½ Module : Blockchain
Pr. Ikram BENABDELOUAHAB
2ème année Master IASD
2025/2026

ATELIER 2 : AUTOMATE CELLULAIRE ET FONCTION DE HACHAGE DANS LA BLOCKCHAIN (C++)

Questions :

1. Implémente en C++ un automate cellulaire 1D avec état binaire et voisinage r = 1.

1.1. Crée une fonction **init_state()** pour initialiser l'état à partir d'un vecteur de bits.
1.2. Implémente une fonction **evolve()** qui applique la règle de transition donnée (Rule 30, Rule 90, ou Rule 110).
1.3. Vérifie que ton automate reproduit correctement la règle choisie sur un petit état initial.

2. Implémente une fonction de hachage basée sur ton automate cellulaire :

2.1. Fonction demandée :

std::string ac_hash(const std::string& input, uint32_t rule, size_t steps);

2.2. Décris le mode de conversion du texte d'entrée en bits.
2.3. Explique le processus utilisé pour produire un hash final fixe de 256 bits.
2.4. Vérifie par un test que deux entrées différentes donnent deux sorties différentes.

3. Intègre la fonction ac_hash() dans la blockchain existante :

3.1. Ajoute une option de sélection du mode de hachage (SHA256 ou AC_HASH).
3.2. Modifie le code du minage pour utiliser **ac_hash()** lors du calcul du hash de bloc.
3.3. Vérifie que la validation de bloc reste fonctionnelle avec cette nouvelle fonction.

4. Compare le comportement de ac_hash et de SHA256 :

4.1. Mesure le temps moyen de minage de 10 blocs avec chaque méthode.
4.2. Mesure le nombre moyen d'itérations pour trouver un hash valide avec difficulté fixe.
4.3. Donne les résultats dans un tableau.

5. Analyse l'effet avalanche de ac_hash :

5.1. Calcule le pourcentage moyen de bits différents entre les hashes de deux messages ne différant que par un bit.
5.2. Donne le résultat numérique et ton code de test.

6. Analyse la distribution des bits produits par ac_hash :

6.1. Calcule le pourcentage de bits à 1 sur un échantillon d'au moins 10^5 bits.
6.2. Indique si la distribution est équilibrée (≈50 % de 1).

7. Teste plusieurs règles d'automates :

7.1. Exécute ac_hash avec Rule 30, Rule 90 et Rule 110.
7.2. Compare la stabilité des résultats et les temps d'exécution.
7.3. Indique quelle règle te semble la plus adaptée pour le hachage et pourquoi.

8. Décris en quelques lignes les avantages potentiels d'un hachage basé sur automate cellulaire dans une blockchain.
9. Décris en quelques lignes les faiblesses ou vulnérabilités possibles de cette approche.
10. Propose une amélioration ou une variante (ex. combinaison AC + SHA256, règle dynamique, voisinage variable).
11. Fournis les résultats de tous tes tests (hashs, temps, avalanche, uniformité) sous forme de tableau clair dans le rapport.
12. Ajoute dans le code un mode d'exécution automatique (run_tests.sh ou make test) qui produit ces résultats.

ANNEXE

1. RÔLE DE LA CRYPTOGRAPHIE DANS UNE BLOCKCHAIN

Dans une blockchain, la **cryptographie** garantit :

- **l'intégrité** des blocs (hachage),
- **l'authenticité** des transactions (signatures),
- **la sécurité du consensus** (preuve de travail, preuve d'enjeu, etc.).

Chaque bloc contient un **hash** du bloc précédent, ce qui lie toute la chaîne de manière immuable.
Si la fonction de hachage change, tout le mécanisme de sécurité du réseau change aussi.

2. POURQUOI INTRODUIRE LES AUTOMATES CELLULAIRES ?

Les **automates cellulaires (AC)** sont des systèmes dynamiques discrets capables de produire une **complexité élevée** à partir de **règles locales simples**.
Certaines règles, comme ***Rule 30*** ou ***Rule 110***, présentent :

- une **forte non-linéarité**,
- un **effet avalanche naturel** (petit changement → grand impact),
- une **difficulté de prédiction** (comportement chaotique).

Ces propriétés sont recherchées en **cryptographie** — elles rappellent celles d'un bon **algorithme de hachage**.

3. UTILISATION CRYPTOGRAPHIQUE DES AC DANS UNE BLOCKCHAIN

3.1. AC comme fonction de hachage

L'idée principale est de remplacer la fonction standard (SHA-256) par une **fonction de hachage dérivée d'un automate cellulaire** :

- L'entrée du bloc (index, timestamp, transactions, nonce, etc.) est convertie en **état initial binaire**.
- L'automate évolue selon une **règle donnée** (par ex. Rule 30) pendant un certain nombre d'itérations.
- L'état final (ou une combinaison des états successifs) devient le **hash du bloc**.

Cette approche permet :

- de générer des **hashes uniques et difficiles à inverser**,
- d'obtenir un **comportement imprévisible**,
- d'expérimenter de nouvelles architectures cryptographiques légères (utile en **IoT-blockchain**).

3.2. AC comme générateur pseudo-aléatoire

Les AC peuvent aussi produire des **séquences pseudo-aléatoires** à partir d'un état initial.
Ces séquences peuvent servir :

- à la génération de **nonces** pour le minage,
- à la **construction de clés** temporaires,
- à des **protocoles de consensus aléatoires**

3.3. AC dans la preuve de travail (PoW)

Au lieu de chercher un hash inférieur à une cible numérique (comme avec SHA-256),
on peut définir une **preuve de travail basée sur la complexité d'évolution d'un automate cellulaire**.
Exemple : Trouver un **nonce** qui, combiné aux données du bloc, produit un état AC ayant une configuration particulière (ex. nombre de cellules vivantes précis, motif symétrique, etc.).
Cette approche rend le **minage plus diversifié** et peut introduire des **formes alternatives de difficulté**.

4. EXEMPLE D'INTÉGRATION CONCRÈTE

Dans votre blockchain éducative (implémentée en C++) :

1. Remplacer la fonction sha256(block_data) par ac_hash(block_data, rule, steps).
2. Lancer le minage avec :

```
while (ac_hash(block_data + nonce, 30, 128).substr(0, 4) != "0000") {
   nonce++;
}
```

3. Observer :
   - la **vitesse du minage**,
   - la **stabilité du hash**,
   - et la **distribution des bits** selon la règle choisie.