

BARREIRO Hugo  
RIBEIRO Damien



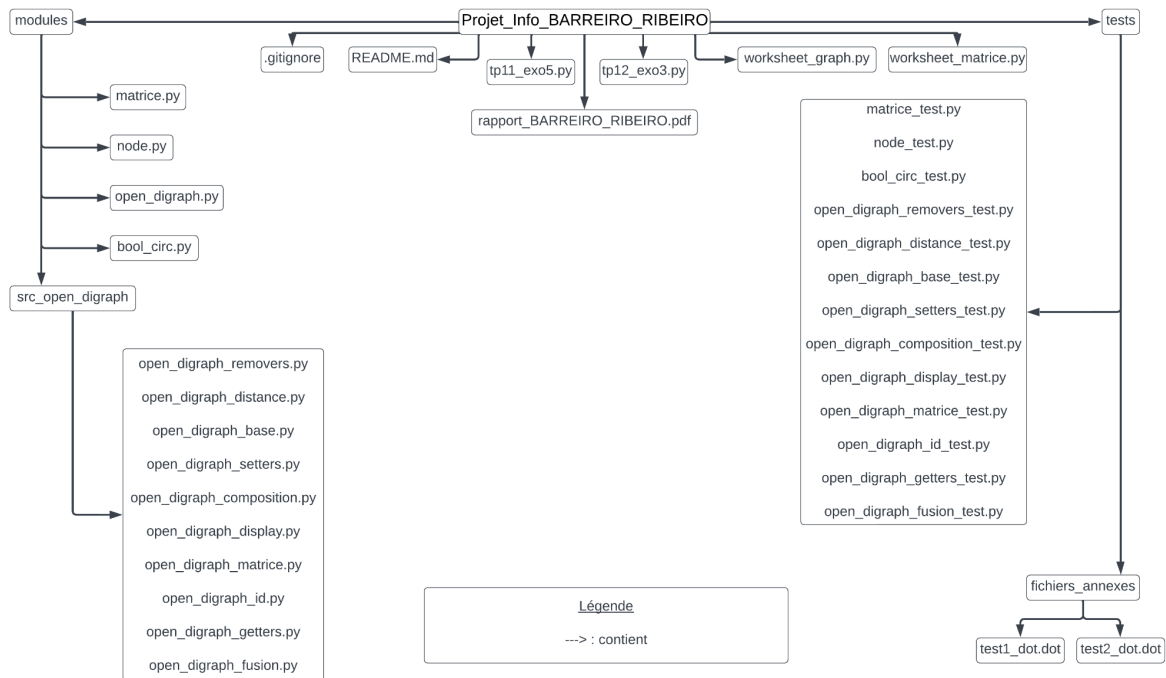
## Projet Informatique - Rendu Final

## Sommaire :

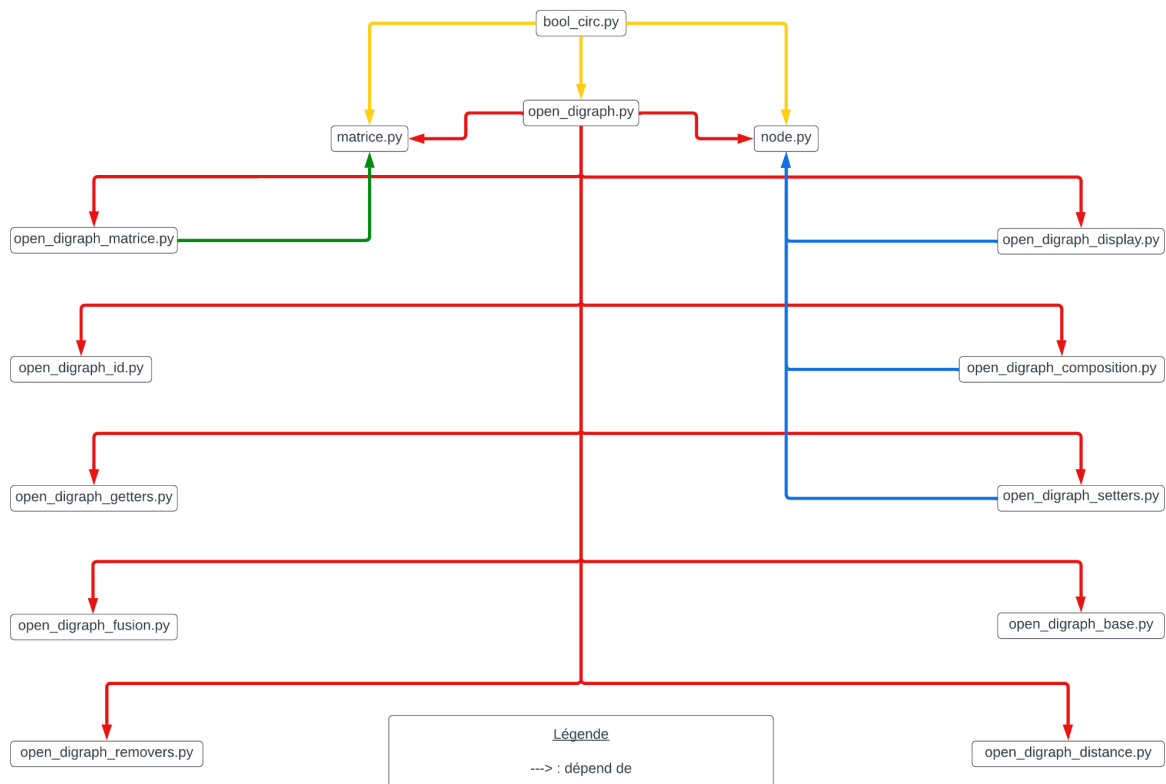
I) Schéma de l'architecture du projet	page 3
II) Evaluation d'un Half-Adder sur deux entiers	page 4
III) Partie théorique sur le Half-Adder	page 10
IV) Vérification de la propriété principale du code de Hamming	page 14

## I) Schéma de l'architecture du projet

### A) Arborescence du projet :



### B) Dépendances du projet :



## II) Evaluation d'un Half-Adder sur deux entiers

Pour tester et réaliser cette évaluation nous avons créé un fichier à part (tp11\_exo5.py à la racine du projet).

Voici son contenu :

```
from modules.bool_circ import *

# On verifie qu'une addition fonctionne bien grace au half-adder

print()
print(
    "On verifie qu'on obtient le bon resultat de l'addition avec un half-adder"
)
print()

ha = bool_circ.Half_Adder(3)

for i in range(0, 16):
    for j in range(0, 32):

        reg1 = bool_circ.registre(i * i, size=8)
        reg2 = bool_circ.registre(4 * j + 1, size=8)

        reg = reg1.parallel(reg2)
        comp = ha.compose(reg)

        comp.evaluate()

# On test si on trouve la bonne reponse

comp_nodes = comp.get_nodes()
comp_outputs = comp.get_output_ids()

assert len(comp_nodes) == 18
assert len(comp_outputs) == 9

s = ""

for k in comp_outputs:
```

```

        s += comp[comp[k].get_parents_ids()[0]].get_label()

    res = i * i + 4 * j + 1
    b = str(bin(res)[2:])

    if len(b) < 9:
        b = "0" * (8 - len(b)) + b

    if res >= 2**8:
        b = "1" + b

    else:
        b = "0" + b

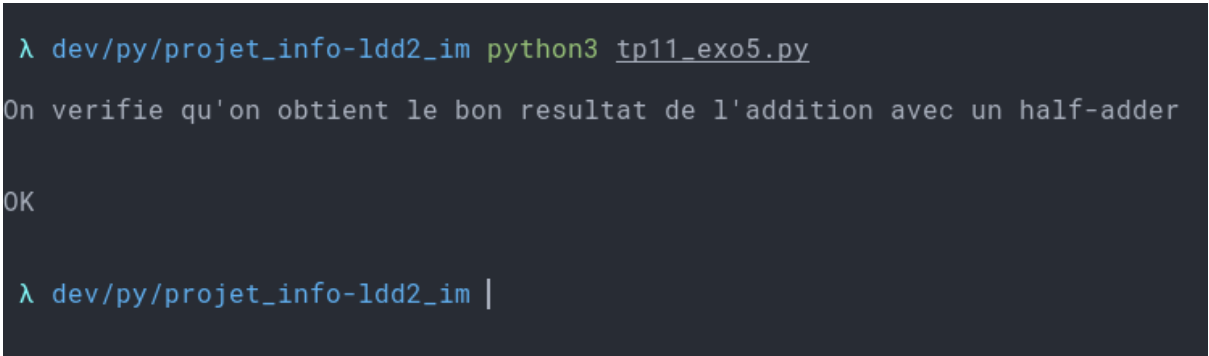
    assert s == b

print()
print("OK")
print()

```

Le but de ce fichier est de vérifier qu’une addition de deux registres (représentant deux nombres entiers) se déroule bien grâce au Half-Adder et donne le bon résultat.

Lorsqu’on exécute le fichier dans le terminal, un message indiquant que le début du test s’affiche, puis soit il s’affiche “OK” si tout s’est bien déroulé, soit il s’affiche une erreur, soulevée par un assert, si quelque chose s’est mal passé.

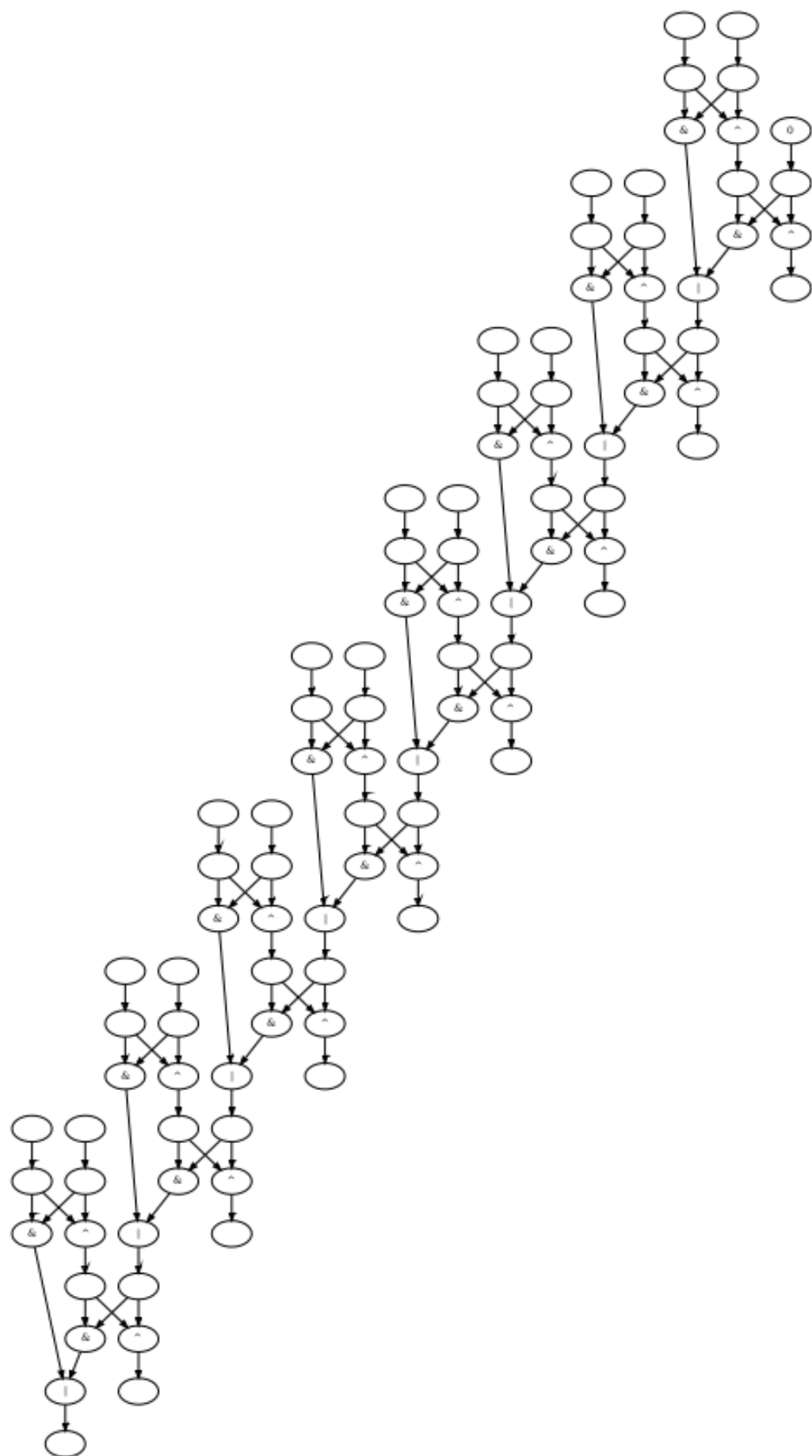


```

λ dev/py/projet_info-ldd2_im python3 tp11_exo5.py
On verifie qu'on obtient le bon resultat de l'addition avec un half-adder
OK
λ dev/py/projet_info-ldd2_im |

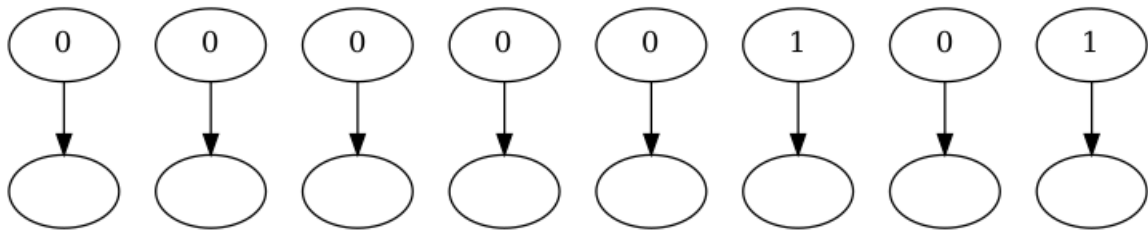
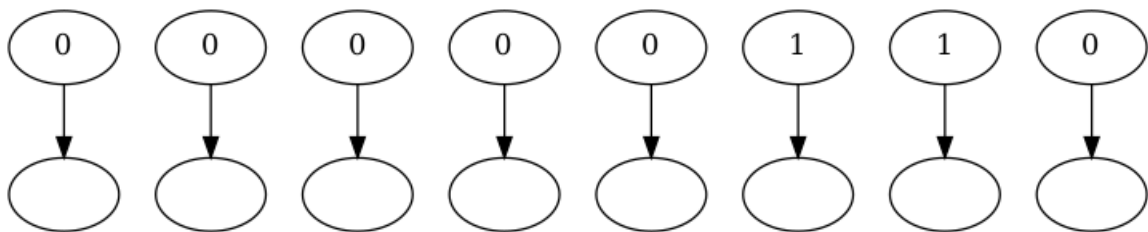
```

Au départ, on va créer un Half-Adder de taille 3. Nous avons besoin d’un Half-Adder de taille 3 car nos registres utilisés pour l’addition sont de taille 8 (pour 8 bits). Or, le Half-Adder a besoin d’être de taille au moins 3 car  $\log_2(8) = 3$ .

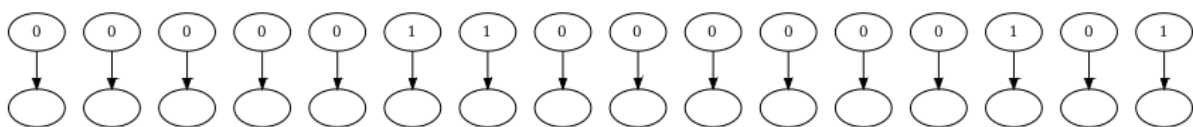


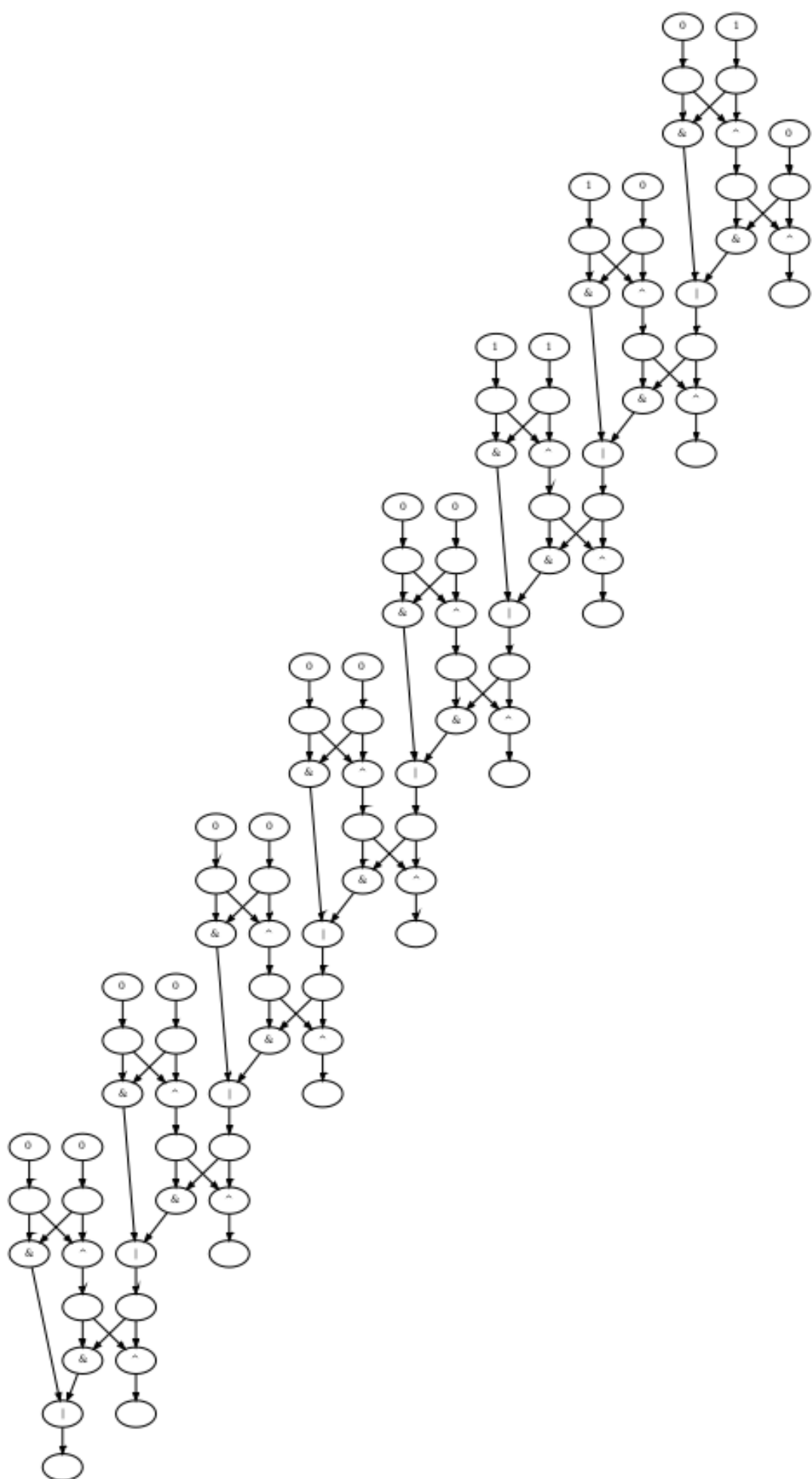
Ensuite, on va créer un tas de registre correspondant à différents nombres codés sur 8 bits.

Par exemple, on peut créer le registre correspondant au nombre 6 sur 8 bits, puis le registre correspond au nombre 5 sur 8 bits.



On compose parallèlement 2 registres pour les donner en input du Half-Adder. (ce qui correspond à la composition entre du Half-Adder par le “grand registre”)







Enfin, on évalue cette composition. On espère trouver le registre correspondant au nombre 11 (car  $6 + 5 = 11$ ).

Avant de vérifier qu'on trouve bien le nombre 11, on va vérifier que le graph obtenu soit bien cohérent.

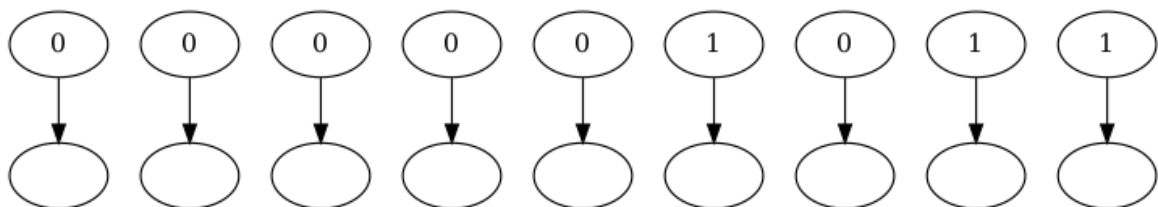
Pour cela, on regarde s'il comporte 9 outputs (les 8 bits + 1 bit pour la retenue). Puis on regarde s'il comporte 18 nœuds (les 9 outputs + les 9 bits du résultat de l'addition).

Si cela est bien respecté, on peut alors vérifier que les 9 nœuds réponse correspondent bien au nombre 11 codé sur 9 bits.

Si tous tests sont passés sans erreurs alors on affiche "OK" ce qui signifie que le résultat obtenu est celui souhaité et est correct.

Sinon, une erreur aurait été soulevée. (on a également utilisé print et display lors de la programmation de tout cela pour vérifier si tout était en ordre)

Ici, tout est vérifié, on obtient donc (ce qui correspond bien à 11) :



### III) Partie théorique sur le Half-Adder

#### A) Profondeur et nombre de portes :

La profondeur d'un Half-Adder est donné par la formule :

$$2^{(n+2)} - 2^n + 1$$

Le nombre de porte d'un Half-Adder est donné par la formule :

$$10 * 2^n - 2^n$$

(On ne prend pas en compte les inputs/outputs et la constante 0)

#### B) Plus court chemin entre une entrée et une sortie :

La longueur du plus court chemin entre une entrée et une sortie est toujours égale à 5.

#### C) Vérifications :

Pour vérifier la profondeur d'un Half-Adder, on a écrit le programme suivant :

```
from modules.bool_circ import *

for i in range(9):
    ha = bool_circ.Half_Adder(i)

    prof = ha.graph_depth()
    calc = 2**(i + 2) - 2**i + 1
    reste = calc - prof

    print(i, " : ", prof, " | ", calc, " | ", reste)
```

Ce programme calcule la profondeur de chaque Half-Adder et vérifie si notre formule donne le bon résultat.

On obtient :

```
λ dev/py/projet_info-ldd2_im python3 tmp3.py
0 : 4 | 4 | 0
1 : 7 | 7 | 0
2 : 13 | 13 | 0
3 : 25 | 25 | 0
4 : 49 | 49 | 0
5 : 97 | 97 | 0
6 : 193 | 193 | 0
7 : 385 | 385 | 0
8 : 769 | 769 | 0

λ dev/py/projet_info-ldd2_im
```

Ce qui confirme notre hypothèse.

Pour vérifier le nombre de porte d'un Half-Adder, on a écrit le programme suivant :

```
from modules.bool_circ import *

for i in range(10):
    ha = bool_circ.Half_Adder(i)

    ha_nodes = ha.get_nodes()
    ha_outputs = ha.get_output_ids()
    ha_inputs = ha.get_input_ids()

    nodes = []
    for node in ha_nodes:
        iden = node.get_id()
        if iden not in ha_inputs and iden not in ha_outputs and node.get_label(
        ) != "0":
            nodes.append(node)

    nb = len(nodes)
    calc = 10 * (2**i) - (2**i)
    reste = calc - nb

    print(i, " : ", nb, " | ", calc, " | ", reste)
```

Ce programme calcule le nombre de portes d'un Half-Adder puis vérifie si notre formule donne le bon résultat pour les Half-Adder de taille  $0 \leq n \leq 9$ .

Obtient le résultat suivant :

```
λ dev/py/projet_info-ldd2_im python3 tmp.py
0 : 9 | 9 | 0
1 : 18 | 18 | 0
2 : 36 | 36 | 0
3 : 72 | 72 | 0
4 : 144 | 144 | 0
5 : 288 | 288 | 0
6 : 576 | 576 | 0
7 : 1152 | 1152 | 0
8 : 2304 | 2304 | 0
9 : 4608 | 4608 | 0

λ dev/py/projet_info-ldd2_im
```

On observe qu'on obtient exactement le nombre de portes du Half-Adder.

Pour vérifier que la longueur du plus court chemin entre une entrée et une sortie est toujours égale à 5, on a écrit le programme suivant :

```
from modules.bool_circ import *

for i in range(8):
    ha = bool_circ.Half_Adder(i)
    ha_outputs = ha.get_output_ids()
    ha_inputs = ha.get_input_ids()

    long = len(ha.shortest_path(ha_inputs[0], ha_outputs[0]))

    for inp in ha_inputs:
        for out in ha_outputs:
            cur_long = len(ha.shortest_path(inp, out))

            if cur_long < long:
                long = cur_long

    print(i, " : ", long)
```

Ce programme calcule la longueur du plus court chemin entre chaque entrée et sortie, il garde seulement la longueur minimale pour chaque Half-Adder.

On obtient le résultat suivant :

```
λ dev/py/projet_info-ldd2_im python3 tmp2.py
0 : 5
1 : 5
2 : 5
3 : 5
4 : 5
5 : 5
6 : 5
7 : 5

λ dev/py/projet_info-ldd2_im
```

Ce qui confirme notre hypothèse.

#### D) Discussion-Problème :

Notre programme ne permet pas de créer des adder avec  $n \geq 10$ . Cela est causé par le fait que lorsque l'on construit un `bool_circ`, on appelle une fonction appelée `is_well_formed()` qui appelle la méthode `is_cyclic`, cette dernière fait de la récursion pour vérifier si le graphe est acyclique. Pour adder  $n \geq 10$ , le graphe est trop grand, on atteint alors la profondeur maximale de récursion.

#### IV) Vérification de la propriété principale du code de Hamming

Pour tester et réaliser la vérification de la propriété principale du code de Hamming nous avons créé un fichier à part (tp12\_exo3.py à la racine du projet).

Voici son contenu :

```
from modules.bool_circ import *

# On verifie qu'on obtient bien les memes bits au depart et apres l'evaluation
# On test cela pour tous les entiers allant de 0 a 15 (inclus)

print()
print("Verif 1: on obtient bien les bits de depart a la fin")
print()

enc = bool_circ.enc()
dec = bool_circ.dec()

for i in range(16):
    reg = bool_circ.registre(i, size=4)

    comp = dec.compose(enc)
    comp.icompose(reg)

    comp.evaluate()

    outputs_comp = comp.get_output_ids()
    outputs_reg = reg.get_output_ids()

    assert len(outputs_comp) == len(outputs_reg) == 4

    for i in range(4):
        output_comp = comp[outputs_comp[i]]
        output_reg = reg[outputs_reg[i]]
        n_comp = comp[output_comp.get_parents_ids()][0]
        n_reg = reg[output_reg.get_parents_ids()][0]

        assert n_comp.get_label() == n_reg.get_label()

    nodes_comp = comp.get_nodes()
```

```

nodes_reg = reg.get_nodes()

assert len(nodes_comp) == len(nodes_reg) == 8

print()
print("OK")
print()

def add_neg(dec, no_input):
    """
    Ajoute une porte negative sur un decodeur
    dec : bool_circ -> correspond a un decodeur d'Hamming
    no_input : int -> l'input que l'on souhaite erroner
    """
    dec_input = dec.get_input_ids()

    i_no = dec[no_input]
    fils_id = i_no.get_children_ids()[0]
    dec.remove_edge(no_input, fils_id)

    dec.add_node('~', {no_input: 1}, {fils_id: 1})

    dec.set_input_ids(dec_input)

# On verifie qu'avoir une erreur dans le decodeur ne change pas le resultat
# On va tester cela sur chaque input, pour chaque nombre allant de 0 a 15 (inclus)

print()
print("Verif 2: avoir une erreur dans le decodeur ne change pas le resultat")
print()

enc = bool_circ.enc()

for j in range(7):

    dec = bool_circ.dec()
    add_neg(dec, j)

    for i in range(16):
        reg = bool_circ.registre(i, size=4)

    comp = dec.compose(enc)

```

```

comp.icompose(reg)

comp.evaluate()

outputs_comp = comp.get_output_ids()
outputs_reg = reg.get_output_ids()

assert len(outputs_comp) == len(outputs_reg) == 4

for i in range(4):
    output_comp = comp[outputs_comp[i]]
    output_reg = reg[outputs_reg[i]]
    n_comp = comp[output_comp.get_parents_ids()[0]]
    n_reg = reg[output_reg.get_parents_ids()[0]]

    assert n_comp.get_label() == n_reg.get_label()

nodes_comp = comp.get_nodes()
nodes_reg = reg.get_nodes()

assert len(nodes_comp) == len(nodes_reg) == 8

```

```

print()
print("OK")
print()

```

# On verifie qu'avoir 2 erreurs dans le decodeur ne permet pas de retrouver le resultat

# On va tester cela sur chaque input (2 par 2), pour chaque nombre allant de 0 a 15 (inclus)

```

print()
print("Verif 3: avoir 2 erreurs dans le decodeur change le resultat")
print()

```

```

enc = bool_circ.enc()

```

```

for j in range(6):

```

```

    dec = bool_circ.dec()
    add_neg(dec, j)
    add_neg(dec, j + 1)

```

```

    for i in range(16):

```



```

reg = bool_circ.registre(i, size=4)

comp = dec.compose(enc)
comp.icompose(reg)

comp.evaluate()

outputs_comp = comp.get_output_ids()
outputs_reg = reg.get_output_ids()

assert len(outputs_comp) == len(outputs_reg) == 4

flag = True

for i in range(4):
    output_comp = comp[outputs_comp[i]]
    output_reg = reg[outputs_reg[i]]
    n_comp = comp[output_comp.get_parents_ids()][0]
    n_reg = reg[output_reg.get_parents_ids()][0]

    if n_comp.get_label() != n_reg.get_label():
        flag = False

nodes_comp = comp.get_nodes()
nodes_reg = reg.get_nodes()

assert len(nodes_comp) == len(nodes_reg) == 8

assert flag == False

print()
print("OK")
print()

```

Le but de ce fichier est de vérifier que le message envoyé par l'encodeur est bien retrouvé par le décodeur. Puis de retrouver le message malgré une erreur dans le décodeur. Et enfin, de montrer qu'on ne retrouve pas le message s' il y a deux erreurs dans le décodeur.

Lorsqu'on exécute le fichier dans le terminal, un message indiquant que le début du test commence s'affiche, puis soit il s'affiche "OK" si tout s'est bien déroulé, soit il s'affiche une erreur, soulevée par un assert, si quelque chose s'est mal passé.

```
λ dev/py/projet_info-ldd2_im python3 tp12_exo3.py
```

Verif 1: on obtient bien les bits de depart a la fin

OK

Verif 2: avoir une erreur dans le decodeur ne change pas le resultat

OK

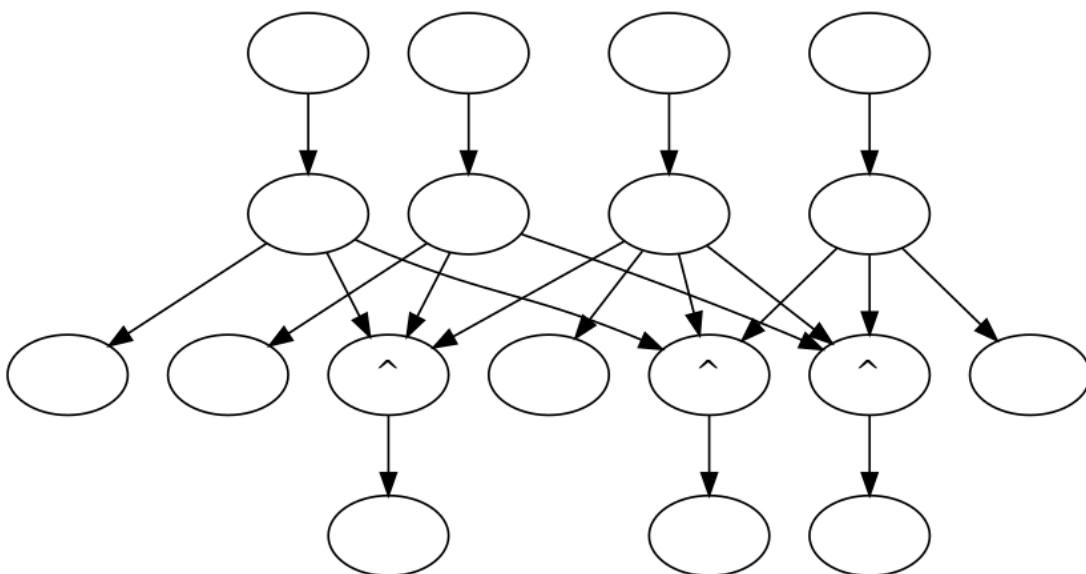
Verif 3: avoir 2 erreurs dans le decodeur change le resultat

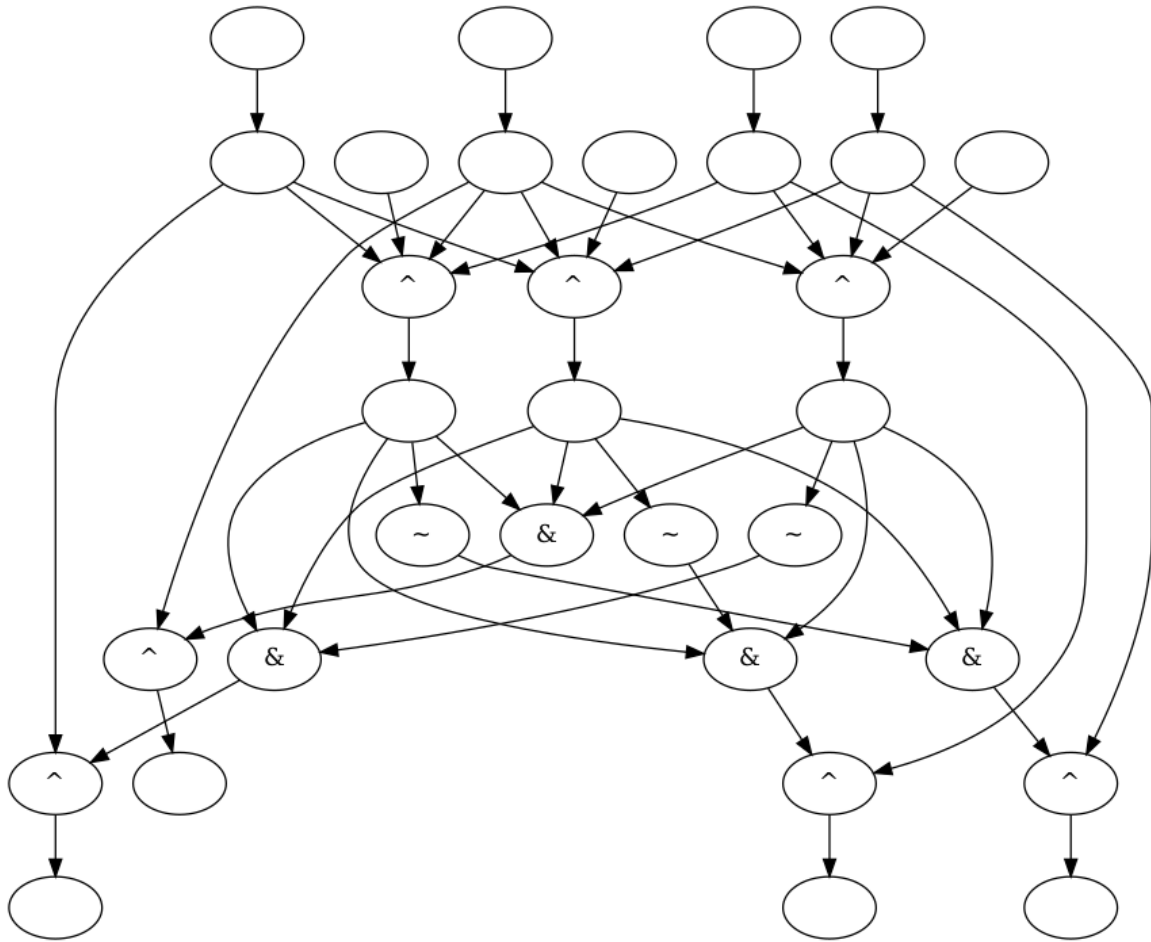
OK

```
λ dev/py/projet_info-ldd2_im |
```

A) On retrouve le message (sans erreur dans le décodeur) :

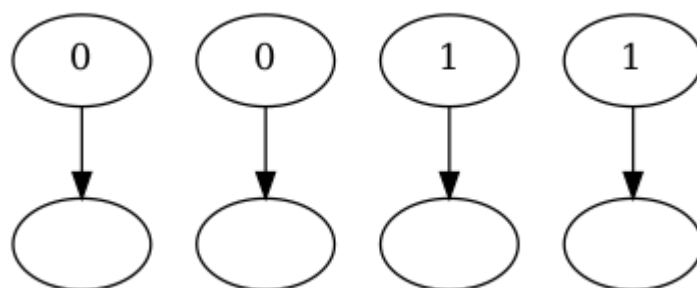
Au départ on créer un encodeur et un décodeur. Puis on va vérifier que si on envoie tous les nombres possibles entre 0 et 15 inclus alors on retrouve bien le bon nombre à la fin.



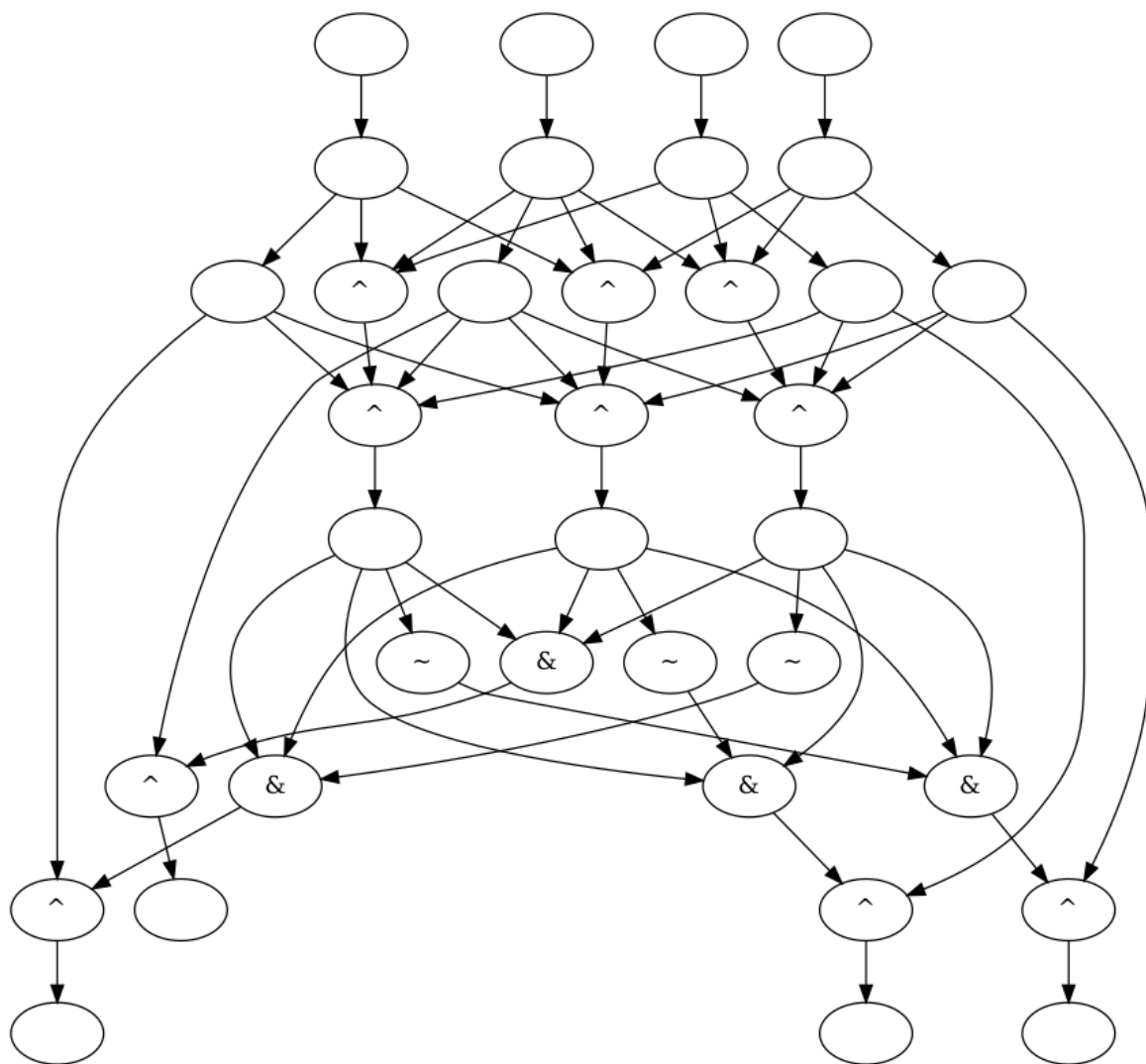


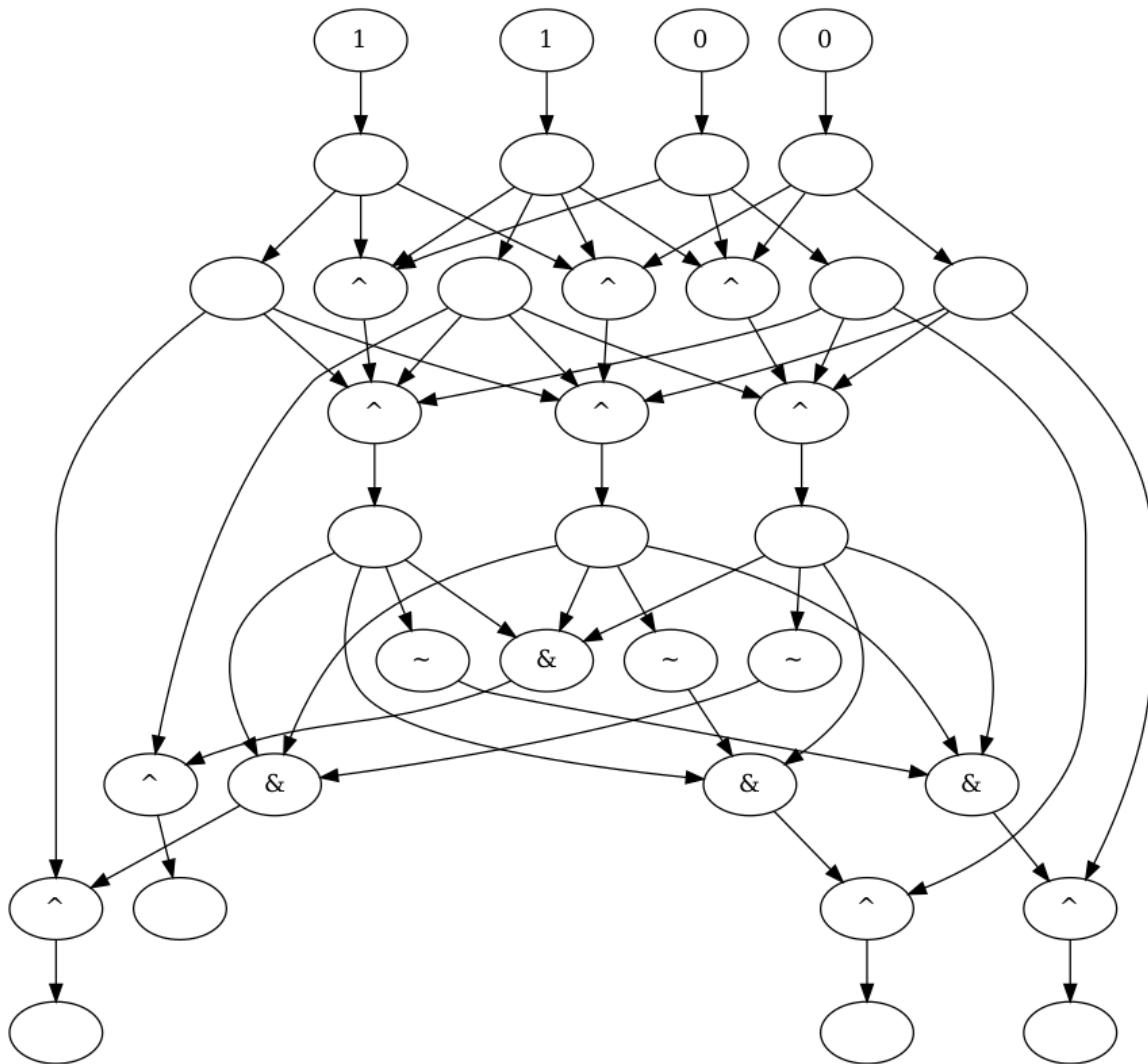
Donc, on va créer un registre correspond au nombre  $i$  ( $0 \leq i \leq 15$ ) de taille 8 (codé sur 8 bits).

Ici on prend 3 :



Puis on va composer le décodeur avec l'encodeur et également composé le résultat avec le registre précédent (cela permet d'initialiser les inputs).





(Les inputs sont bien dans le bon ordre, c'est l'affichage du display qui donne l'impression que non)

On va donc évaluer cette composition en espérant retrouver le bon résultat.

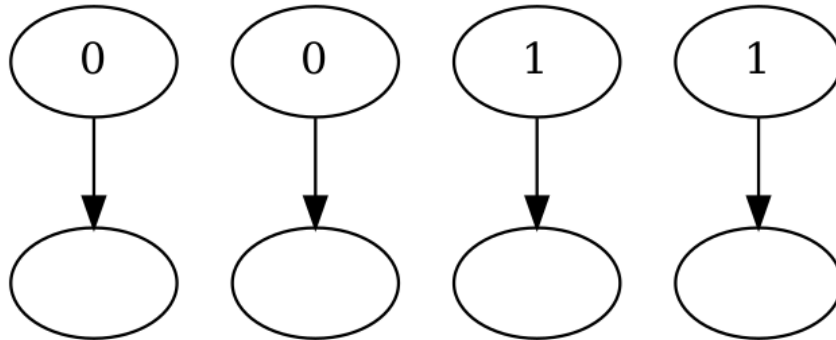
Pour ce faire, on vérifie que le nombre total d'outputs est égal à 4 (le résultat est codé sur 4 bits). On vérifie également que le nombre total de nœuds est égal à 8 (4 outputs + 4 bits réponse).

On va maintenant vérifier que les bits réponses sont bien les bons. Pour ce faire, on crée un registre correspond à la réponse attendue, et on vérifie que les bits réponses obtenues par l'évaluation soient bien les mêmes que ceux du registre.

Si tous tests sont passés sans erreurs alors on affiche "OK" ce qui signifie que le résultat obtenu est celui souhaité et est correct.

Sinon, une erreur aurait été soulevée. (on a également utilisé print et display lors de la programmation de tout cela pour vérifier si tout était en ordre)

Ici, tout est vérifié, on obtient donc (ce qui correspond bien à 3) :



#### B) On retrouve le message (avec 1 erreur dans le décodeur) :

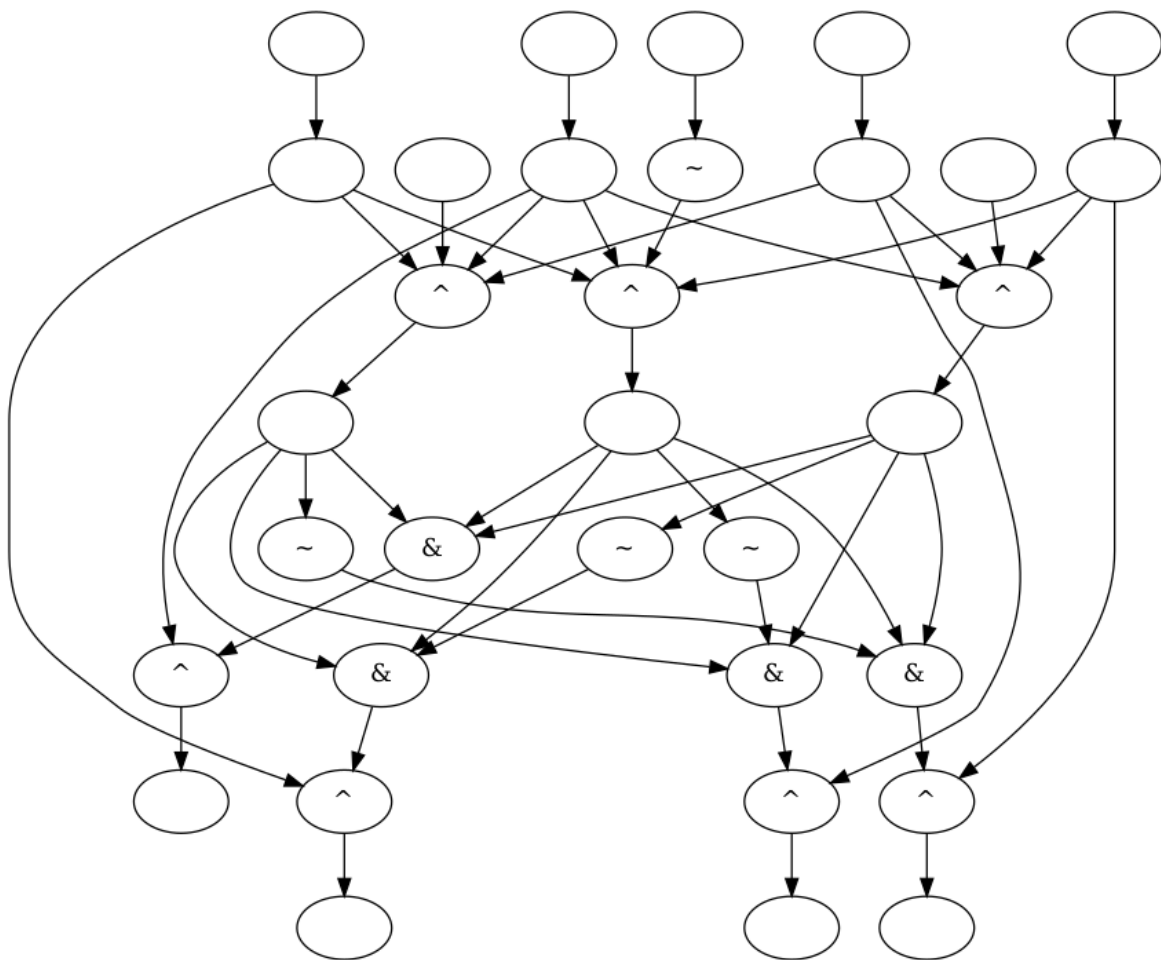
On répète exactement la même opération sauf qu'on va introduire une erreur dans le décodeur.

On va ajouter une porte “~” au décodeur et vérifier qu'on obtient bien la bonne réponse. (on va tester l'ajout de la porte “~” pour chaque input du décodeur).

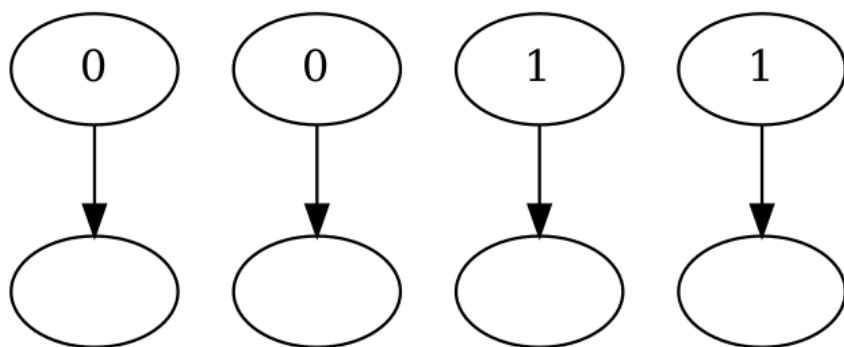
Pour rajouter une porte “~” nous avons créé une fonction qui prend en argument un décodeur et qui renvoie un décodeur modifié avec une porte “~” suivant un input.

Voici le code cette fonction :

```
def add_neg(dec, no_input):  
    """  
    Ajoute une porte negative sur un decodeur  
    dec : bool_circ -> correspond a un decodeur d'Hamming  
    no_input : int -> l'input que l'on souhaite erroner  
    """  
    dec_input = dec.get_input_ids()  
    i_no = dec[no_input]  
    fils_id = i_no.get_children_ids()[0]  
  
    dec.remove_edge(no_input, fils_id)  
    dec.add_node('~', {no_input: 1}, {fils_id: 1})  
    dec.set_input_ids(dec_input)
```



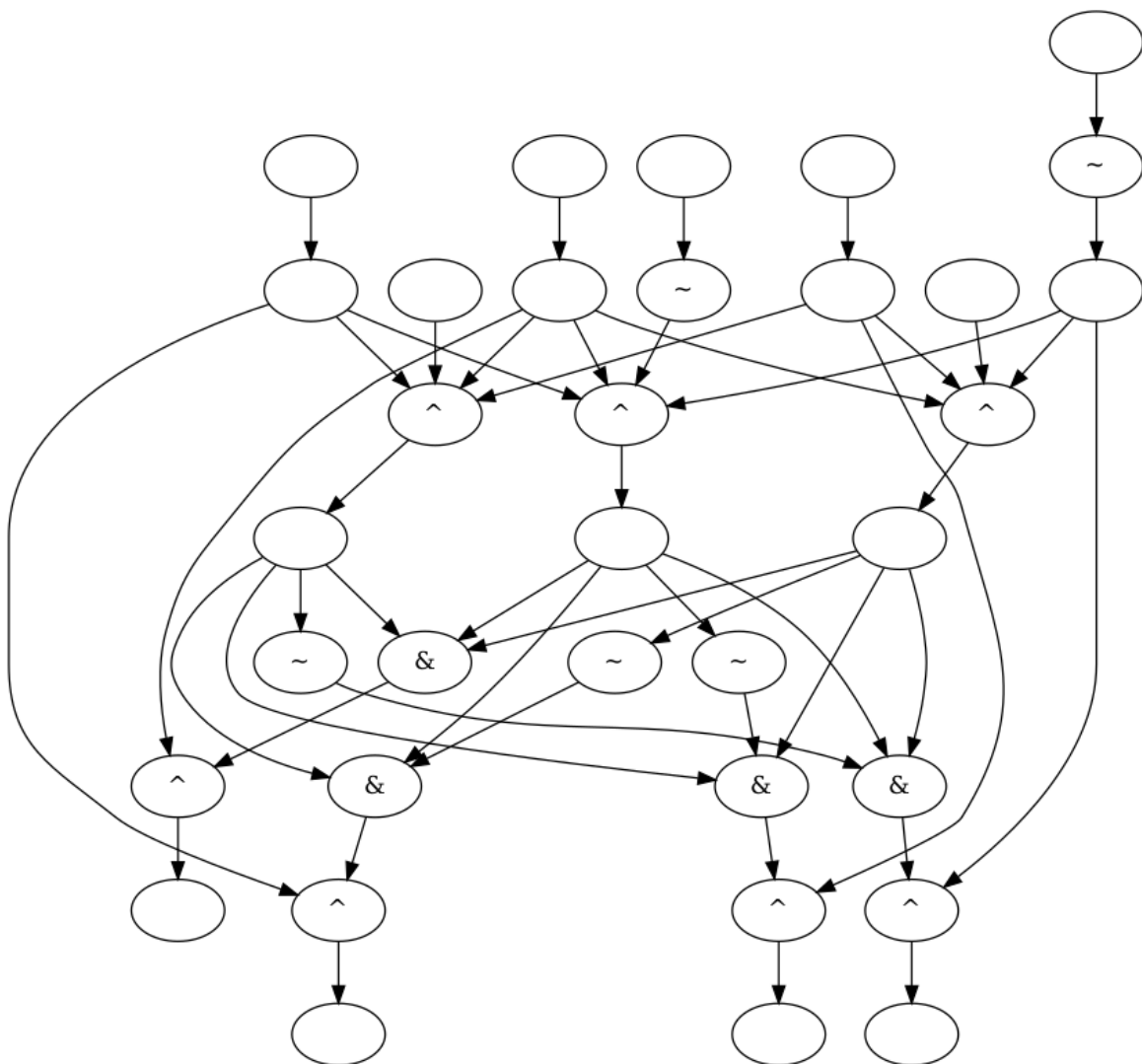
Ici, tout est vérifié, on obtient donc (ce qui correspond bien à 3) :



C) On ne retrouve pas le message (avec 2 erreurs dans le décodeur) :

On va répéter la même opération sauf qu'on va introduire 2 erreurs dans le décodeur.

On va ajouter deux portes “~” au décodeur et vérifier qu'on n'obtient pas la bonne réponse. (on va tester deux à deux tous les inputs du décodeur).





Ici, une erreur est apparue, on obtient donc (ce qui ne correspond pas à 3) :

