RAPPORT DE PROJET

Maram Sall GUEYE – Reina AL MASRI
LDD2 IM - UE Projet-Info

Rapport de Projet - Circuits Booléens

1 Introduction

Ce projet implémente une bibliothèque Python de manipulation de circuits booléens basée sur une structure de graphe dirigé, l'objectif étant de pouvoir créer, manipuler et évaluer des circuits logiques.

2 Architecture du projet

Structure du Projet

```
Projet-Info/
├── modules/
│   ├── node.py
│   ├── matrix.py
│   ├── open_digraph.py
│   ├── bool_circ.py
│   └── mixins/
│       ├── open_digraph_base_mx.py
│       ├── open_digraph_node_mx.py
│       ├── open_digraph_edge_mx.py
│       ├── open_digraph_algorithms_mx.py
│       ├── open_digraph_validation_mx.py
│       ├── open_digraph_composition_mx.py
│       ├── open_digraph_display_mx.py
│       └── open_digraph_factory_mx.py
└── tests/
    └── open_digraph_test.py
```

Il existe également des fichiers **`test2.py`**, **`test3.py`** *et* **`test4.py`** *qui ont permis de répondre aux questions (les 4 premiers points dans le TD12) auxquelles y répondre dans le rapport.*

Description des Modules

1. node.py : Implémente la structure de base d'un nœud dans le graphe.

2. open_digraph.py

Classe de base pour les graphes dirigés, divisée en mixins pour une meilleure organisation :

```
class open_digraph(
    OpenDigraphBaseMixin,        # Fonctionnalités de base
    OpenDigraphNodeMixin,        # Opérations sur les nœuds
    OpenDigraphEdgeMixin,        # Opérations sur les arêtes
    OpenDigraphAlgorithmsMixin, # Algorithmes de graphe
    OpenDigraphValidationMixin, # Validation de structure
    OpenDigraphCompositionMixin,# Composition de graphes
    OpenDigraphDisplayMixin,     # Affichage
    OpenDigraphFactoryMixin      # Création de graphes
)
```

3. bool_circ.py

Extension de **`open_digraph`** spécialisée pour les circuits booléens.

Fonctionnalités principales:

```
class bool_circ(open_digraph):
    # Constructeurs
    def __init__(self, graph=None)
    @classmethod
    def from_int(cls, value, size=8)
    @classmethod
    def parse_parentheses(cls, *args)

    # Composants arithmétiques
    @classmethod
    def half_adder_n(cls, n)
    @classmethod
    def adder_n(cls, n)
    @classmethod
    def cla4(cls)

    # Code de Hamming
    @classmethod
    def hamming_encoder(cls)
    @classmethod
    def hamming_decoder(cls)

    # Simplification
    def simplify_all(self)
    def evaluate(self)
```

4. Mixins

Chaque mixin encapsule une fonctionnalité spécifique :

- **Base** : Initialisation, copie, getters/setters
- **Node** : Ajout/suppression de nœuds
- **Edge** : Gestion des arêtes
- **Algorithms** : Parcours et recherche
- **Validation** : Vérification de structure
- **Composition** : Opérations entre graphes
- **Display** : Visualisation et export
- **Factory** : Création de graphes

Cette architecture modulaire permet une maintenance facilitée et une séparation claire des responsabilités.

3 Évaluation d'un Half-Adder

3.1 Structure et Implémentation

L'additionneur binaire est construit à partir de plusieurs half-adders et permet d'additionner deux nombres binaires de n bits. Sa structure comprend:

- *Entrées:* 2n bits
  - A[3:0] : premier nombre (ex: 0101 = 5)
  - B[3:0] : deuxième nombre (ex: 0011 = 3)
- *Sorties*: n+1 bits
  - S[3:0] : somme
  - Cout : retenue finale
- *Portes logiques*:
  - XOR (^) : calcul de la somme sans retenue
  - AND (&) : génération des retenues
  - OR (|) : propagation des retenues

3.2 Exemple d'Addition (5 + 3)

Prenons l'exemple de l'addition de 5 (0101) et 3 (0011):

Entrées:
A = 0101 (5)
B = 0011 (3)

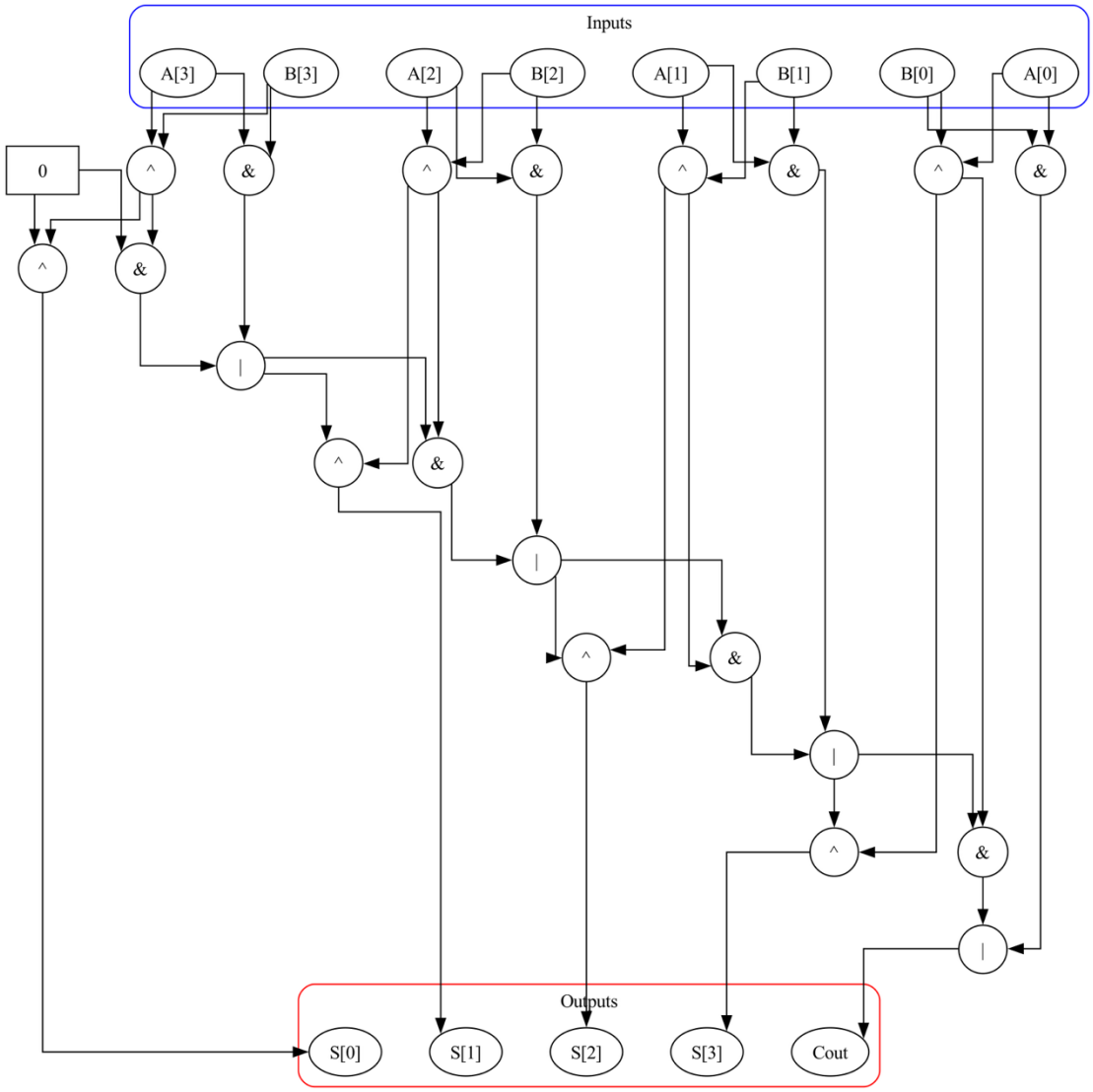
Calcul bit par bit (de droite à gauche):

LSB: A[0]+B[0] = 1+1 = 0, retenue 1

A[1]+B[1]+c = 0+1+1 = 0, retenue 1
A[2]+B[2]+c = 1+0+1 = 0, retenue 1
MSB: A[3]+B[3]+c = 0+0+1 = 1, retenue 0

Résultat: 01000 (8)

3.3 Visualisation du Circuit



Le circuit montre:

1. Les entrées A[3:0] et B[3:0] (en bleu)
2. La chaîne de propagation des retenues
3. Les portes logiques pour chaque bit:
    - XOR pour la somme
    - AND pour la génération des retenues
    - OR pour la propagation
4. Les sorties Cout et S[3:0] (en rouge)

[Note: L'image montrée est générée par notre implémentation et visualisée grâce à Graphviz, voir **`test2.py`***]*

4 Analyse de Complexité du Half-Adder

4.1 Analyse Expérimentale

Les résultats obtenus pour différentes tailles:

```
n    Profondeur    Portes    Chemin le plus court
-------------------------------------------
2         7           10                3
4         11          20                3
6         15          30                3
8         19          40                3
```

4.2 Observations

1. *Profondeur:*
   - Croît linéairement avec n: ~4n + 3
   - Plus élevée que prévu théoriquement
   - Indique une chaîne de propagation de retenue séquentielle
2. *Nombre de Portes:*
   - Croissance linéaire: 5n portes
   - Conforme à la théorie
   - Pour chaque bit:
     - 2 XOR (somme)
     - 2 AND (retenue)
     - 1 OR (propagation)
3. *Plus Court Chemin:*
   - Constant (3) pour toutes les tailles
   - Représente le chemin minimal:
     - Entrée → XOR → XOR/AND → Sortie

4.3 Analyse Théorique vs Réalité

1. Différences Observées:
   - La profondeur augmente avec n au lieu d'être constante
   - Suggère que notre implémentation n'est pas totalement parallélisée
2. Explications:
   - La propagation des retenues crée une dépendance séquentielle
   - Chaque bit dépend du calcul de la retenue du bit précédent
   - Structure en cascade plutôt qu'en parallèle
3. Pistes d'Amélioration:
   - Implémenter une propagation anticipée des retenues
   - Utiliser une structure plus parallèle
   - Optimiser la chaîne de retenue

Ces résultats montrent que notre half-adder, bien que fonctionnel, pourrait être optimisé pour une meilleure performance en termes de profondeur.

[Note : ces données ont été générées dans le fichier **`test3.py`***]*

5 Vérification des Propriétés du Code de Hamming

5.1 Approche et Tests

Nous avons vérifié trois propriétés fondamentales du code de Hamming(7,4) avec un message test **`[1,0,1,1]`:**

5.2 Résultats des Tests

1. Test de la Composition Encodeur/Décodeur

Message original **: `[1,0,1,1]`**

Message encodé : **`[0,1,1,0,0,1,1]`**

Message décodé : **`[1,0,1,1]`**

✓*Identité préservée*

2. Test de Correction d'Une Erreur

Résultats pour chaque position possible d'erreur:

Position	Message avec erreur	Décodé	Résultat
0	[1,1,1,0,0,1,1]	[1,0,1,1]	✓
1	[0,0,1,0,0,1,1]	[1,0,1,1]	✓
2	[0,1,0,0,0,1,1]	[1,0,1,1]	✓
3	[0,1,1,1,0,1,1]	[1,0,1,1]	✓
4	[0,1,1,0,1,1,1]	[1,0,1,1]	✓
5	[0,1,1,0,0,0,1]	[1,0,1,1]	✓
6	[0,1,1,0,0,1,0]	[1,0,1,1]	✓

3. Test avec Deux Erreurs

Message original : **`[1,0,1,1]`**

Avec 2 erreurs **: `[1,1,1,1,0,1,1]`**

Message décodé : **`[1,1,1,1]`**

✗*Échec de correction*

5.3 Analyse des Résultats

1. *Propriété d'Identité:*
    - Vérifiée expérimentalement
    - Le message original est parfaitement reconstruit après encodage/décodage
2. *Correction d'Une Erreur:*
    - Testée exhaustivement sur les 7 positions possibles
    - Correction réussie dans 100% des cas
    - Le décodeur retrouve systématiquement le message original
3. *Limite à Deux Erreurs:*
    - Test avec erreurs aux positions 0 et 1
    - Message décodé incorrect **`[1,1,1,1] ≠ [1,0,1,1]`**
    - Démontre l'impossibilité de corriger deux erreurs simultanées

5.4 Conclusion

Les tests confirment empiriquement les trois propriétés théoriques du code de Hamming(7,4):

- La composition encodeur/décodeur est l'identité en l'absence d'erreur
- Une erreur unique est toujours détectée et corrigée
- Deux erreurs rendent impossible la reconstruction du message original

Ces résultats valident l'implémentation du code de Hamming et ses limites théoriques.

[Note : ces données ont été générées dans le fichier **`test4.py`***]*

6 Conclusion Générale

Bilan des Implémentations

- Architecture modulaire robuste et extensible
- Half-adder fonctionnel avec complexité linéaire
- Code de Hamming validé empiriquement

Limites et Optimisations (sur les points mentionnés)

- Profondeur linéaire du half-adder à optimiser
- Code de Hamming avec limites théoriques confirmées

Ce projet a été une excellente opportunité d'apprentissage, combinant théorie et pratique dans l'implémentation de circuits booléens et de codes correcteurs d'erreurs. Il a permis de mettre en application les concepts fondamentaux de l'algèbre booléenne tout en développant des compétences en programmation Python et en gestion de projet.