05,
            0.05,
            0.05,
            0.05,
            0.05,
            0.05,
            0.02,
            0.02,
            0.02,
            0.01,
            0.01,
            0.005,
            0.005,
            0.001,
            0.001],
           [<__main__.Bernoulli at 0x7f1810163898>,
            <__main__.Bernoulli at 0x7f1810163400>,
```

```
 <__main__.Bernoulli at 0x7f181035f080>,
 <__main__.Bernoulli at 0x7f181035fc50>,
 <__main__.Bernoulli at 0x7f181035feb8>,
 <__main__.Bernoulli at 0x7f181008f358>,
 <__main__.Bernoulli at 0x7f1810254c50>,
 <__main__.Bernoulli at 0x7f18102547f0>,
 <__main__.Bernoulli at 0x7f1810254e80>,
 <__main__.Bernoulli at 0x7f1810254eb8>,
 <__main__.Bernoulli at 0x7f1810254240>,
 <__main__.Bernoulli at 0x7f18100ca7f0>,
 <__main__.Bernoulli at 0x7f18100cacc0>,
 <__main__.Bernoulli at 0x7f18100caac8>,
 <__main__.Bernoulli at 0x7f18100ca1d0>,
 <__main__.Bernoulli at 0x7f18100c0860>],
16,
0)
```

```
In [112]: algorithmes = [UCB1(K, alpha=1), UCB1(K, alpha=0.5),
                         UCBV(K, alpha=1), UCBH(K, horizon, alpha=1),
                         KLUCB(K), ThompsonSampling(K),
                         ApproximatedFHGittins(K, 1.1 * T, alpha=4),
                         ApproximatedFHGittins(K, 1.1 * T, alpha=1),
                         ApproximatedFHGittins(K, 1.1 * T, alpha=0.5)
                        ]
          algorithmes
```

```
Out[112]: [<__main__.UCB1 at 0x7f18103a49b0>,
           <__main__.UCB1 at 0x7f18103a4358>,
           <__main__.UCBV at 0x7f180bf82550>,
           <__main__.UCBH at 0x7f180bf82eb8>,
           <__main__.KLUCB at 0x7f180bf82128>,
           <__main__.ThompsonSampling at 0x7f181005f1d0>,
           <__main__.ApproximatedFHGittins at 0x7f181038a438>,
           <__main__.ApproximatedFHGittins at 0x7f180bffa630>,
           <__main__.ApproximatedFHGittins at 0x7f180bffaf98>]
```

```
In [113]: noms = ["UCB1(alpha=1)", "UCB1(alpha=0.5)",
                  "UCBV(alpha=1)", "UCBH(alpha=1)",
                  "KLUCB", "ThompsonSampling",
                  "ApproximatedFHGittins(T, alpha=4)",
                  "ApproximatedFHGittins(T, alpha=1)",
                  "ApproximatedFHGittins(T, alpha=0.5)"
                 ]
```

On peut commencer la simulation, pour chaque algorithme.

```
In [114]: %%time
          N = len(algorithmes)
          recompenses, choix = np.zeros((N, repetitions, horizon)), np.zeros((N, repetitions, h
```

```
            # Pour chaque répétitions
            for rep in tqdm(range(repetitions), desc="Répétitions"):
                for i, alg in enumerate(algorithmes):
                    rec, ch = simulation(bras, alg, horizon)
                    recompenses[i, rep] = rec
                    choix[i, rep] = ch
```

```
HBox(children=(IntProgress(value=0, description='Répétitions', max=10), HTML(value='')))
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:24: RuntimeWarning: invalid value
```

```
CPU times: user 1min 16s, sys: 527 ms, total: 1min 17s
Wall time: 1min 16s
```

```
In [115]: # On moyenne
          recompenses_moy = np.mean(recompenses, axis=1)
          choix_moy = np.mean(choix, axis=1)
          print("Dimension de 'recompenses_moy' =", np.shape(recompenses_moy))  # juste pour v

          # On garde la distribution à t=T
          recompenses_fin = np.sum(recompenses, axis=2)
          choix_fin = np.sum(choix, axis=2)
          print("Dimension de 'recompenses_fin' =", np.shape(recompenses_fin))  # juste pour v
```

```
Dimension de 'recompenses_moy' = (9, 10000)
Dimension de 'recompenses_fin' = (9, 10)
```

Cette fois, les statistiques accumulées ne sont plus entières.

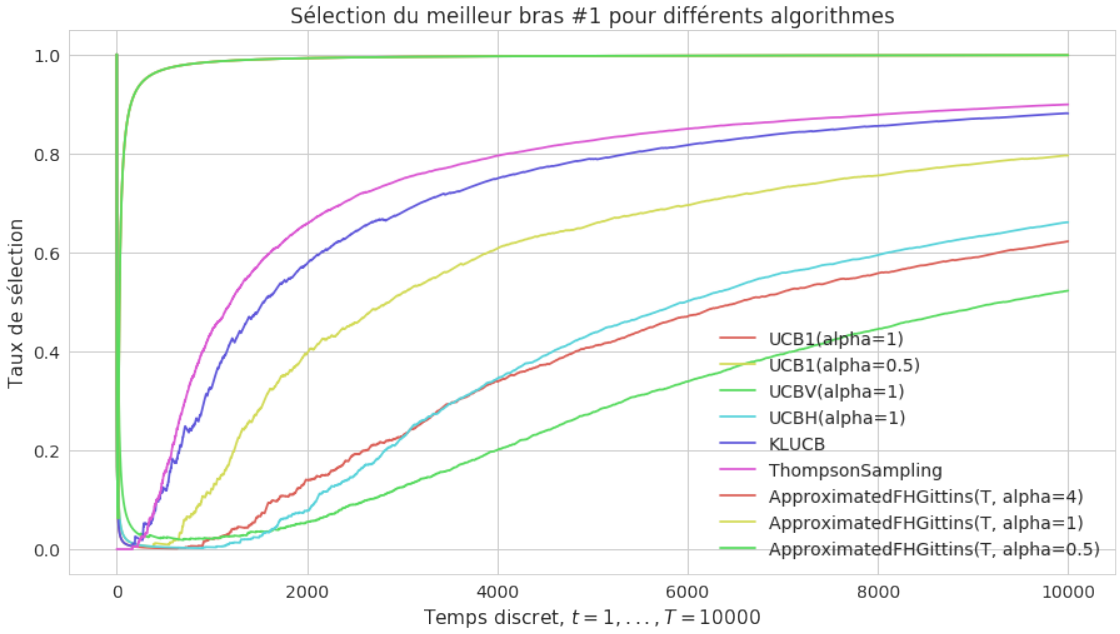
```
In [116]: recompenses_moy, choix_moy
```

```
Out[116]: (array([[0.4, 0.1, 0.3, ..., 0.3, 0.1, 0.3],
                  [0.2, 0. , 0.1, ..., 0.4, 0.2, 0.1],
                  [0.3, 0.2, 0. , ..., 0.3, 0.2, 0.3],
                  ...,
                  [0.3, 0.1, 0.1, ..., 0.2, 0.1, 0.3],
                  [0.3, 0. , 0. , ..., 0.4, 0.3, 0.4],
                  [0.3, 0.1, 0. , ..., 0.2, 0.1, 0.1]]),
           array([[0., 1., 2., ..., 0., 0., 0.],
                  [0., 1., 2., ..., 0., 0., 0.],
                  [0., 1., 2., ..., 0., 0., 0.],
                  ...,
                  [0., 1., 2., ..., 0., 0., 0.],
                  [0., 1., 2., ..., 0., 0., 0.],
                  [0., 1., 2., ..., 0., 0., 0.]]))
```

On affiche et vérifie les résultats attendus :

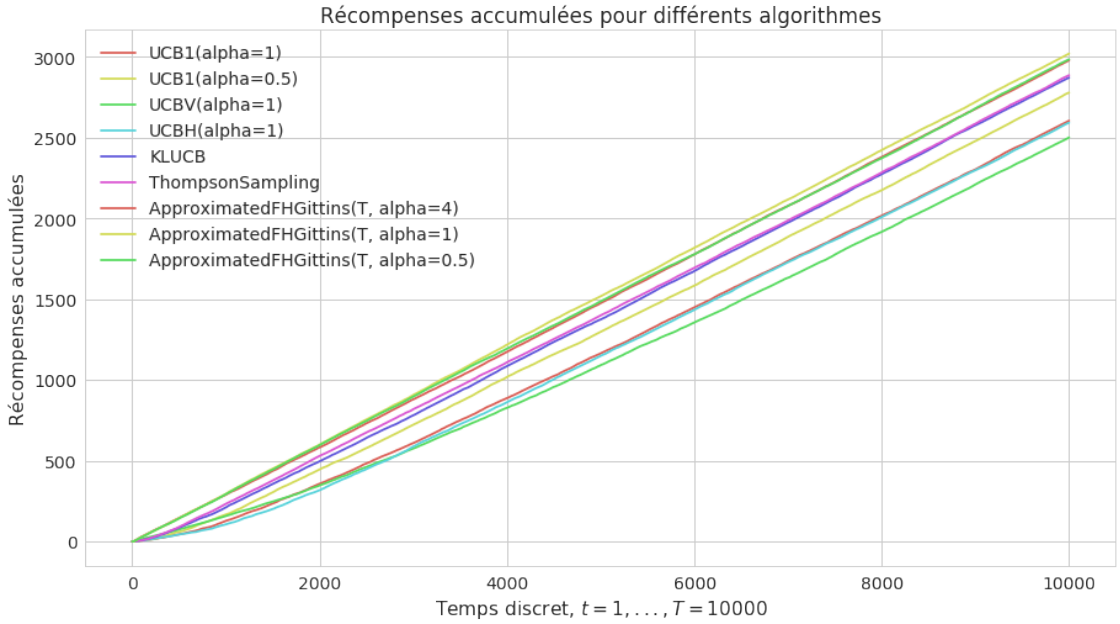
```
In [117]: choix_moy = np.floor(choix_moy)
```

```
In [118]: affiche_selections(choix_moy, noms, kstar)
```



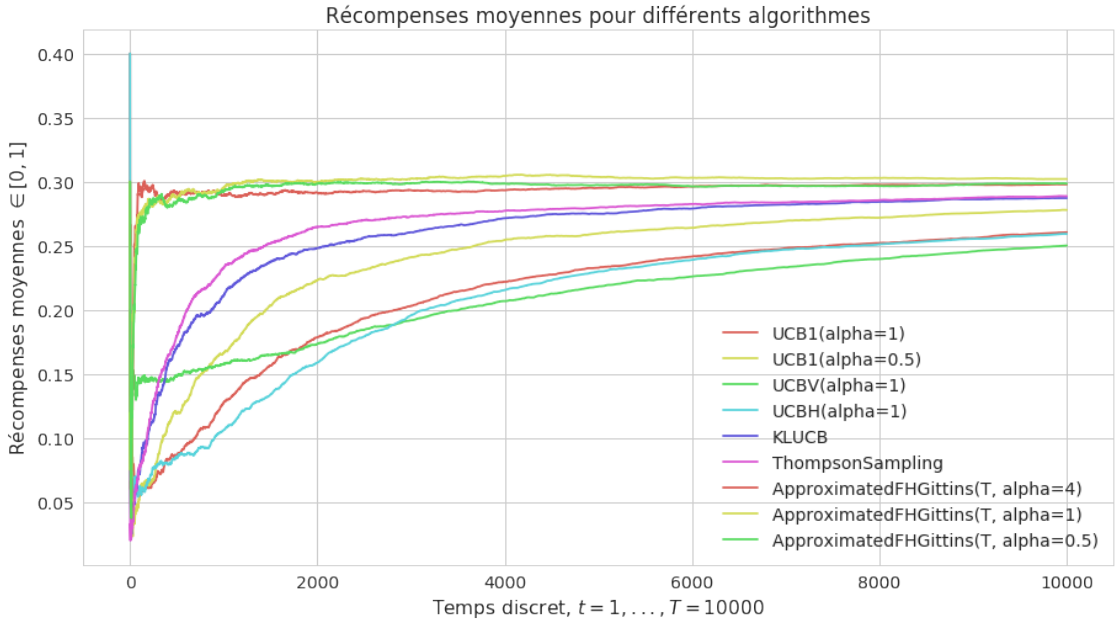
On commence à voir une différence de vitesse de convergence, pour l'identification du meilleur bras.

```
In [119]: affiche_recompenses(recompenses_moy, noms)
```



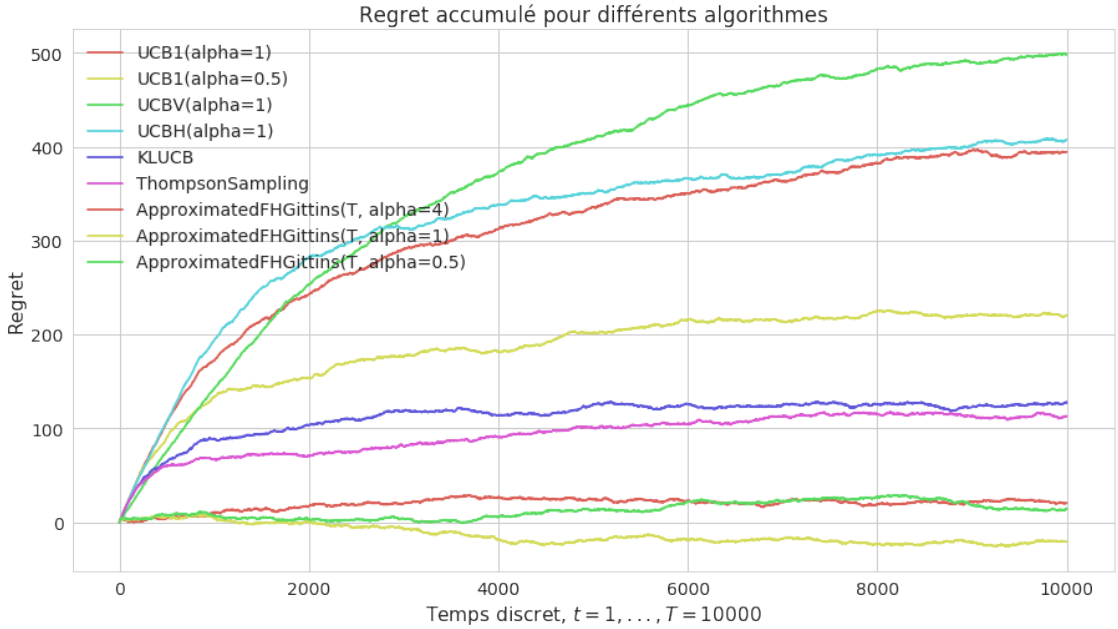
Difficile de différence quel algorithme est le plus efficace, même si clairement les plus grandes valeurs de $\alpha = 4,1$ pour UCB semblent avoir moins bien fonctionner.

```
In [120]: affiche_recompenses_moyennes(recompenses_moy, noms)
```



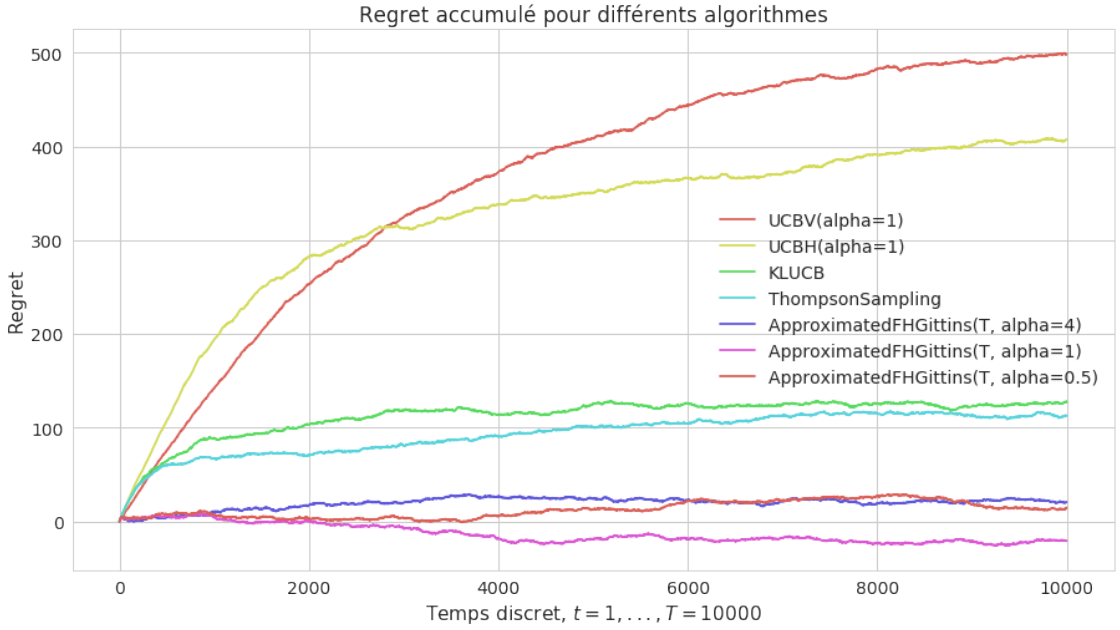
En terme de récompenses moyennes au cours du temps, les 4 algorithmes les plus efficaces convergent vers $\mu^* = \mu_1 = 0.9$.

```
In [121]: affiche_regret(recompenses_moy, noms, mustar=mus[kstar])
```

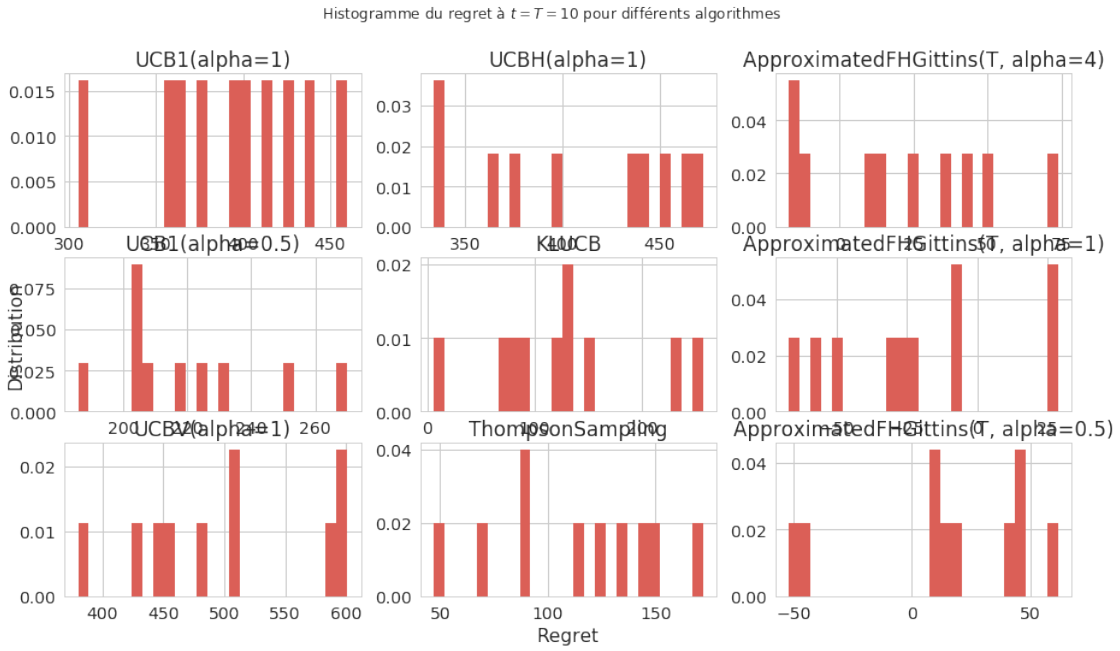


Note : un regret négatif n'est pas possible, mais sur peu d'expériences, on a une forte variance, et les échantillons tirés peuvent être suffisamment favorables pour avoir un regret faiblement négatif (comme c'est le cas ici pour un des `ApproximatedFHGittins`).

In [122]: `affiche_regret(recompenses_moy[2:], noms[2:], mustar=mus[kstar])`



In [124]: `affiche_hist_regret(recompenses_fin, noms, horizon, mustar=mus[kstar])`



2.12 Conclusion

J'espère que ce petit document a pu vous aider à mieux comprendre les bases de la simulation et l'implémentation de problèmes et d'algorithmes de bandits.

J'ai adopté une approche très modulaire, pour chaque composant de la simulation (bras, algorithmes, fonctions de simulation, d'affichages etc). C'est une façon de faire, bien-sûr ce n'est pas nécessaire, mais ça me semble clair, efficace et facile à lire et à compléter (si vous voulez rajouter un algorithme de plus).

- Pour plus de détails, je recommande de lire en détails ce petit article, datant de 2017, en français, écrit par Émilie Kaufmann.
- Cette petite simulation s'inspire de mon environnement plus lourd et plus complet, AlgoBandits.