

Bibliothèque Python pour la modélisation de graphes dirigés et la synthèse de circuits booléens

Yehor Korotenko

Sipan Bareyan

Ivan Kharkov

9 mai 2025

Résumé

Les digraphes ouverts, graphes orientés avec des noeuds d'entrée et de sortie spécifiés, et les circuits booléens, graphes acycliques orientés représentant des opérations logiques, sont fondamentaux dans des domaines tels que l'analyse de réseaux, la vérification formelle et l'IA. Ce projet présente une nouvelle bibliothèque Python conçue pour pallier ces limitations. La bibliothèque fournit des fonctionnalités pour la construction, la représentation et la manipulation de digraphes ouverts et de circuits booléens, ainsi qu'une documentation riche. Les principales caractéristiques comprennent : une représentation efficace des graphes et des circuits, des algorithmes pour le calcul de la profondeur et la simplification des circuits, des opérations booléennes et des capacités de visualisation. La bibliothèque a été rigoureusement testée à l'aide d'une suite de tests complète. Cet ensemble d'outils convient à une variété d'applications, notamment l'analyse de la sécurité des réseaux et les études de théorie des graphes. Ce projet a fourni une expérience précieuse en algorithmes de graphes, en ingénierie logicielle et en pratiques de développement collaboratif.

1 Objectifs du projet

Ce projet vise à développer une bibliothèque Python composée de deux modules complémentaires :

- `open_digraph` : définition, manipulation et analyse modulaires de graphes dirigés,
- `bool_circ` : surcouche spécialisée pour la création, l'optimisation et l'évaluation de circuits booléens acycliques.

Une suite de tests automatisés (tests unitaires) couvre tous les modules (`node`, `open_digraph`, `bool_circ`, mixins, parseur, additionneur...) pour assurer robustesse et non-régression.

2 Architecture modulaire de la bibliothèque OpenDigraph & BoolCirc

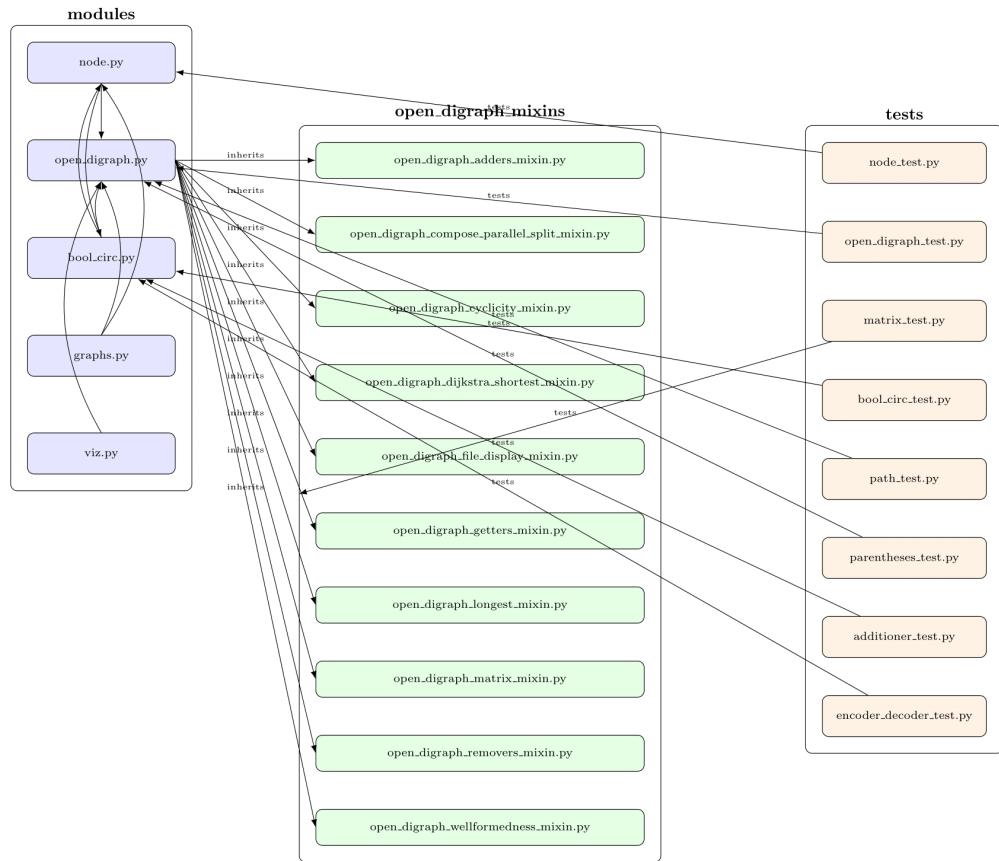


FIGURE 1 – La structure du projet

Module `open_digraph`.

- Structure de base : noeuds et arêtes dirigées.
- Système de *mixins* (CRUD, parcours, calcul de distances) pour enrichissement progressif.

Extension `bool_circ`.

- Spécialisation en circuits booléens acycliques avec portes prédéfinies (AND, OR, NOT, XOR...).
- Moteur de simplification : associativité, éradication de constantes, fusion de portes.
- Évaluateur de signaux binaires.

3 Quelques mots sur le processus de développement du projet

Le développement de notre projet s'est effectué de manière structurée et progressive. Nous avons débuté en créant différents types de graphes aléatoires, notamment des graphes orientés sans cycles, afin d'établir une base solide avant d'aborder la complexité supérieure des circuits booléens.

Lors du passage aux circuits booléens, nous avons dû résoudre des défis techniques spécifiques. Parmi ces défis figuraient la gestion des noeuds sans parents ou sans étiquettes, ainsi

que les connexions complexes entre les nœuds. Pour améliorer la modularité et faciliter l'organisation du code, nous avons adopté les mixins, ce qui nous a permis une réutilisation efficace du code.

Par la suite, nous avons implémenté une classe spécifique nommée `bool_circ`, qui étend notre classe existante `open_digraph`. Cette classe est conçue pour gérer précisément les particularités des circuits booléens, telles que les portes logiques et les entrées/sorties. Nous avons également mis en place des méthodes permettant de sauvegarder et visualiser ces circuits via le format `dot`, simplifiant ainsi considérablement le processus de débogage et d'analyse structurelle.

De plus, nous avons développé des algorithmes capables de générer automatiquement des circuits à partir de formules logiques propositionnelles. Tout au long du projet, nous avons ajouté de nombreux tests pour garantir la fiabilité et la robustesse de notre code. Ces tests continus ont permis de détecter rapidement les erreurs et d'assurer un développement stable. Nous avons exploré la composition des circuits en parallèle et en séquence, ainsi que l'analyse des chemins d'information pour optimiser leur fonctionnement.

Nous avons intégré des méthodes d'évaluation des circuits par l'application de règles de réécriture et développé des techniques pour valider leur exactitude, en particulier dans le cas de circuits spécifiques comme ceux du code de Hamming.

Enfin, l'utilisation de *Git* pour le contrôle des versions dès les premières étapes a été essentielle pour assurer une collaboration efficace et un suivi précis des différentes évolutions de notre projet.

4 Analyse de l'additionneur

Cette section présente une analyse théorique de la profondeur et du nombre de portes logiques dans les circuits additionneurs, construits à partir de cellules adders de 1 bit.

4.1 1-bit Full Adder

Une cellule d'additionneur complet de 1 bit prend trois entrées : a_i , b_i (les bits à additionner) et c_i (le carry-in). Elle produit deux sorties : s_i (le bit de somme) et c_{i+1} (le carry-out). Les expressions booléennes régissantes sont les suivantes :

$$\begin{aligned}s_i &= (a_i \hat{\wedge} b_i) \hat{\wedge} c_i \\c_{i+1} &= (a_i \& b_i) | ((a_i \hat{\wedge} b_i) \& c_i)\end{aligned}$$

Ici, $\hat{\wedge}$ indique XOR, $\&$ indique AND, et $|$ indique OR.

4.1.1 Profondeur logique (niveaux de porte)

La profondeur logique est le nombre maximal de portes logiques connectées en série entre une entrée primaire (supposée au niveau 0) et une sortie.

- **Sum** s_i : Le calcul implique deux opérations XOR séquentielles : $x_i = a_i \hat{\wedge} b_i$ (1er niveau de porte), suivi de $s_i = x_i \hat{\wedge} c_i$ (2ème niveau de porte). La profondeur logique de s_i est de **2 niveaux de porte**.
- **Carry-out** c_{i+1} : Le chemin vers c_{i+1} implique :
 1. $p_i = a_i \& b_i$ (1er niveau de porte, AND).
 2. $x_i = a_i \hat{\wedge} b_i$ (1er niveau de porte, XOR, parallel a p_i).
 3. $q_i = x_i \& c_i$ (2ème niveau de la porte, AND, depend de x_i).

4. $c_{i+1} = p_i | q_i$ (3ème niveau de la porte, OR, dépend de p_i et q_i).

La profondeur logique de c_{i+1} est de **3 niveaux de porte**.

La profondeur logique globale de la cellule de l'additionneur complet de 1 bit est déterminée par le chemin le plus long, qui va jusqu'à c_{i+1} , soit **3 niveaux de porte**.

4.1.2 Nombre de portes logiques

En supposant que les sous-expressions soient recalculées si elles sont utilisées dans des fonctions de sortie distinctes (par exemple, si les expressions de somme et de report sont analysées indépendamment, comme avec un utilitaire `parse_parentheses`) :

- For $s_i = (a_i \wedge b_i) \wedge c_i$: 2 XOR gates.
- For $c_{i+1} = (a_i \& b_i) | ((a_i \wedge b_i) \& c_i)$: 1 XOR (pour $a_i \wedge b_i$), 2 portes ET et 1 porte OU (4 portes au total pour la logique de report).

Le nombre total de portes logiques est de $2 + 4 = 6$.

4.2 Additionneur simple de M bits

Un additionneur simple de M bits est formé par la mise en cascade de M cellules de l'additionneur complet de 1 bit. La sortie c_{i+1} de l'étage i devient l'entrée de l'étage $i + 1$. La retenue initiale, c_0 , est généralement égale à '0'.

4.2.1 Profondeur logique (niveaux de porte)

Le chemin critique qui détermine la profondeur est la propagation de la retenue du LSB(Least significant bit) au MSB(Most significant bit).

- Stage 0 (LSB) : Produces c_1 at a depth of 3 gate levels.
- Subsequent stages $i > 0$: The logic to produce c_{i+1} from c_i adds 2 gate levels to the path of c_i . Thus, $\text{depth}(c_{i+1}) = \text{depth}(c_i) + 2$.
- Tracing carry depths : $\text{depth}(c_1) = 3$; $\text{depth}(c_k) = 3 + (k - 1) \times 2 = 2k + 1$.
- The final carry-out c_M has $\text{depth}(c_M) = 2M + 1$.
- The MSB sum s_{M-1} depends on c_{M-1} (depth $2(M - 1) + 1$) and requires 2 more XOR levels, resulting in $\text{depth}(s_{M-1}) = (2(M - 1) + 1) + 2 = 2M + 1$.

La profondeur logique d'un additionneur de M bits est $D(M) = 2M + 1$.

Par exemple :

1. $M = 1 \Rightarrow \text{Profondeur} = 3$;
2. $M = 2 \Rightarrow \text{Profondeur} = 5$;
3. $M = 4 \Rightarrow \text{Profondeur} = 9$.

4.2.2 Chemin le plus court (niveaux de porte)

Le chemin logique le plus court est généralement celui de la somme LSB $s_0 = (a_0 \wedge b_0) \wedge c_0$. Cela implique **2 niveaux de porte** (deux XOR).

4.3 Influence de la duplication des signaux sur la profondeur du graphe structurel

Dans les représentations graphiques des circuits, les signaux (sorties de portes ou entrées primaires) servent souvent d'entrées à plusieurs portes ultérieures (split-out). Pour gérer cette situation, les structures des graphes comprennent des "nœuds de copie" (nœuds avec une étiquette vide ""), qui dupliquent ou traversent les signaux.

L'introduction de ces nœuds de copie ajoute des couches structurelles au graphe. Alors que la profondeur logique ne prend en compte que la cascade de portes opérationnelles (AND, OR, XOR), la profondeur structurelle mesurée sur le graphe peut être plus importante. Chaque cas où un signal (par exemple, a_i, b_i, c_i) est acheminé vers plusieurs portes opérationnelles distinctes peut nécessiter une couche intermédiaire de nœuds de copie. Par conséquent, la profondeur structurelle globale du graphe, reflétant toutes ces couches opérationnelles et d'acheminement des signaux, peut dépasser la profondeur de la porte purement opérationnelle. Par exemple, pour un additionneur complet de 1 bit, la profondeur augmente de 2.

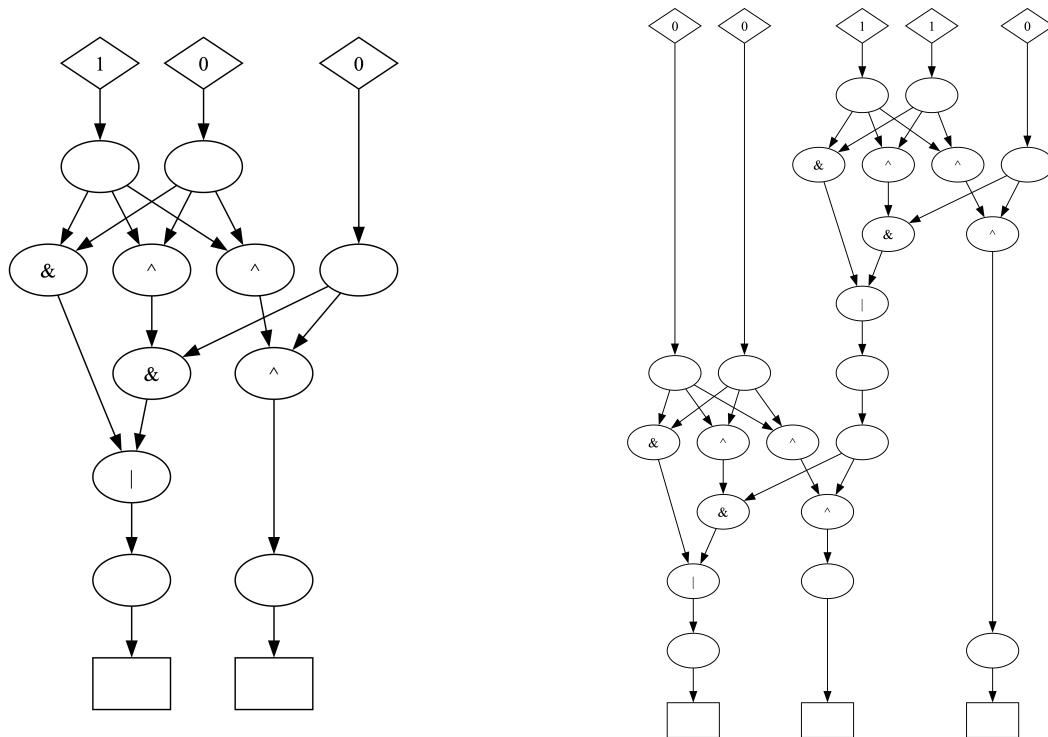


FIGURE 2 – Exemples des adders des registres 2^0 et 2^1 réspéctivement

5 Carry-Lookahead Adders

L'additionneur de retenue (CLA) réduit la profondeur de calcul en pré-calculant les signaux de "propagation" (P_i) et de "génération" (G_i) pour chaque bit i : $P_i = a_i \wedge b_i$ et $G_i = a_i \& z b_i$. Cela permet de déterminer les reports de manière plus directe. L'implémentation discutée utilise des blocs CLA de 4 bits, qui sont ensuite enchaînés pour des additionneurs plus grands de $4N$ bits.

5.1 Analyse du bloc Carry-Lookahead 4 bits

Un bloc de 4 bits traite les entrées a_i, b_i (pour $i \in [0, 3]$) et un *carry-in* de bloc C_{in_0} , produisant les sommes S_i et un *carry-out* de bloc C_{out_3} (également C_4).

5.1.1 Génération des signaux P et G (bloc de 4 bits)

- **Nombre de portes** : 4 XORs (for $P_0..P_3$) + 4 ANDs (for $G_0..G_3$) = **8 portes**.
- **Profondeur logique** : Tous les P_i, G_i sont calculés en parallèle : **1 niveau de porte**.

5.1.2 Calcul de retenue (itératif dans un bloc de 4 bits)

On utilise $C_{out,i} = G_i | (P_i \& C_{in,i})$, où $C_{in,i+1} = C_{out,i}$.

- **Nombre de portes** : Pour 4 étapes ($C_{out,0}$ à $C_{out,3}$) : $4 \times (1 \text{ AND} + 1 \text{ OR}) = **8 portes**$.
- **Profondeur logique** :

P_i, G_i sont à la profondeur 1. $C_{in,0}$ est à la profondeur 0.

1. $depth(C_{out,0}) = 3$
2. $depth(C_{out,1}) = 5$
3. $depth(C_{out,2}) = 7$
4. $depth(C_{out,3}) = 9$.

Le report de bloc $C_{out,3}$ est à **9 niveaux de porte**.

5.1.3 Calcul de la somme (bloc de 4 bits))

La somme des bits est calculée à l'aide de la formule $S_i = P_i \hat{\wedge} C_i$. La profondeur de chaque S_i dépend des temps d'arrivée (profondeurs) de P_i et C_i . Nous rappelons que tous les signaux P_i sont disponibles à la profondeur 1, et que les porteuses C_i (entrées de l'étage i) sont disponibles à des profondeurs variables : C_0 à la profondeur 0, C_1 à la profondeur 3, C_2 à la profondeur 5, et C_3 à la profondeur 7. La porte XOR pour S_i ajoute un niveau à la profondeur maximale de ses entrées.

- Pour $S_0 = P_0 \hat{\wedge} C_0$: $depth(S_0) = \max(depth(P_0), depth(C_0)) + 1 = \max(1, 0) + 1 = **2**$.
- Pour $S_1 = P_1 \hat{\wedge} C_1$: $depth(S_1) = \max(depth(P_1), depth(C_1)) + 1 = \max(1, 3) + 1 = **4**$.
- Pour $S_2 = P_2 \hat{\wedge} C_2$: $depth(S_2) = \max(depth(P_2), depth(C_2)) + 1 = \max(1, 5) + 1 = **6**$.
- Pour $S_3 = P_3 \hat{\wedge} C_3$: $depth(S_3) = \max(depth(P_3), depth(C_3)) + 1 = \max(1, 7) + 1 = **8**$.

La profondeur logique maximale parmi les sorties de la somme est pour S_3 , qui est de **8 niveaux de porte**.

5.1.4 Totaux pour un bloc CLA de 4 bits

- **Nombre total de portes logiques** : 8 (P/G) + 8 (Carry) + 4 (Sums) = **20 portes**.
- **Profondeur logique globale** : La profondeur déterminée est de **9 niveaux de porte** car pour la retenue, nous avons besoin de 9 niveaux de porte.

5.1.5 Chemin le plus court (niveaux de porte)

Il reste S_0 dans le premier bloc (LSB) : **2 niveaux de porte**.

5.2 Influence de la duplication des signaux sur la profondeur du graphe structurel (CLA)

La construction directe de blocs CLA de 4 bits implique souvent des "nœuds de copie" comme dans l'additionneur simple. Ces nœuds non opérationnels augmentent la profondeur structurelle du graphe. Par exemple, un seul bloc CLA de 4 bits avec une profondeur logique de 9 peut présenter une profondeur structurelle de graphe plus élevée (15 lorsque l'on calcule la profondeur en utilisant le tri topologique du graphe).

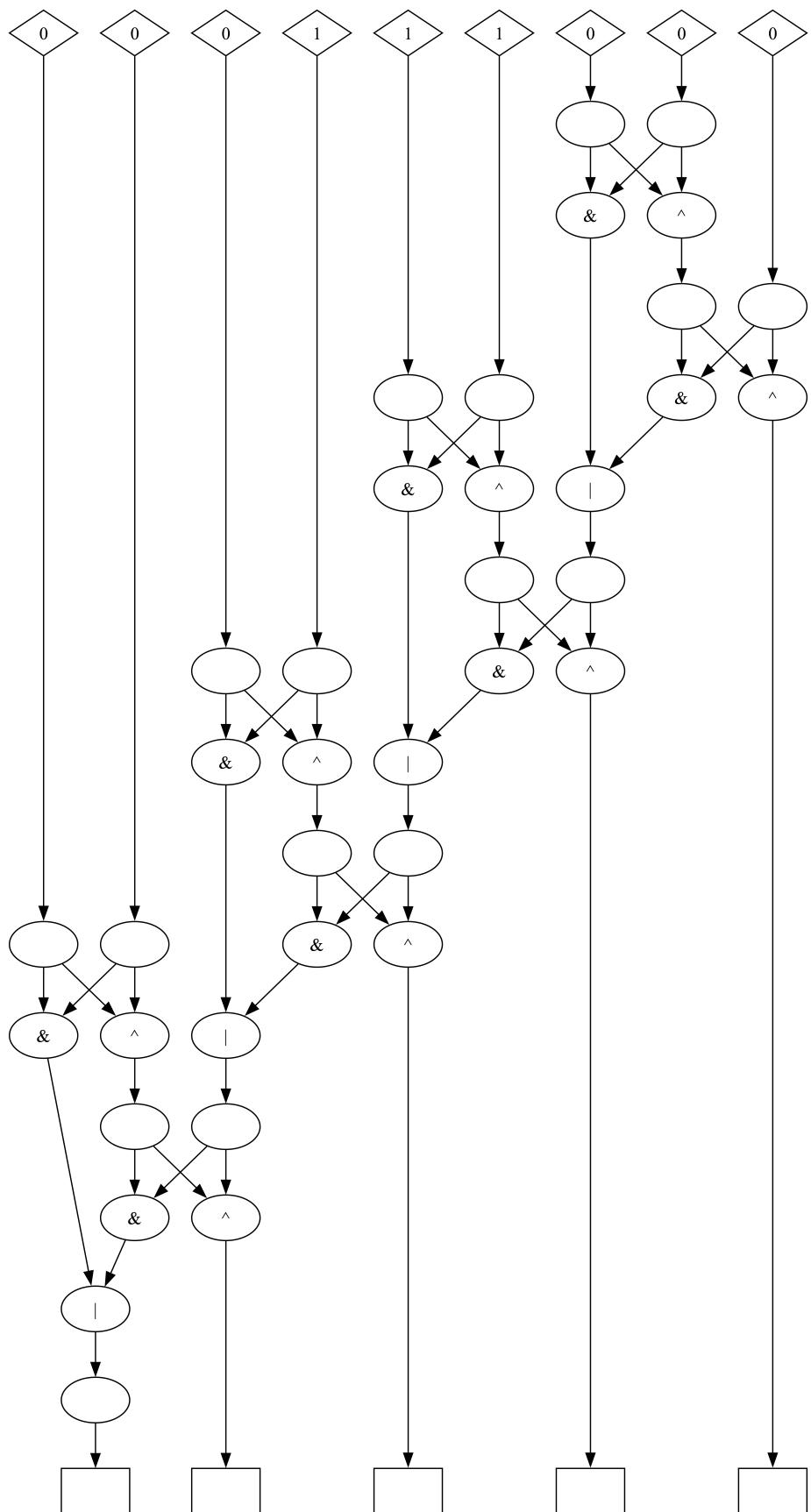


FIGURE 3 – Exemple d'un carry look ahead

6 Évaluation d'un demi-additionneur (Half Adder)

Nous illustrons le mécanisme interne du module `bool_circ` en évaluant un demi-additionneur appliqué à deux entiers codés sur 4 bits (par exemple 2 et 3).

6.1 Création du graphe initial

1. **Génération des circuits booléens.** Chaque entier est converti en un sous-graphe de largeur 4, chaque bit devenant un nœud d'entrée.
2. **Construction du half adder.** Pour chaque position de bit, les deux nœuds d'entrée se connectent à un sous-graphe implémentant une porte `XOR` (somme partielle) et une porte `AND` (retenue).
3. Assemblage en un graphe complet du demi-additionneur.

6.2 Évaluation et simplification itérative du graphe

1. **Initialisation.** Graphe complet issu de l'assemblage.
2. **Passes de simplification.** À chaque passe, pour chaque nœud, on teste et applique localement la première règle de simplification disponible. On répète jusqu'au point fixe (aucune règle applicable).

6.3 Règles de simplification

Lors de chaque passe de l'algorithme `evaluate()`, on parcourt tous les noeuds du graphe et on applique localement l'une des règles ci-dessous dès qu'elle est applicable. On répète les passes tant qu'au moins une règle peut encore s'appliquer.

6.3.1 Propagation et élimination de constantes

Copie de constante (constant_copy_transform) Si une porte de copie ("") reçoit en entrée une constante (0 ou 1), on la remplace par autant de constantes fraîches qu'il y a de sorties, puis on supprime la porte de copie et éventuellement la constante d'origine si elle devient isolée.

Inversion de constante (constant_not_transform) Si une porte NOT (\sim) a pour unique parent une constante, on la remplace par la constante inversée directement connectée à la sortie, et on supprime la porte NOT (et éventuellement la constante source si elle devient isolée).

6.3.2 Simplifications logiques basées sur AND/OR/XOR

AND-0 (transform_and_zero) Si un AND ($\&$) a un parent 0, le résultat est 0 : on reconnecte 0 à la sortie, on remplace chaque autre entrée par sa propre porte de copie, puis on supprime l'AND.

AND-1 (transform_and_one) Si un AND a un parent 1, cette entrée est neutre : on supprime simplement tous les fils issus de ce 1 vers l'AND, puis on supprime le 1 isolé.

OR-0 (transform_or_zero) Si un OR ($|$) a un parent 0, cette entrée est neutre : on supprime tous les fils de ce 0 vers l'OR, puis on supprime le 0 isolé.

OR-1 (transform_or_one) Si un OR a un parent 1, le résultat est 1 : on reconnecte ce 1 à la sortie, on remplace chaque autre entrée par sa propre copie, puis on supprime l'OR.

XOR-0 (`transform_xor_zero`) Dans un XOR (\wedge), un 0 en entrée est neutre : on supprime simplement ses fils vers le XOR, et on supprime le 0 isolé.

XOR-1 (`transform_xor_one`) Un 1 dans un XOR équivaut à une inversion : on enlève l'arête $1 \rightarrow$ XOR, on place un NOT à la sortie du XOR et on supprime le 1 isolé.

Porte sans entrée — OR/XOR vide \rightarrow 0 (`transform_in_zero`)

— AND vide \rightarrow 1 (`transform_in_one`)

Si une porte logique n'a plus d'entrées, on la remplace par la constante neutre appropriée, reconnectée à ses enfants.

6.3.3 Règles d'associativité et d'involution

Associativité du XOR (`transform_associative_xor`) Deux XOR consécutifs sont fusionnés en un seul XOR dont les parents sont la réunion de leurs anciennes entrées.

Fusion de copies (`transform_associative_copy`) Deux portes de copie successives sont fusionnées en une seule.

Involution du XOR (`transform_involution_xor`) Si deux copies identiques se reroutent vers un même XOR avec une multiplicité paire, on supprime les deux (effet neutre), sinon on laisse une connexion unique.

Double négation (`transform_not_involution`) Deux NOT en série ($\sim\sim$) s'annulent : on raccorde directement l'entrée à la sortie et on supprime les deux portes NOT.

6.3.4 Gestion des portes mortes et déplacement de NOT

Effacement d'opérateur isolé (`transform_erase_operator`) Une porte de copie sans sortie (mais avec un parent) est supprimée, et on reconnecte ses anciens entrées directement à de nouvelles copies si nécessaire.

Sortie du NOT hors du XOR (`transform_xor_if_has_parent_not`) Si un XOR a un parent NOT, on déplace ce NOT en aval du XOR (équivalence de $\sim(a \oplus b) = a \oplus b$ puis NOT).

Sortie du NOT hors de la copie (`transform_copy_if_has_parent_not`) Si une copie a un parent NOT, on récupère le signal original avant NOT, on fuse et on refait les NOT nécessaires en aval.

6.3.5 Répétition jusqu'au point fixe

- On recommence autant de passes que nécessaire, tant qu'au moins une règle a été appliquée lors de la passe précédente.
- Le processus se termine lorsque plus aucune simplification n'est possible (atteinte d'un point fixe).

6.4 Résultat final

Le graphe maximalement réduit fournit les valeurs de Somme (S) et Retenue (C). Pour notre exemple $(2 + 3)$, 62 passes d'évaluation ont été nécessaires.

La visualisation d'évaluation de *half-adder* est disponible à la fin du rapport (voir les figures 4, 5, 6, 7, 8)

7 Vérification de la propriété principale du code de Hamming

Pour valider que l'encodeur et le décodeur se composent en l'identité (correction d'une erreur simple), nous suivons :

7.1 Approche générale

Pour valider que l'encodeur et le décodeur se composent en l'identité (et que l'on corrige toute erreur simple, mais pas toute double erreur), nous suivons quatre grandes étapes :

1. Génération des graphes

- **Encodeur 4 bits** : à partir d'un mot de 4 bits, on construit le graphe d'encodage. Chaque bit de donnée est relié à des portes de parité (XOR) qui produisent les trois bits de contrôle.
- **Décodeur 4 bits** : on élabore un graphe de décodage composé de modules de syndrome (XOR et AND) et d'un réseau de correction (portes conditionnelles) capable de détecter et corriger une seule erreur.

2. Composition des graphes

- On relie les sorties de l'encodeur aux entrées du décodeur pour obtenir un seul graphe « encodeur→décodeur ».
- Cette étape est automatique : la méthode de composition s'appuie sur les listes d'identifiants de noeuds de sortie et d'entrée pour tisser les connexions appropriées.

3. Simplification (évaluation)

- Appel de la méthode `evaluate()` sur le graphe composé :
 - Chaque passe balaie tous les noeuds à la recherche de motifs simplifiables (fusion de portes identiques, élimination de constantes, propagation de NOT).
 - Dès qu'une transformation s'applique, le graphe est modifié et une nouvelle passe recommence.
 - Le processus s'arrête au point fixe, c'est-à-dire lorsqu'aucune règle n'est plus applicable.

4. Extraction et comparaison des résultats

- On lit les quatre bits de sortie restants et on forme un mot binaire.
- On compare ce mot au mot d'entrée d'origine, pour chacun des scénarios de test.

7.2 Cas de test

Sans erreur

- Composition de l'encodeur et du décodeur sur le mot initial (voir les figures de l'encodeur : 9, le décodeur : 10, et la composition des deux derniers : 11)
- Après évaluation, le graphe final se réduit à quatre fils directs (identité). (voir la figure : 12)
- Résultat : mot de sortie = mot d'entrée.

Une erreur unique

- Injection d'une inversion sur l'un des quatre bits en sortie de l'encodeur. (voir les figures du décodeur : 13 et de la composition : 14)
- Le décodeur calcule un syndrome, active la porte de correction correspondante, puis le graphe se simplifie en identité. (voir la figure : 12)
- Résultat : même mot restauré, quelle que soit la position de l'erreur.

Deux erreurs

- Inversions sur deux positions distinctes. (voir le décodeur : figure 15)
- Le graphe final, simplifié au maximum, reste différent de l'identité, et le mot en sortie ne correspond plus à l'entrée. (voir l'identité avec erreur : figure 16)

Exemples des tests en python

```
1 def test_enc_dec_compose_eval_gives_identity(self):
2     # define the sequence of bits to encode
3     bit1, bit2, bit3, bit4 = '0', '1', '0', '1'
4     res = bit1+bit2+bit3+bit4
5
6     # construct encoder
7     enc_g = bool_circ.generate_4bit_encoder(bit1, bit2, bit3, bit4)
8     # construct decoder
9     dec_g = bool_circ.generate_4bit_decoder(' ', ' ', ' ', ' ', ' ', ' ', ' ')
10    # compose both
11    comp = dec_g.compose(enc_g)
12    comp = bool_circ(comp)
13    # evaluate to get the result
14    comp.evaluate()
15
16    # verify that we obtain an identity graph
17    self.assertEqual(len(comp.get_nodes_ids()), 4*2)
18
19    # compare the evaluated result
20    self.assertEqual(get_result_of_evaluated_enc_dec(comp), res)
21
22 def test_enc_dec_compose_eval_gives_identity_with_changed_bit_1(self):
23     bit1, bit2, bit3, bit4 = '0', '1', '0', '1'
24     res = bit1+bit2+bit3+bit4
25
26     enc_g = bool_circ.generate_4bit_encoder(bit1, bit2, bit3, bit4)
27     # ajoute d'une porte non
28     dec_g = bool_circ.generate_4bit_decoder('~', ' ', ' ', ' ', ' ', ' ', ' ')
29     comp = dec_g.compose(enc_g)
30     comp = bool_circ(comp)
31     comp.evaluate()
32
33     self.assertEqual(len(comp.get_nodes_ids()), 4*2)
34
```

```

35     self.assertEqual(get_result_of_evaluated_enc_dec(comp), res)
36
37 def test_enc_dec_compose_eval_gives_identity_with_changed_bit_2(self):
38     bit1, bit2, bit3, bit4 = '0', '1', '0', '1'
39     res = bit1+bit2+bit3+bit4
40
41     enc_g = bool_circ.generate_4bit_encoder(bit1, bit2, bit3, bit4)
42     # ajoute d'une porte non pour l'autre bit
43     dec_g = bool_circ.generate_4bit_decoder(' ', ' ', ' ', ' ', ' ', ' ', ' ')
44     comp = dec_g.compose(enc_g)
45     comp = bool_circ(comp)
46     comp.evaluate()
47
48     self.assertEqual(len(comp.get_nodes_ids()), 4*2)
49
50     self.assertEqual(get_result_of_evaluated_enc_dec(comp), res)
51
52 def test_enc_dec_compose_eval_with_2bits_changed(self):
53     bit1, bit2, bit3, bit4 = '0', '1', '0', '1'
54     res = bit1+bit2+bit3+bit4
55
56     enc_g = bool_circ.generate_4bit_encoder(bit1, bit2, bit3, bit4)
57     # ajoute de deux portes non
58     dec_g = bool_circ.generate_4bit_decoder(' ', ' ', ' ', ' ', ' ', ' ', ' ')
59     comp = dec_g.compose(enc_g)
60     comp = bool_circ(comp)
61     comp.evaluate()
62
63     self.assertEqual(len(comp.get_nodes_ids()), 4*2)
64
65     self.assertNotEqual(get_result_of_evaluated_enc_dec(comp), res)

```

Listing 1 – tests

8 Visualisation des étapes intermédiaires

8.1 Les figures de l'additionneur

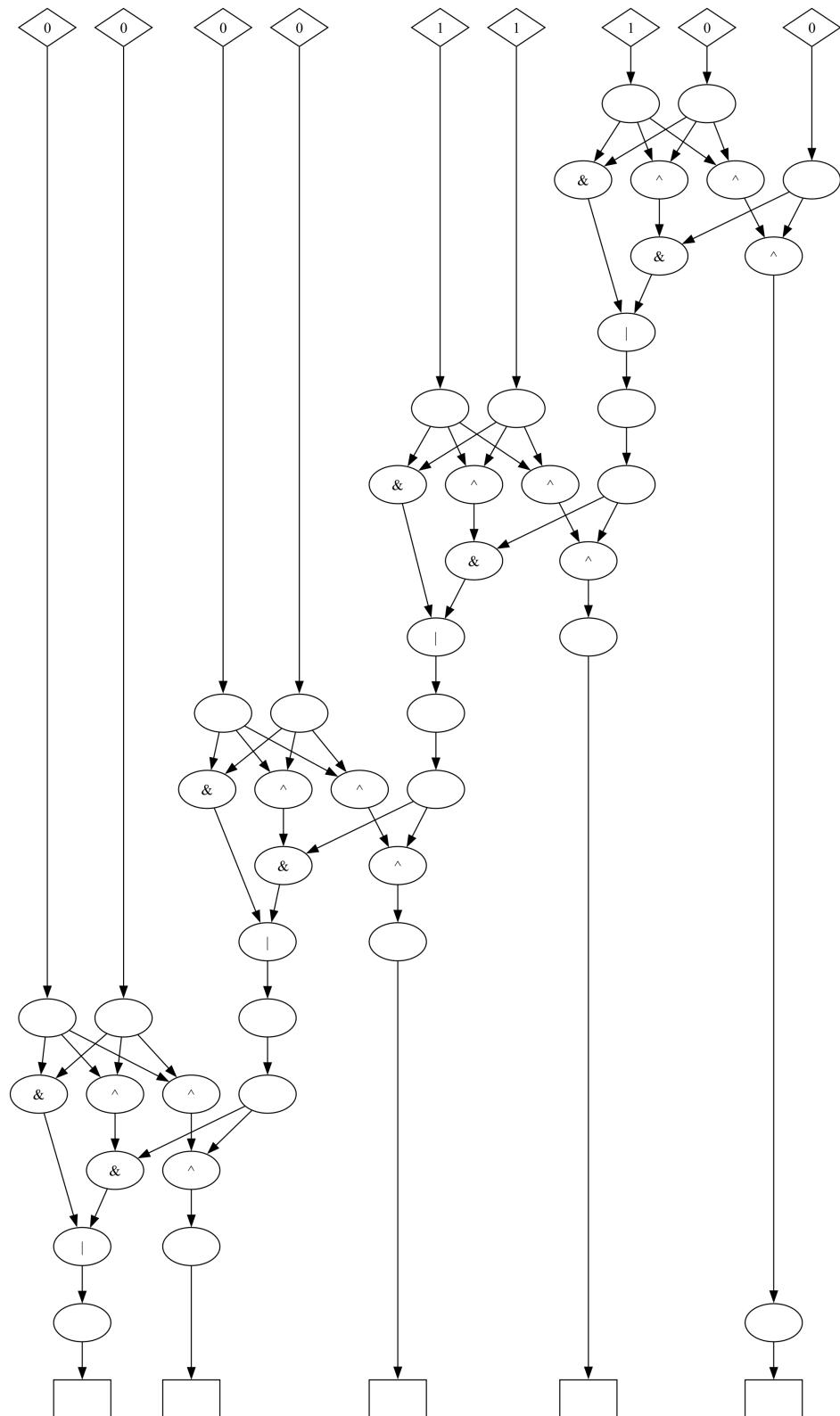


FIGURE 4 – Half-adder juste après l'initialisation

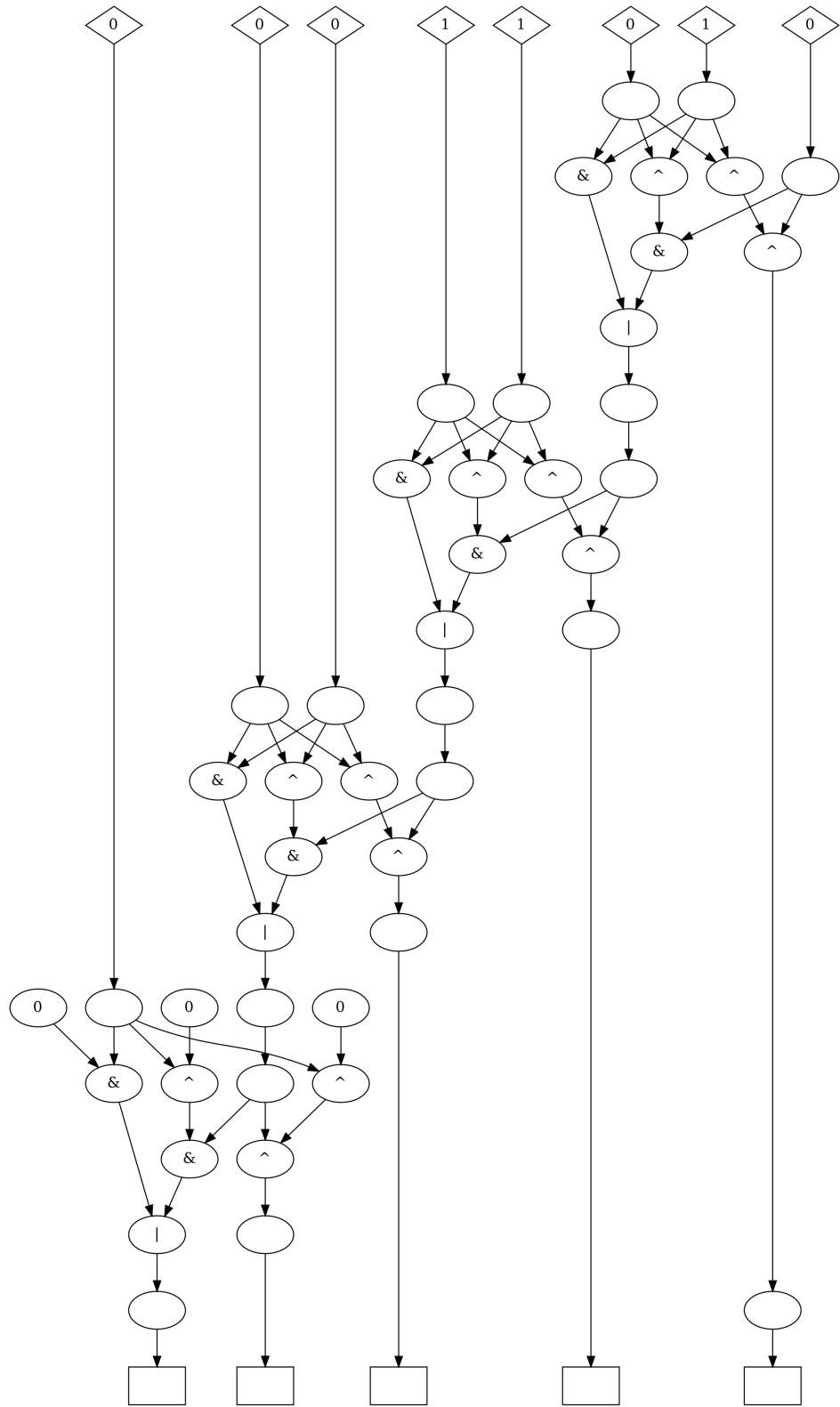


FIGURE 5 – Half-adder après le premier pas d'évaluation

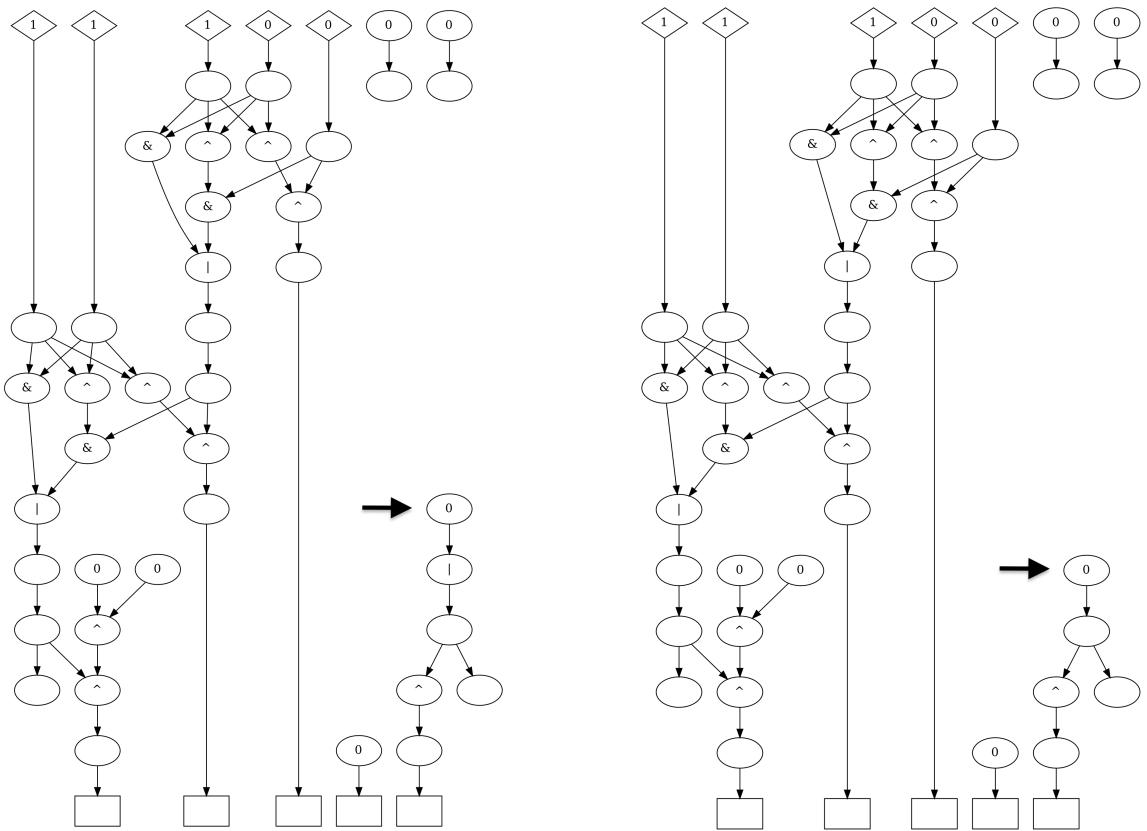


FIGURE 6 – Comparaison de l'état de half-adder avant et apres d'une application d'une transformation intermédiaire ($0|0 = 0$)

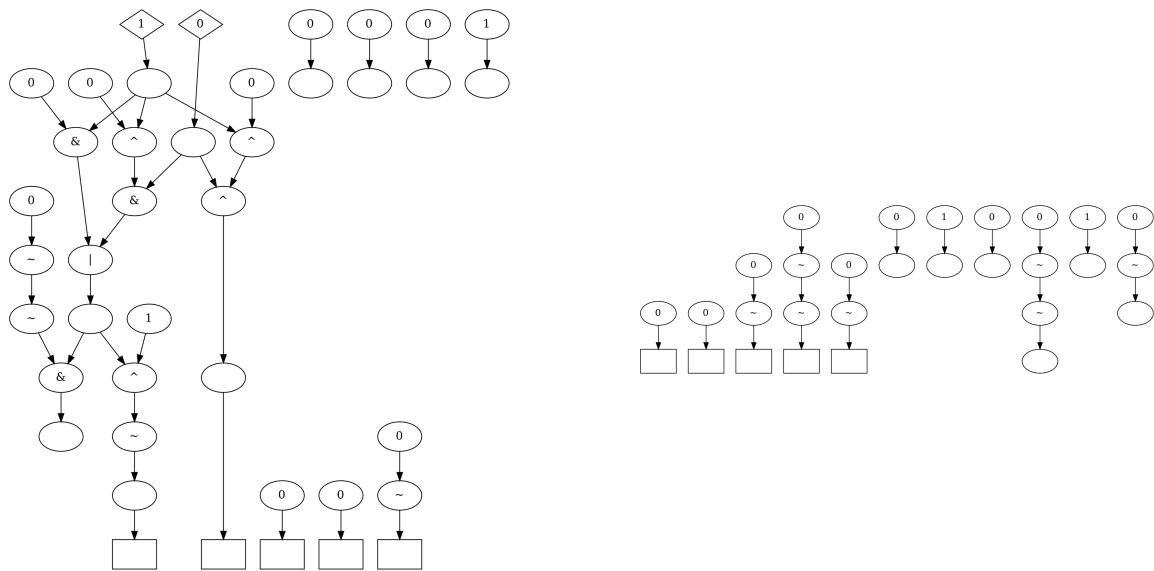


FIGURE 7 – Les étapes avant la finalisation de l'évaluation de half-adder

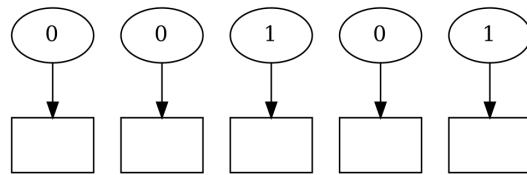


FIGURE 8 – L'état de *half-adder* après l'évaluation complète

8.2 Les figures de l'encodeur et décodeur

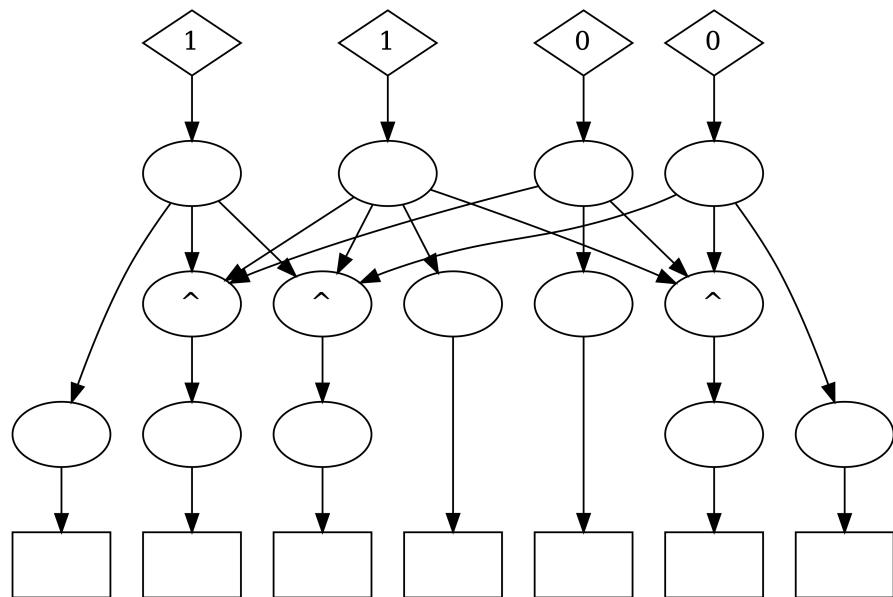


FIGURE 9 – L'encodeur

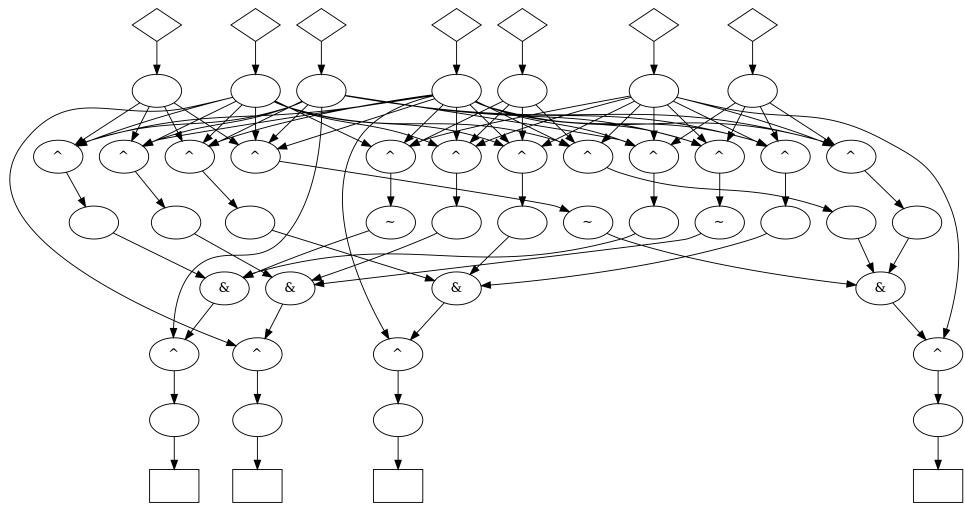


FIGURE 10 – Le décodeur

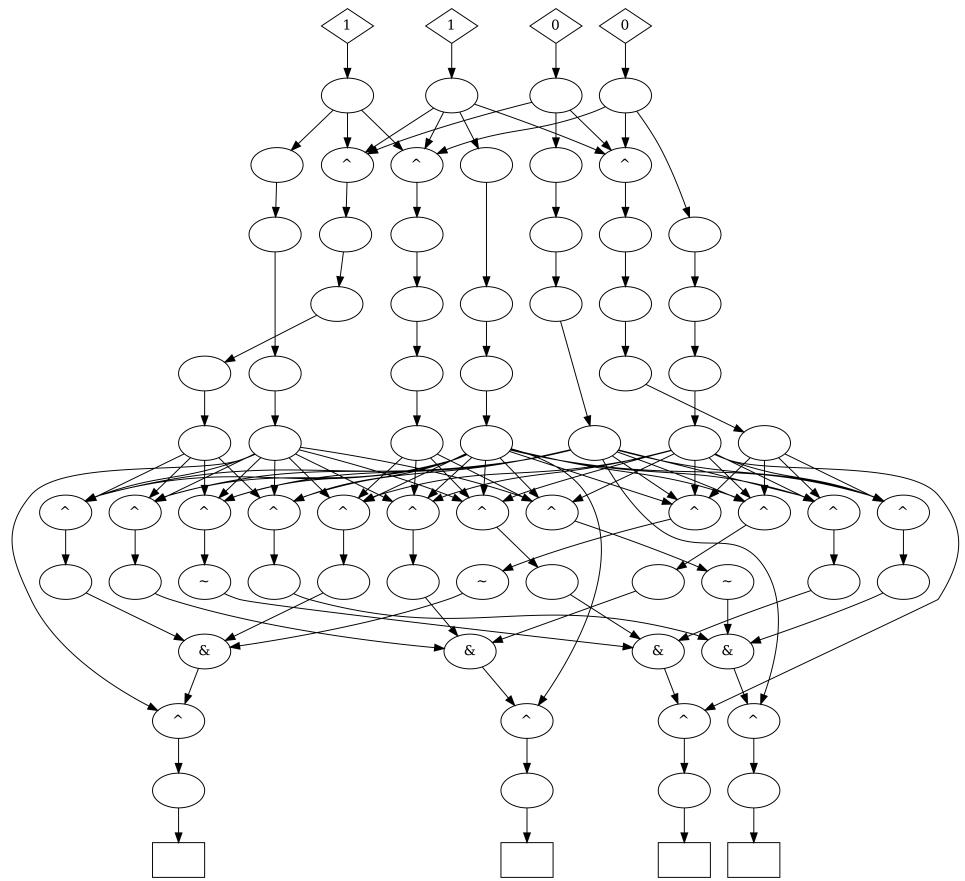


FIGURE 11 – La composition de l'encodeur et décodeur

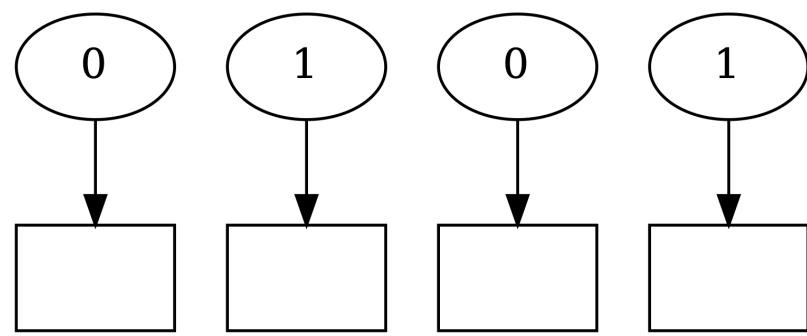


FIGURE 12 – L'identité après l'évaluation de la composition de l'encodeur et le décodeur

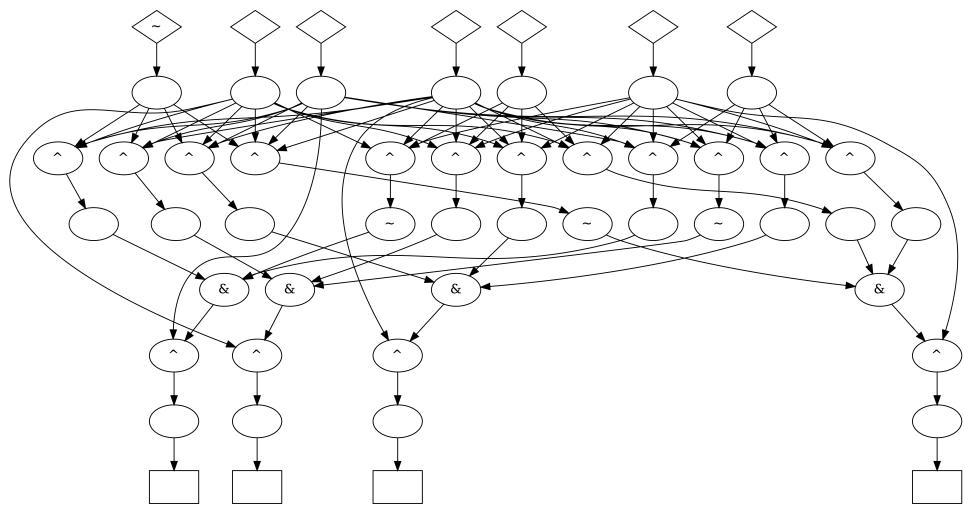


FIGURE 13 – Le décodeur avec une erreur (une porte non)

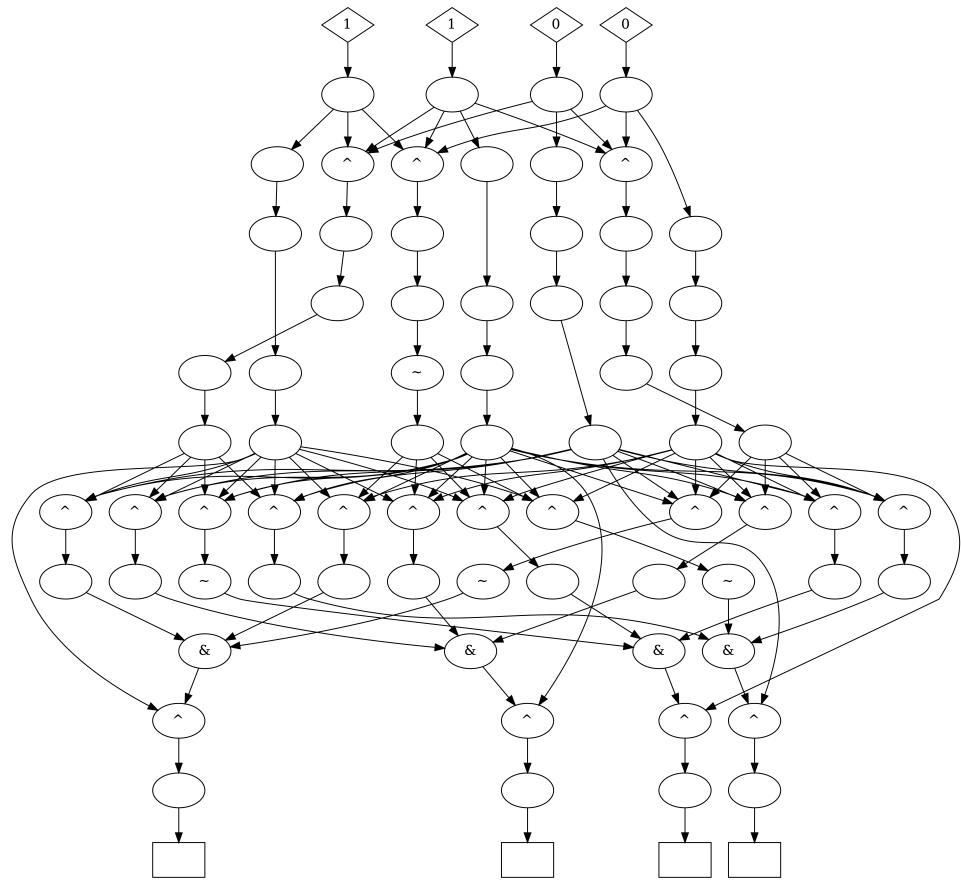


FIGURE 14 – La composition de l'encodeur et du décodeur avec une erreur (une porte non)

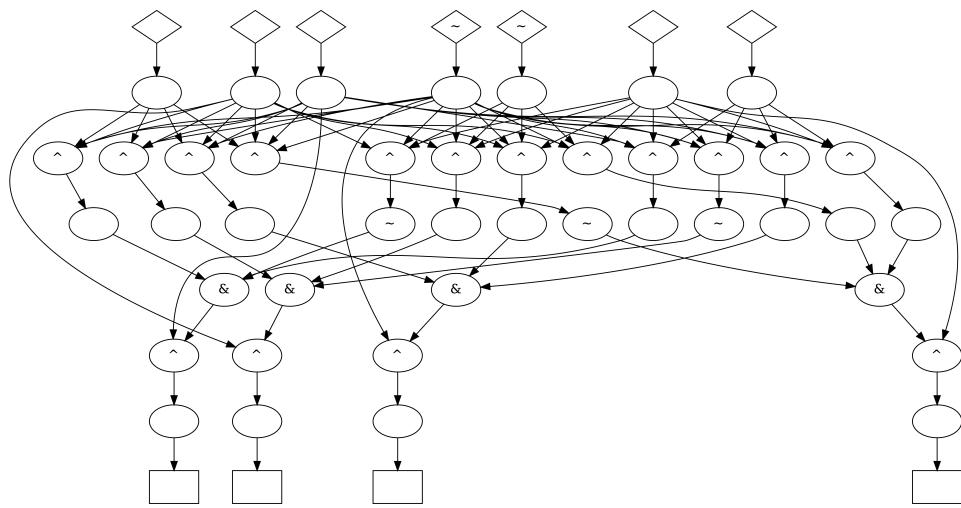


FIGURE 15 – Le décodeur avec deux erreurs (voir deux portes non dans les inputs)

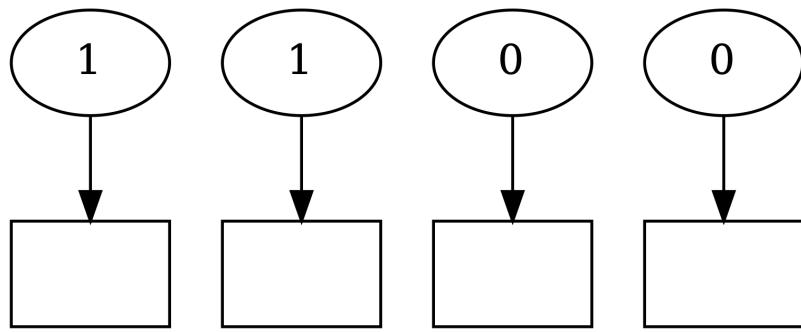


FIGURE 16 – L’identité avec les bits changés qui ne correspondent pas aux bits initiales

Table des figures

1	La structure du projet	2
2	Exemples des adders des registres 2^0 et 2^1 réspéctivement	5
3	Exemple d’un carry look ahead	8
4	Half-adder juste après l’initialisation	14
5	Half-adder après le premier pas d’evaluate	15
6	Comparaison de l’état de half-adder avant et apres d’une application d’une transformation intermédiaire ($0 0 = 0$)	16
7	Les étapes avant la finalisation de l’évaluation de half-adder	16
8	L’état de <i>half-adder</i> après l’évaluation complète	17
9	L’encodeur	17
10	Le décodeur	18
11	La composition de l’encodeur et décodeur	18
12	L’identité après l’évaluation de la composition de l’encodeur et le décodeur . . .	19
13	Le décodeur avec une erreur (une porte non)	19
14	La composition de l’encodeur et du décodeur avec une erreur (une porte non) . .	20
15	Le décodeur avec deux erreurs (voir deux portes non dans les inputs)	20
16	L’identité avec les bits changés qui ne correspondent pas aux bits initiales . . .	21