DELPECH Chloé
PIGNIER Baptiste
PINPONT
02/05/2022

Projet mini BlockChain : Compte rendu

L'intégralité du code du projet est à retrouver par le dernier lien de ce document

INTRO :

La BlockChain est considérée par de nombreux informaticiens comme la principale avancée technologique du 21ème siècle, à côté de l'intelligence artificielle. Ce projet de mini blockchain nous amène à voir de manière abstraite le fonctionnement de la blockchain, sans s'attarder sur les détails techniques ou mathématiques des fonctions de hash utilisées. Comme nous le verrons, la blockchain peut se résumer à un système de clé de hash et de vérifications ordonnées.

Partie 1 : Mettre en place le principe de la BlockChain

0. Création des fonctions util à la réalisation de ce projet.

Dans la suite de ce compte rendu, plusieurs fonctions seront utilisées. Elles sont décrites ici, pour ne pas surcharger la mise en forme de ce compte rendu.

La première fonction est celle qui calculera la sommes des 5 premiers caractères du hash fournie en base 10. Elle est retrouvable ici. On convertit la chaine de caractère (string) en une liste d'entier (int)pour ensuite sommer en incrémentant une variable, les 5 premiers éléments de cette liste.

La seconde fonction calcule le hash en base 10 d'un bloc { clé 1, contenu, clé 2 } grâce au module *hashlib.* Elle est retrouvable ici. Le procéder étant explicite, il ne sera pas d'avantage détaillé.

La troisième fonction génère aléatoirement une clé aléatoire de longueur *x* prédéterminée.
Cette clé est une suite de letttre, de *a* à *z* et en minuscule. Elle est retrouvable ici.

La quatrième fonction génère la clé utilisée pour le chiffrement. Elle concatène une chaine aléatoire de lettre, (choisies par la fonction *clef*) avec les marqueurs de la forme « -000- » en fonction du rang du bloc (avec un maximum de 100. Ainsi, en lui fournissant en entrée la valeur *10,* cette fonction renvoie « -010-vhgvhrzopf-010- ». Elle est retrouvable ici.

1. On convertit chaque hash en base 10 et on regarde si la somme des 5 premiers termes est inférieure à 2 à l'aide de la fonction *Somme_5*. Si c'est le cas, alors notre hash est valide. Voici le code utilisé (en remplaçant la variable *a* par la valeur de la chaîne héxadécimale à vérifier).

Par exemple pour le premier hash, on a :

Héxadécimal : 0x4b3c4feeab8708750b12bd7a1ba03077
Base 10 : 100005258715127328605470008084938109047

Somme des 5 premiers caractères : 1
Le premier hash est donc valide.

Ainsi de suite, le hash n°2 n'est pas valide car la somme de ces 5 premiers termes en base 10 vaut 4. Le hash n°3 est valide car la somme de ces 5 premiers termes en base 10 est 1. Le hash n°4 n'est pas valide car la somme de ces 5 premiers termes en base 10 vaut 9.

2. Il faut calculer le hash de l'ensemble {clef de hash n°0 + bloc 1} pour chaque clé de hash n°0. Il faut alors que le hash en base 10 obtenu soit tel que la somme de ses 5 premiers termes soit strictement inférieur à 2.
Voici le code utilisé.
Voici les résultats obtenues:

La clef n°1 et la clef n°2 ne sont pas valide. La clef n°3 et la clef n°4 sont valide.

3. Il faut que la clef trouvée pour le bloc n°0 donne un hash dont la somme des 5 premiers caractères en base 10 soit strictement inférieure à 2. Pour cela, on peut utiliser une boucle while qui, tant que la somme des 5 premiers caractères du hash de l'ensemble {bloc n°0 + clé de hash n°0} n'est pas strictement inférieure à 2, va chercher une nouvelle valeur pour la clef. Voici le code utilisé.

Exemple de valeur trouvée :

La clef valide du bloc 0 est : -000-rwdntcwzx-000-
Le hash de l'ensemble 0 est : 10000673013066248993019945082032847681 2

4.

La mise en place de la blockchain se fait sous la forme d'objets. Un objet "Block" représente un bloc de la blockchain et un objet "Chain" représente un enchaînement d'objet "Block". Cette programmation orientée objet (POO) a pour avantage de manipuler plus facilement les différentes caractéristiques d'un bloc comme ses principales fonctions telles que l'application de fonction de hash ou la validation d'une clé de hash.

Pour calculer le hash, plusieurs méthodes sont possibles mais elles ne donnent pas toutes le même résultat. En fait, elles vont être différentes par rapport à la manière d'agencer les éléments qui constituent le bloc et par rapport à la manière de manipuler le fichier contenant les transactions.

Dans un second temps, nous avons remarqué qu'il était nécessaire d'insérer, entre la première clé et le contenu du bloc sur lequel on applique la fonction de hash, un saut de ligne. En effet, sans ce saut de ligne, le bloc ne s'agence pas correctement, et le résultat issu de la fonction de hash risque d'être défaillant. Il s'avère que cette modification est nécessaire pour répondre à la question 5. Une fois le hash en base 10 obtenu, il faut vérifier que la somme des 5 premiers caractères soit égale à 1.

Finalement, voici le bloc obtenu, avec en première ligne, la clé précédemment trouvée, puis le contenue du bloc (à savoir, les transactions de monnaies) et enfin, la clé de hash valide du bloc 1 :

```
-000-vemvkcrwvyev-000-
22106656        donne   4       coins a         22101498
22106351        donne   7       coins a         22106222
22111285        donne   1       coins a         22104461
22100139        donne   2       coins a         22106351

-001-ngyeerkoRqKXy-001-
```

5. Pour cette question, les deux clés nous sont imposées et nous devons trouver parmi plusieurs blocs de transaction, lequel correspondrait à un bloc valide, au regard des clés prédéterminées. Une simple itération sur les blocs proposés permet, en leur appliquant tour à tour, la fonction de hash, de vérifier quel(s) bloc(s) vérifie(nt) la condition sur la somme des 5 premiers caractère de leur hash en base 10. En utilisant ce code, on obtiens comme résultat :

Le bloc correct est le 3 ième bloc n°3

6. En créant un object Block de la classe du même nom, et en lui appliquant les clés imposées, son hash est :

100000638448729162339785838573392389502

Indépendamment des clés ou du contenu du bloc de transaction, il est impossible de reconstruire une clé de hash pouvant compenser la moindre modification apportée à la donnée appliquée à la fonction de hash. En effet, sauf cas de collision, (ce qui rendrait l'algorithme de hash vulnérable), le principe de hash permet le respect de l'intégrité des données. Seul un cas de collision serait un risque pour la blockchain. Heuresement, au vu de la puissance des algorithmes utilisées, personne n'a, à ce jour et à notre connaissance, réussi à compromettre la sécurité de la blockchain.

Partie 2 : Etude du temps nécessaire pour valider un maillon de la blockchain

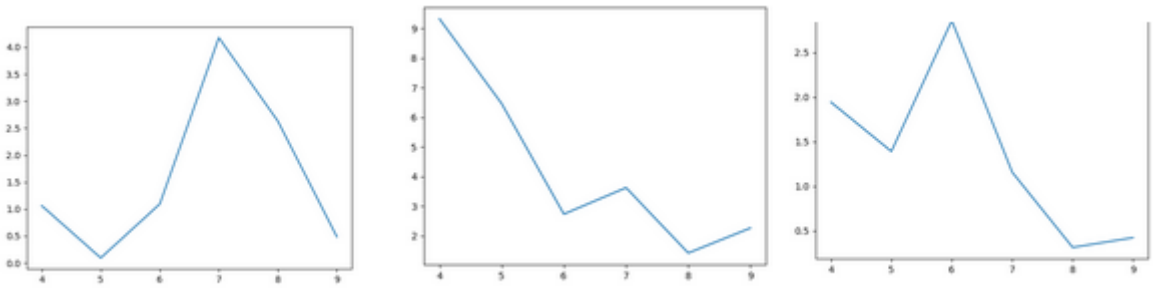
1.

On cherche à savoir comment évolue la durée nécessaire pour calculer une clef qui fonctionne en fonction de la longueur de la clef. On rappelle qu'une clef qui fonctionne doit être telle que l'ensemble { clef + bloc } donne un hash tel que la somme des premiers 5 termes soit strictement inférieure à 2 (on choisit une difficulté de 5).

On peut d'abord créer une fonction qui génère une clef dont on rentre la longueur.

Ensuite, on va donc créer une boucle *while* dans une « *for i in range (x) »*, avec x le nombre de points que l'on veut. La boucle tourne pour chaque *i* en recherchant une clef de longueur *i* et en calculant le temps mis pour trouver une clef qui fonctionne. On rentre les temps mis pour chaque longueur *i* dans une liste *T*, ainsi que les *i* dans une autre liste *I*. Voici le code utilisé.

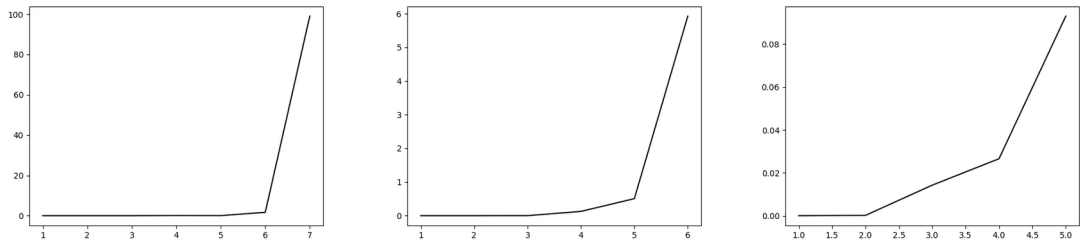
On essaye plusieurs fois et on regarde les courbes que l'on obtient.



On remarque que ces courbes ne sont ni croissantes, ni décroissantes et qu'elles sont différentes à chaque fois. On en conclut finalement que la longueur de la clef ne détermine pas vraiment le temps mis par le programme pour trouver la clef qui correspond. En effet, on peut très bien tomber sur la première clef qui fonctionne, comme la dernière. Il est aussi important de remarquer que pour une longueur de clef trop courte, le programme ne trouve pas toujours de clef qui fonctionne car il n'y a pas assez de possibilité. Par exemple, pour une clef de longueur 2, il y a 26 *26 possibilités de clef donc la probabilité d'en trouver une qui génère une hash dont la somme des 5 premiers termes soit inférieure à 2 est assez faible.

2. Pour cette étude, nous allons mettre en évidence l'impact de la difficulté sur le temps nécessaire à la machine pour calculer le hash.

Pour cela, nous allons utiliser le code suivant, qui calcule le hash d'un même bloc (le bloc de transaction 1 fourni avec l'énoncé de ce projet, muni des clés. A priori, la croissance de la courbe est exponentielle. Pour cette raison, nous allons réaliser la même mesure de manière progressive en faisant croître la limite de l'abscisse.



A posteriori, la croissance est réellement exponentielle. La difficultés est donc un paramètre très sensible qui peut avoir un énorme impact sur le temps de calcul. Il est nécessaire pour des raisons pratiques de la maintenir aussi bas que possible. Cependant, la difficulté est aussi un des principaux facteurs de sécurité garantissant la complexité de la clé et dont sa résistance face aux attaques. Un bon compromis entre temps de calcul et sécurité est donc primordiale au bon fonctionnement de la BlockChain.

ANNEXE :

somme :

```
def somme_5(hash_bloc):
    somme = 0
    list_hash = [int(c) for c in str(hash_bloc)]  # Je décompose mon hash en liste de chiffres pour pouvoir les additionner
    for i in range(5):
        somme = somme + list_hash[i]
    return(somme) # renvoi la somme des 5 premiers termes du hash
```

hash :

```
# fonction permettant d'obtenir un Hash pour un bloc
def hash(bloc_0):
    MdP = bloc_0
    MdP = MdP.encode('utf-8')
    m = hashlib.md5(MdP)
    a = m.hexdigest()
    a = int(a,16)     # convertir en base 10
    return(a)
```

clef :

```
def clef(x):
    c = []
    for i in range (x):
        j = random.randint(1,25)
        c.append(lettres[j])
        f = ''.join(c)
    return(f) # j'obtiens une clef de x lettres
```

genkey :

```
def gen_key(rank):
    rand_key = clef()
    delimiter = "".join(["0" for i in range(3-len(str(rank)))]+[str(rank)])  # convert 0 to 001 and 24 to 024
    end_key = "-"+str(delimiter) +"-"+ rand_key +"-"+str(delimiter)+"-"
    return end_key
```

Question 1)

```
a = "0x4b3c4feeab8708750b12bd7a1ba03077"
a = int(a,16) # Permet de convertir en base 10
somme = somme_5(a)
print("La somme des  premiers termes du hash en base 10 est",somme)
```

Question 2)

```
bloc_0 = open("blocs/bloc0.txt",'r').read()
bloc_0 = bloc_0 + "-000-uuqubeebuz-000-"
somme = somme_5(hash(bloc_0))
if somme < 2:
    print("La clef est valide")
else:
    print("La clef n'est pas valide")
```

Question 3)

```
bloc_0 = open("blocs/bloc0.txt",'r').read()

while somme_5(hash(bloc_0)) > 1:
    clef_hash_0 = '-000-' + clef() + "-000-"
    bloc_0 = bloc_0 + clef_hash_0

print("la clef valide du bloc 0 est :",clef_hash_0)
print("la hash de l'ensemble 0 est :",hash(bloc_0))
```

Question 5)

```
block = Block(2,"-002-zityslwbhb-002-")
for i in range(1,5):
    print()
    block.setFilePath("Bloc3/bloc3n°"+str(i)+".txt")
    result = block.hash("-003-nosfbpbxmv-003-")
    if somme_5(result) < 2:
        print("Le bloc correct est le "+str(i)+" ièmebloc n°3")
```

Partie 2 Question 1

```
for i in range (4,10):
    t1 = time.time()
    bloc_0 = '000-' + clef(i) + '-000'
    while somme_5(hash(bloc_0)) > 1: #si somme des 5 premiers termes du hash > 1 on continue de chercher une clef
        bloc_0 = '000-' + clef(i) + '-000'
    t2 = time.time()
```

lien github : https://github.com/BaptistePignier/Projet-MiniBlockChain/