TP4 - Codes détecteurs et correcteurs d’erreur

1 Code détecteur d’erreur

Question 1 :

Le code de parité est un simple mécanisme d'erreur pour la détection des erreurs dans les données. Pour un message donné, il ajoute un bit de parité pour indiquer si le nombre de bits 1 dans le message est pair ou impair.

Voici une fonction Python qui prend un message en entrée et renvoie le message avec le bit de parité ajouté :

```
def add_parity_bit(message):
    # Convertir le message en une chaîne binaire
    binary_message = ''.join(format(ord(c), '08b') for c in message)

    # Calculer le nombre de bits 1
    num_ones = binary_message.count('1')

    # Déterminer le bit de parité (1 si impair, 0 si pair)
    parity_bit = '1' if num_ones % 2 != 0 else '0'

    # Retourner le message binaire avec le bit de parité ajouté
    return binary_message + parity_bit

# Exemple d'utilisation
message = "hello"
parity_message = add_parity_bit(message)
print(f"Message original : {message}")
print(f"Message avec bit de parité : {parity_message}")
```

Explications

1. **Conversion du message en binaire** :
    - `format(ord(c), '08b')` convertit chaque caractère en son équivalent binaire sur 8 bits.
    - `''.join(...)` combine ces représentations binaires en une seule chaîne binaire.

2. **Calcul du nombre de bits 1** :
    - `binary_message.count('1')` compte le nombre de bits 1 dans le message binaire.
3. **Détermination du bit de parité** :
    - Si le nombre de bits 1 est impair, le bit de parité est `1`, sinon il est `0`.
4. **Ajout du bit de parité au message binaire** :
    - Le bit de parité est ajouté à la fin du message binaire.

Question 2 :

Pour améliorer la détection d'une multitude d'erreurs, nous pouvons utiliser le code de parité bidimensionnel, également connu sous le nom de "parité matricielle" ou "parité longitudinale". Ce code ajoute des bits de parité pour chaque ligne et chaque colonne d'un bloc de données, ce qui permet de détecter non seulement les erreurs simples, mais aussi certaines erreurs multiples.

Voici comment fonctionne la parité bidimensionnelle :

1. **Diviser le message en blocs** : Diviser le message en blocs de taille fixe.
2. **Ajouter des bits de parité** : Ajouter des bits de parité à la fin de chaque ligne et colonne du bloc.
3. **Vérifier les parités** : Lors de la vérification, recalculer les bits de parité pour chaque ligne et colonne et les comparer aux bits de parité reçus.

Implémentation en Python

Fonction pour ajouter des bits de parité bidimensionnels

```
def add_parity_2d(message, block_size):
    # Convertir le message en une liste de caractères
    chars = list(message)

    # Diviser le message en blocs de taille block_size
    blocks = [chars[i:i + block_size] for i in range(0, len(chars), block_size)]

    # Ajouter des bits de parité pour chaque ligne
    for block in blocks:
        num_ones = sum(format(ord(c), '08b').count('1') for c in block)
        parity_bit = '1' if num_ones % 2 != 0 else '0'
        block.append(parity_bit)

    # Ajouter des bits de parité pour chaque colonne
    transposed_blocks = list(zip(*blocks))
    for col in transposed_blocks:
        num_ones = sum(format(ord(c), '08b').count('1') for c in col)
        parity_bit = '1' if num_ones % 2 != 0 else '0'
```

```
        col = list(col)
        col.append(parity_bit)

    # Combiner les blocs avec les bits de parité
    parity_blocks = [block + [parity_bit] for block, parity_bit in zip(blocks, transposed_blocks)]

    # Convertir les blocs de parité en une chaîne de caractères
    parity_message = ''.join([''.join(block) for block in parity_blocks])

    return parity_message

# Exemple d'utilisation
message = "hello world"
block_size = 5
parity_message = add_parity_2d(message, block_size)
print(f"Message original : {message}")
print(f"Message avec bits de parité 2D : {parity_message}")
```

Fonction pour vérifier les bits de parité bidimensionnels

```
def check_parity_2d(parity_message, block_size):
    # Diviser le message en blocs de taille block_size + 1 (pour inclure les bits de parité)
    chars = list(parity_message)
    blocks = [chars[i:i + block_size + 1] for i in range(0, len(chars), block_size + 1)]

    # Vérifier les bits de parité pour chaque ligne
    for block in blocks:
        num_ones = sum(format(ord(c), '08b').count('1') for c in block[:-1])
        expected_parity = '1' if num_ones % 2 != 0 else '0'
        if block[-1] != expected_parity:
            return False

    # Vérifier les bits de parité pour chaque colonne
    transposed_blocks = list(zip(*blocks))
    for col in transposed_blocks:
        num_ones = sum(format(ord(c), '08b').count('1') for c in col[:-1])
        expected_parity = '1' if num_ones % 2 != 0 else '0'
        if col[-1] != expected_parity:
            return False

    return True

# Exemple d'utilisation
is_valid = check_parity_2d(parity_message, block_size)
print(f"Le message avec bits de parité 2D est {'valide' si is_valid else 'invalide'}.")
```

Explications

1. **Diviser le message en blocs** : Le message est divisé en blocs de taille `block_size`.
2. **Ajouter des bits de parité pour chaque ligne** : Pour chaque ligne du bloc, nous comptons les bits 1 et ajoutons un bit de parité.
3. **Ajouter des bits de parité pour chaque colonne** : Nous faisons la même chose pour chaque colonne en utilisant une transposition des blocs.
4. **Vérification des bits de parité** : Lors de la vérification, nous recalculons les bits de parité pour chaque ligne et colonne et comparons avec les bits de parité reçus.

Ce mécanisme permet de détecter non seulement les erreurs simples mais aussi certaines erreurs multiples, rendant le système plus robuste pour la détection d'erreurs.

Questions 3

Pour implémenter une fonction qui prend en entrée un message `a` et un paramètre `k` et renvoie le résultat après application de `k` répétitions, nous devons définir ce que signifie "k répétitions" dans ce contexte. Une interprétation courante est de répéter l'application d'une transformation donnée (comme un hachage ou une autre opération) `k` fois.

Supposons que nous voulons répéter l'application d'un code de parité `k` fois sur le message. Voici comment nous pouvons implémenter cela en Python :

Implémentation

Nous allons d'abord définir la fonction de code de parité, puis implémenter la fonction qui applique ce code de parité `k` fois.

Fonction pour appliquer `k` répétitions

```
def apply_k_repetitions(message, k):
    result = message
    for _ in range(k):
        result = add_parity_bit(result)
    return result

# Exemple d'utilisation
message = "hello"
k = 3
result = apply_k_repetitions(message, k)
print(f"Message original : {message}")
print(f"Message après {k} répétitions du code de parité : {result}")
```

Explications

1. **Fonction `add_parity_bit`** :

- Convertit chaque caractère du message en binaire.
- Compte les bits 1 dans la représentation binaire.
- Ajoute un bit de parité (`1` si le nombre de bits 1 est impair, `0` si pair).

2. **Fonction `apply_k_repetitions`** :
   - Prend un message et un paramètre `k`.
   - Applique la fonction `add_parity_bit` `k` fois au message.

Question 4)

Le code de Hamming est un code de correction d'erreurs qui peut détecter et corriger des erreurs simples dans les données. Nous allons implémenter une fonction qui applique le code de Hamming à un message. Pour simplifier, nous allons travailler avec le code de Hamming (7,4), qui encode des blocs de 4 bits en des blocs de 7 bits avec 3 bits de parité.

Étapes de l'Implémentation

1. **Convertir le message en binaire**.
2. **Diviser le message en blocs de 4 bits**.
3. **Calculer les bits de parité pour chaque bloc**.
4. **Assembler les blocs encodés avec les bits de parité**.

Fonction de Code de Hamming

Fonction pour convertir un message en binaire

```
def text_to_bin(message):
    return ''.join(format(ord(char), '08b') for char in message)

def bin_to_text(binary_message):
    chars = [binary_message[i:i+8] for i in range(0, len(binary_message), 8)]
    return ''.join(chr(int(char, 2)) for char in chars)
```

Fonction pour appliquer le code de Hamming (7,4)

```
def hamming_encode_4bit(block):
    # Calculate the parity bits
    p1 = (int(block[0]) + int(block[1]) + int(block[3])) % 2
    p2 = (int(block[0]) + int(block[2]) + int(block[3])) % 2
    p3 = (int(block[1]) + int(block[2]) + int(block[3])) % 2

    # Return the 7-bit encoded block
    return f'{p1}{p2}{block[0]}{p3}{block[1:]}'

def hamming_encode(message):
    binary_message = text_to_bin(message)
    # Split binary message into 4-bit blocks
    blocks = [binary_message[i:i+4] for i in range(0, len(binary_message), 4)]
```

```
    encoded_message = ''
    for block in blocks:
        # Pad the block if it's less than 4 bits
        if len(block) < 4:
            block = block.ljust(4, '0')
        encoded_message += hamming_encode_4bit(block)

    return encoded_message

# Exemple d'utilisation
message = "hello"
encoded_message = hamming_encode(message)
print(f"Message original : {message}")
print(f"Message encodé avec le code de Hamming : {encoded_message}")
```

Explications

1. **Conversion du texte en binaire** :
    - `text_to_bin` : Convertit chaque caractère du message en une chaîne binaire de 8 bits.
    - `bin_to_text` : Convertit une chaîne binaire en texte.
2. **Encodage Hamming (7,4)** :
    - `hamming_encode_4bit` : Calcule les bits de parité pour un bloc de 4 bits et retourne un bloc encodé de 7 bits.
    - `hamming_encode` : Divise le message en blocs de 4 bits, applique l'encodage Hamming à chaque bloc, et combine les blocs encodés.

Note sur la Correction d'Erreurs

Cette implémentation montre comment encoder un message avec le code de Hamming pour détecter et corriger des erreurs simples. La correction d'erreurs, qui n'est pas implémentée ici, impliquerait la vérification des bits de parité et l'identification des erreurs en utilisant les syndromes des bits de parité.

2 Code correcteur d’erreur

Question 5

Pour implémenter une fonction permettant de retrouver le message originel u à partir d'un message v codé avec k répétitions et qui a pu être altéré lors de l'envoi, nous devons suivre ces étapes :

1. **Diviser le message reçu v en blocs de la longueur correspondant à une seule répétition.**

2. **Compter les occurrences de chaque bloc pour chaque position.**
3. **Déterminer le bloc le plus fréquent pour chaque position pour retrouver le message original.**

Voici une implémentation en Python pour ce processus :

Implémentation

Fonction pour retrouver le message originel à partir du message codé v

```
def retrieve_original_message(v, k):
    # La longueur d'un bloc codé
    block_length = len(v) // k

    # Diviser le message en k blocs
    blocks = [v[i * block_length:(i + 1) * block_length] for i in range(k)]

    # Reconstituer le message original en utilisant le vote majoritaire pour chaque position
    original_message_bits = []

    for i in range(block_length):
        # Compter les occurrences de '0' et '1' pour la position i
        count_0 = sum(1 for block in blocks if block[i] == '0')
        count_1 = sum(1 for block in blocks if block[i] == '1')

        # La valeur la plus fréquente à la position i est ajoutée au message original
        original_message_bits.append('0' if count_0 > count_1 else '1')

    # Convertir la liste de bits en une chaîne binaire
    original_message = ''.join(original_message_bits)

    return original_message

# Exemple d'utilisation
v = "011010000110010101101100011011000110111101" * 3  # "hello" codé avec 3 répétitions
# Altérer un peu le message pour simuler les erreurs de transmission
v = v[:5] + '1' + v[6:]  # Modifier le 6ème bit du premier bloc

# Retrouver le message original
u = retrieve_original_message(v, 3)
print(f"Message original reconstitué : {u}")
```

Explications

1. **Diviser le message v en blocs** :
    - Nous divisons le message v en k blocs de longueur égale.

2. **Compter les occurrences pour chaque position** :
    - Pour chaque position dans le bloc, nous comptons les occurrences de '0' et '1' parmi les k blocs.
3. **Déterminer la valeur la plus fréquente** :
    - Pour chaque position, nous choisissons la valeur ('0' ou '1') qui apparaît le plus souvent parmi les k blocs.
4. **Assembler le message original** :
    - Nous assemblons les bits déterminés pour chaque position en une chaîne binaire représentant le message original.

Note

Cette approche utilise la technique du vote majoritaire pour corriger les erreurs dans le message codé avec des répétitions. Elle suppose que la majorité des blocs reçus à chaque position sont corrects. En cas d'erreurs multiples dans les blocs, il est possible que le message reconstitué soit incorrect, mais cette méthode reste efficace pour corriger un petit nombre d'erreurs aléatoires.

Pour tester avec un message plus réaliste, vous pouvez ajuster la fonction d'encodage pour inclure des répétitions de bits de manière plus flexible et simuler des erreurs de transmission de manière aléatoire.

Question 6

Le code de Hamming est capable de détecter et de corriger des erreurs simples (une seule erreur par bloc de données). Pour un message reçu de 7 bits conditionné par un code de Hamming (7,4), nous devons :

1. **Vérifier les bits de parité pour détecter la position de l'erreur.**
2. **Corriger l'erreur si elle existe.**
3. **Extraire les bits de données originaux.**

Implémentation en Python

Voici une fonction pour corriger un message de 7 bits conditionné par un code de Hamming et retrouver le message originel :

Fonction pour corriger le code de Hamming et retrouver le message original

```
def hamming_correct_and_decode(v):
    # Convertir le message en une liste de bits pour un accès plus facile
    bits = [int(bit) for bit in v]

    # Calculer les bits de parité
    p1 = bits[0] ^ bits[2] ^ bits[4] ^ bits[6]
    p2 = bits[1] ^ bits[2] ^ bits[5] ^ bits[6]
```

```
    p3 = bits[3] ^ bits[4] ^ bits[5] ^ bits[6]

    # Calculer la position de l'erreur (si p1, p2, p3 sont tous 0, il n'y a pas d'erreur)
    error_position = p1 + (p2 << 1) + (p3 << 2)

    # Si error_position est non nul, il y a une erreur à corriger
    if error_position != 0:
        bits[error_position - 1] ^= 1  # Corriger l'erreur en inversant le bit

    # Extraire les bits de données (positions 2, 4, 5, et 6 dans la liste corrigée)
    u = f"{bits[2]}{bits[4]}{bits[5]}{bits[6]}"

    return u

# Exemple d'utilisation
v = "0110011"  # Message reçu de 7 bits codé par Hamming (7,4)

# Retrouver le message original
u = hamming_correct_and_decode(v)
print(f"Message reçu : {v}")
print(f"Message original corrigé : {u}")
```

Explications

1. **Calcul des bits de parité** :
    - `p1`, `p2` et `p3` sont recalculés à partir des bits du message reçu pour vérifier les bits de parité.
2. **Détection de la position de l'erreur** :
    - La position de l'erreur est déterminée par la combinaison des bits de parité (`p1`, `p2`, `p3`).
    - Si `error_position` est différent de 0, une erreur est détectée à la position `error_position`.
3. **Correction de l'erreur** :
    - Si une erreur est détectée, le bit à la position `error_position - 1` (en indexation zéro) est inversé pour corriger l'erreur.
4. **Extraction des bits de données** :
    - Les bits de données originaux sont extraits des positions 2, 4, 5 et 6 du message corrigé (en indexation zéro : 2, 4, 5, 6).

Question 7 :

Pour traiter un message de taille quelconque dans lequel au plus 1 bit est altéré tous les 7 bits, nous pouvons appliquer le code de Hamming (7,4) à chaque bloc de 7 bits du message. Cela implique de diviser le message en blocs de 7 bits, corriger chaque bloc individuellement, puis recombiner les blocs corrigés pour retrouver le message original.

Voici comment nous pouvons implémenter cela en Python :

Implémentation en Python

Fonction pour corriger un bloc de 7 bits avec le code de Hamming

```
def hamming_correct_and_decode(block):
    # Convertir le bloc en une liste de bits pour un accès plus facile
    bits = [int(bit) for bit in block]

    # Calculer les bits de parité
    p1 = bits[0] ^ bits[2] ^ bits[4] ^ bits[6]
    p2 = bits[1] ^ bits[2] ^ bits[5] ^ bits[6]
    p3 = bits[3] ^ bits[4] ^ bits[5] ^ bits[6]

    # Calculer la position de l'erreur (si p1, p2, p3 sont tous 0, il n'y a pas d'erreur)
    error_position = p1 + (p2 << 1) + (p3 << 2)

    # Si error_position est non nul, il y a une erreur à corriger
    if error_position != 0:
        bits[error_position - 1] ^= 1  # Corriger l'erreur en inversant le bit

    # Extraire les bits de données (positions 2, 4, 5, et 6 dans la liste corrigée)
    data_bits = f"{bits[2]}{bits[4]}{bits[5]}{bits[6]}"

    return data_bits

#### Fonction pour traiter un message de taille quelconque

```python
def hamming_decode_message(v):
    # Diviser le message en blocs de 7 bits
    blocks = [v[i:i + 7] for i in range(0, len(v), 7)]

    # Corriger chaque bloc et récupérer les bits de données
    corrected_message = ''
    for block in blocks:
        corrected_message += hamming_correct_and_decode(block)

    return corrected_message

# Exemple d'utilisation
v = "01100110110101101001111011010110100010110110011O"  # Message reçu avec des blocs de 7 bits

# Retrouver le message original
u = hamming_decode_message(v)
print(f"Message reçu : {v}")
```
```

```
print(f"Message original corrigé : {u}")
```

Explications

1. **Fonction `hamming_correct_and_decode` :**
    - Corrige un bloc de 7 bits en utilisant le code de Hamming (7,4).
    - Calcule les bits de parité pour détecter et corriger une erreur.
    - Extrait les bits de données originaux.
2. **Fonction `hamming_decode_message` :**
    - Divise le message reçu en blocs de 7 bits.
    - Applique la correction à chaque bloc et récupère les bits de données.
    - Combine les bits de données corrigés pour reformer le message original.