Implémentation d'une Blockchain Proof of work avec Python

La technologie blockchain, tout d'abord perçue comme le protocole sous-jacent de bitcoin s'est peu à peu imposée comme une technologie au potentiel disruptif immense.
Pour comprendre la technologie blockchain, nous pouvons la comparer à une feuille de calcul google sheet partagée qui permet à plusieurs utilisateurs de collaborer au sein d'un même document simultanément. Toutefois contrairement à une feuille de calcul google , une blockchain n'est pas hébergée par les serveurs Google mais de nombreux ordinateurs en même temps, appelés aussi "noeuds". Une des grande particularité des blockchains est donc la façon d'y inscrire de nouvelles informations Contrairement à une feuille de calcul où vous pouvez écrire et effacer ce que vous voulez, les blockchains utilisent des mécanismes de consensus pour s'assurer qu'on y insérer des informations unilatéralement sans un accord de la communauté, et impossible de modifier les informations déjà présente.

Ce projet est destiné à être une brève introduction à la mise en œuvre d'une Blockchain Proof of Work (PoW). Nous allons mettre en œuvre différents aspects de la technologie Blockchain à l'aide du langage de programmation Python que nous aimons tant 😀. Nous allons découvrir ensemble :
- La structure de données d'une Blockchain en utilisant la programmation orientée objet
- La simulation du preuve de travail
- La compréhension du concept de difficulté à trouver le prochain hachage de bloc.
- Une simulation de plusieurs mineurs avec des puissances de calcul variables.
- Un peu d'analyse de données pour voir si ce que nous avons mis en œuvre a un sens.

- Block.py : Une blockchain peut être considérée comme un ensemble de bloc interconnecté. Ci-dessous nous avons défini la classe Block, et une première fonction d'initialisation qui nous permet de créer un nouvel objet bloc en fonction de certains paramètres. Ensuite une deuxième fonction de hachage des variables de classe.

Title

```python
import hashlib as hasher

class Block:

    # A function that creates a new block given some parameters
    def __init__(self, index, timestamp, data, previous_hash, nonce=0):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.nonce = nonce
        self.previous_hash = previous_hash
        self.hash = self.hash_block()

    # A function that computes the hash of this block based on its class variables.
    def hash_block(self):
        sha = hasher.sha256()
        block_hash = (str(self.index) + str(self.timestamp) + str(self.data) + str(self.previous_hash) +
str(self.nonce))
        block_hash = block_hash.encode('utf-8')
        sha.update(block_hash)
        return sha.hexdigest()
```

- Chain_function.py : Le premier bloc d'une blockchain est nommé "Genesis-Block", la fonction **create_genesis_block()** nous permet de créer ce premier bloc. Pour assurer l'intégrité des données, chaque bloc doit détenir les informations de hash du bloc qui le précède, dans la function **next_block()** nous créons le bloc suivant en fonction de la dernière block créée. La dernière function **create_complete_chain()** permet de créer une chaîne en entier en fonction d'un nombre de bloc total donné.

Title

```python
from datetime import datetime
from main import Block

# Function that creates the first block with current time and generic data
def create_genesis_block():
    return Block(0, datetime.now(), "Genesis Block", "0")

# function that creates the next block, given the last block on the chain you# want to mine on
def next_block(last_block, nonce=0):
    this_index = last_block.index + 1
    this_timestamp = datetime.now()
    this_data = "Hey! I'm block" + str(this_index)
    this_previous_hash = last_block.hash
    return Block(this_index, this_timestamp, this_data, this_previous_hash)

# function that creates a complete blockchain based on the given numbers of block
def complete_chain(num_blocks, blockchain, previous_block):
    for i in range(0, num_blocks):
        block_to_add = next_block(previous_block)
        blockchain.append(block_to_add)
        previous_block = block_to_add
        print("Block #{} has been added to the blockchain!".format(block_to_add.index))
        print("Hash: {}\n".format(block_to_add.hash))
```

- proof_of_work.py : le Proof of Work ou preuve de travail est un système de validation permettant de repousser dans un environnement client-serveur des attaques par déni de service ou autres abus de services (spams). Nous allons utiliser ce système comme consensus à respecter lors de la création d'un nouveau bloc. J'ai défini une fonction **generate_nonce()** qui génère aléatoirement un nombre pour le nonce qui est un nombre qui ne peut être utilisé qu'une seule fois. Comme preuve de travail, le serveur requiert du client d'effectuer une petite tâche sous forme d'un calcul. Une caractéristique essentielle du concept de *preuve de travail* est l'asymétrie du coût de mise en œuvre. En effet, le travail doit être difficilement réalisable pour l'auteur de la requête ou mineur, la fonction **generate_difficulty_bound()** nous permet de générer une limite de la difficulté à trouver le nonce. La fonction **find_next_block()** prend en entrée le dernier bloc créé et le métrique de difficulté, elle nous permet de trouver le nonce pour valider la création d'un nouveau bloc.

Title

```python
import random as rand
import time
from main import Block
from function import chain_function

def generate_nonce(length=20):
    return ''.join([str(rand.randint(0, 9)) for i in range(length)])

def generate_difficulty_bound(difficulty=1):
    diff_str = ""
    for i in range(difficulty):
        diff_str += '0'
    for i in range(64 - difficulty):
        diff_str += 'F'
    diff_str = "0x" + diff_str
    return int(diff_str, 16)

# Given a previous block and a difficulty metric, finds a nonce that results in a lower hash value

def find_next_block(last_block, difficulty, nonce_length):
    difficulty_bound = generate_difficulty_bound(difficulty)
    start = time.process_time()
    new_block = chain_function.next_block(last_block)
    hashes_tried = 1
    while int(new_block.hash,16) > difficulty_bound:
        nonce = generate_nonce(nonce_length)
        new_block = Block(new_block.index, new_block.timestamp, new_block.data,new_block.previous_hash, nonce)
        hashes_tried += 1
    time_taken = time.process_time() - start
    return time_taken, hashes_tried, new_block

```

- Miner_function.py : Dans une blockchain Proof of Work, les mineurs sont chargés de la création des blocs. Le premier à trouver le nonce crée le bloc, en général c'est le mineur qui a le plus de puissance de calcul sur sa machine. Ici nous définissons la classe "MinerNodeNaive" et une fonction d'initialisation init() qui crée un nouveau mineur ainsi qu'une fonction try_hash(), qui permet à un mineur de faire la course avec les autres mineurs pour voir qui peut obtenir un certain nombre de blocs en premier.

Title

```python
from main import Block
import datetime as date
import proof_of_work as PoW

class MinerNodeNaive:
    def __init__(self, name, compute):
        self.name = name
        self.compute = compute

    def try_hash(self, diff_value, chain):
        last_block = chain[-1]
        difficulty = PoW.generate_difficulty_bound(diff_value)
        date_now = date.datetime.now()
        this_index = last_block.index + 1
        this_timestamp = date_now
        this_data = "Hey! I'm block " + str(this_index)
        this_hash = last_block.hash
        new_block = Block(this_index, this_timestamp, this_data, this_hash)
        if int(new_block.hash, 16) < difficulty:
            chain.append(new_block)
            print("Block #{} has been added to the blockchain!".format(new_block.index))
            print("Block found by: {}".format(self.name))
            print("Hash: {}\n".format(new_block.hash))

```

Exemple : Le code ci-dessous illustre une simulation de validation des blocs qui respecte le consensus de preuve de travail, Comme valideurs ou mineurs nous avons des groupes d'étudiants de différentes grandes écoles (ESMT, UCAD, ESP, IAM) avec chacun une puissance de computation qui lui est propre. Chaque groupe essaye de trouver en premier le nonce pour créer un nouveau bloc sur la blockchain

test > simulation_miner.py > ...

```python
from function.miner import MinerNodeNaive
from main import Block
import random as rand

#In this file I try to simulate a bunch of different miners with different compute powers.
# However this isn't completely indicative of how a real system works.

#Initialize multiple miners on the network
ESMT_Miner = MinerNodeNaive("ESMT Miner", 10)
UCAD_Miner = MinerNodeNaive("UCAD Miner", 5)
ESP_Miner = MinerNodeNaive("ESP Miner", 2)
ISM_Miner = MinerNodeNaive("ISM Miner", 1)

miner_array = [ESMT_Miner, UCAD_Miner, ESP_Miner, ISM_Miner]

def create_compute_simulation(miner_array):
    compute_array = []
    for miner in miner_array:
        for i in range(miner.compute):
            compute_array.append(miner.name)
    return(compute_array)

compute_simulation_array = create_compute_simulation(miner_array)
rand.shuffle(compute_simulation_array)

chain_length = 20
blockchain_distributed = [Block.create_genesis_block()]
genesis_block_dist = blockchain_distributed[0]
chain_difficulty = [rand.randint(2,4) for i in range(chain_length)]

for i in range(len(chain_difficulty)):
    while len(blockchain_distributed) < i + 2:
        next_miner_str = rand.sample(compute_simulation_array, 1)[0]
        next_miner = ESMT_Miner #random default (go bears)
        for miner in miner_array:
            if next_miner_str == miner.name:
                next_miner = miner
        next_miner.try_hash(chain_difficulty[i], blockchain_distributed)

```

Nous avons illustré ci-dessus les principaux aspects fonctionnels d'une blockchain proof of work. Le code des fonctionnalités et les fonctions de test sont accessible via ce lien : https://github.com/sysall/PoW_Blockchain_Python

Arborescence du projet :

- un **dossier function** qui regroupe toute les fonctionnalités qu'on a défini au débat : chain_fonction.py, proof_of_work.py, miner.py
- un **dossier test** qui regroupe 3 fichiers de simulation des fonctionnalités : simulation_chain.py, simulation_miner.py, simulation_PoW.py
- un **fichier main.py** où la classe block et ses fonctions sont définis
- un **fichier data_analytic.py** pour un peu d'analyse des données notamment les facteurs qui influent sur l'augmentation du niveau de difficulté.