PROJET INFO – Rapport

PERROD Romain, BOHÈRE Matthieu

Ce rapport traite uniquement des points demandés à la fin du TD12.
Pour pouvoir exécuter les tests, lancer seulement la commande « *./tests.sh* ». S'ils prennent du temps, c'est normal : la simplification et l'évaluation d'un Adder5 et Half_Adder5 est assez conséquente (cf le nombre de portes plus bas)

Tous les calculs réalisés pour les parties 3 et 5 se trouve dans « *worksheet.py* ».

Partie 1 : Arborescence

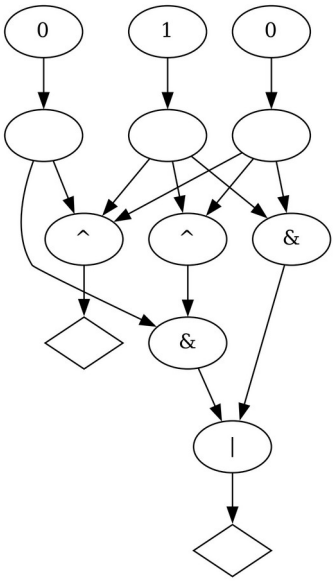
projet-info/
- dot_files/
  - tous les fichier « *.dot* » et les outputs en « *.pdf* », organisé dans différents dossiers
- modules/
  - adjacency_matrix.py
  - bool_circ/
    - bool_circ_classmethod_mx.py
    - bool_circ.py
    - bool_circ_simplify_mx.py
    - bool_circ_transform_mx.py
  - node/
    - node_add_remove_mx.py
    - node.py
  - open_digraph/
    - open_digraph_add_remove_mx.py
    - open_digraph_classmethod_mx.py
    - open_digraph_composition_parallel_mx.py
    - open_digraph_path_mx.py
    - open_digraph.py
    - open_digraph_save_mx.py
- rapport/
  - images pour le rapport + rapport en « *.odt* »
- rapport_bohere_perrod.pdf
- README.md
- sujets/
  - TD01.pdf
  - ...
  - TD12.pdf
- tests/
  - adjacency_matrix_test.py
  - bool_circ_test.py
  - node_test.py
  - open_digraph_test.py
- tests.sh
- worksheet.py

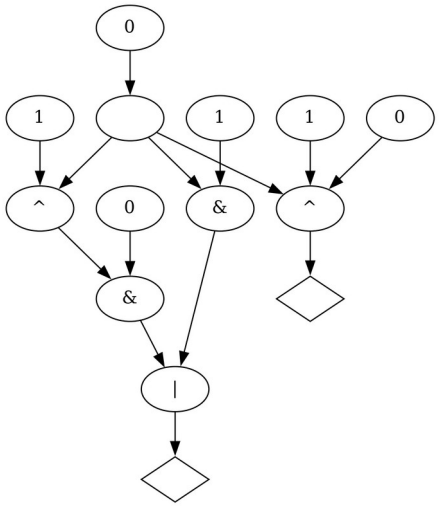
Partie 2 : Addition

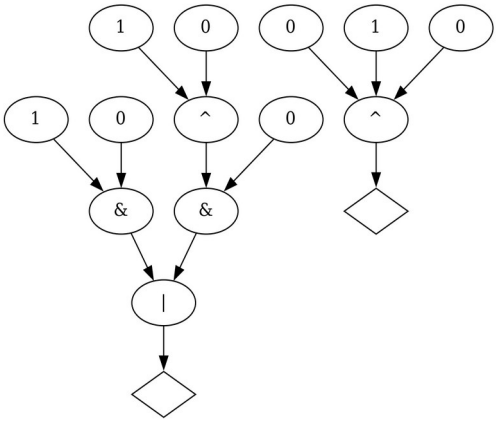
Pour tester le fonctionnement de Half_Adder, il y a dans les fichiers tests ces additions :
- Avec Half_Adder2 :
    - $4 + 6 = 0100_2 + 0110_2 = 1010_2 = 10$, avec 0 comme retenue
    - $12 + 6 = 1100_2 + 0110_2 = 10010_2 = 18$, ou $0010_2 = 2$ avec 1 comme retenue
- Avec Half_Adder5 (nous ne détaillerons pas les calculs):
    - $01001011001011101010010110101010_2 + 01011010100111010000010110101101_2$
      $= 01010010111001011101010110101010111_2$
      (1261348266 + 1520240045 = 2781588311)
    - 1 + 2 = 3
    - 1 + 0 = 1
    - 0 + 1 = 1 (pour tester si la commutativité fonctionne)
    - 15 + 36 = 52
    - 2846423656 + 23243 = 2846446899

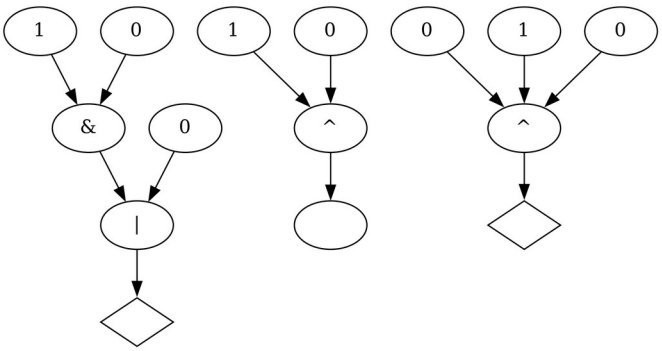
Comme le prouve les fichiers tests, les additions sont correctes (car la méthode evaluate renvoie le résultat sous forme de chaîne de carractère représentant le résultat binaire).
Pour simplifier les tests, nous avons aussi rajouté des méthodes qui prennent direcement deux entiers et qui renvoie le résultat sous forme entière (rendant les tests plus lisibles).

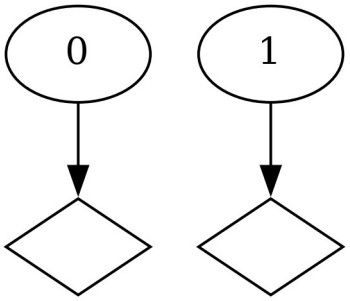
Voici 5 images représentant l'addition de 1 + 0 avec Adder0 :











Partie 3 : Additionneur

Parlons du plus simple : le nombre de porte. En effet, on a cette relation de récurrence par construction :

$$\begin{cases} u_0 = 8 \\ u_{n+1} = 2u_n \end{cases}$$

En effet, Half_Adder(n+1) consiste en deux Half_Adder(n), rien de plus. Ainsi, on obtient facilement que $u_n = u_0 2^n = 8 * 2^n = 2^{n+3}$.
Ainsi, il est clair que la simplification d'un Half_Adder(5) est longue, de par notre algorithme de complexité discutable, et du fait qu'il y ait $2^8 = 256$ portes avant simplification.

NB : ici on compte les nœuds copies comme des portes.

Maintenant, la profondeur. Par différents essais, on conjecture ceci :

$$\begin{cases} u_0 = 4 \\ u_{n+1} = 2u_n - 1 \end{cases}$$

En effet, cela se trouve facilement : Half_Adder(n+1) consiste en deux Half_Adder(n) et il y a un nœud en commun (la retenue), d'où l'expression récurrente. Et donc, on a que la profondeur est de $u_n = a^n \left(u_0 - \frac{b}{1-a} \right) + \frac{b}{1-a} = 2^n (4-1) + 1 = 3 \times 2^n + 1$. Ainsi, on a que la profondeur d'un Half_Adder(5) est de $3*2^5 + 1 = 97$.

NB : ici on ne compte que les nœuds, et pas les inputs et outputs

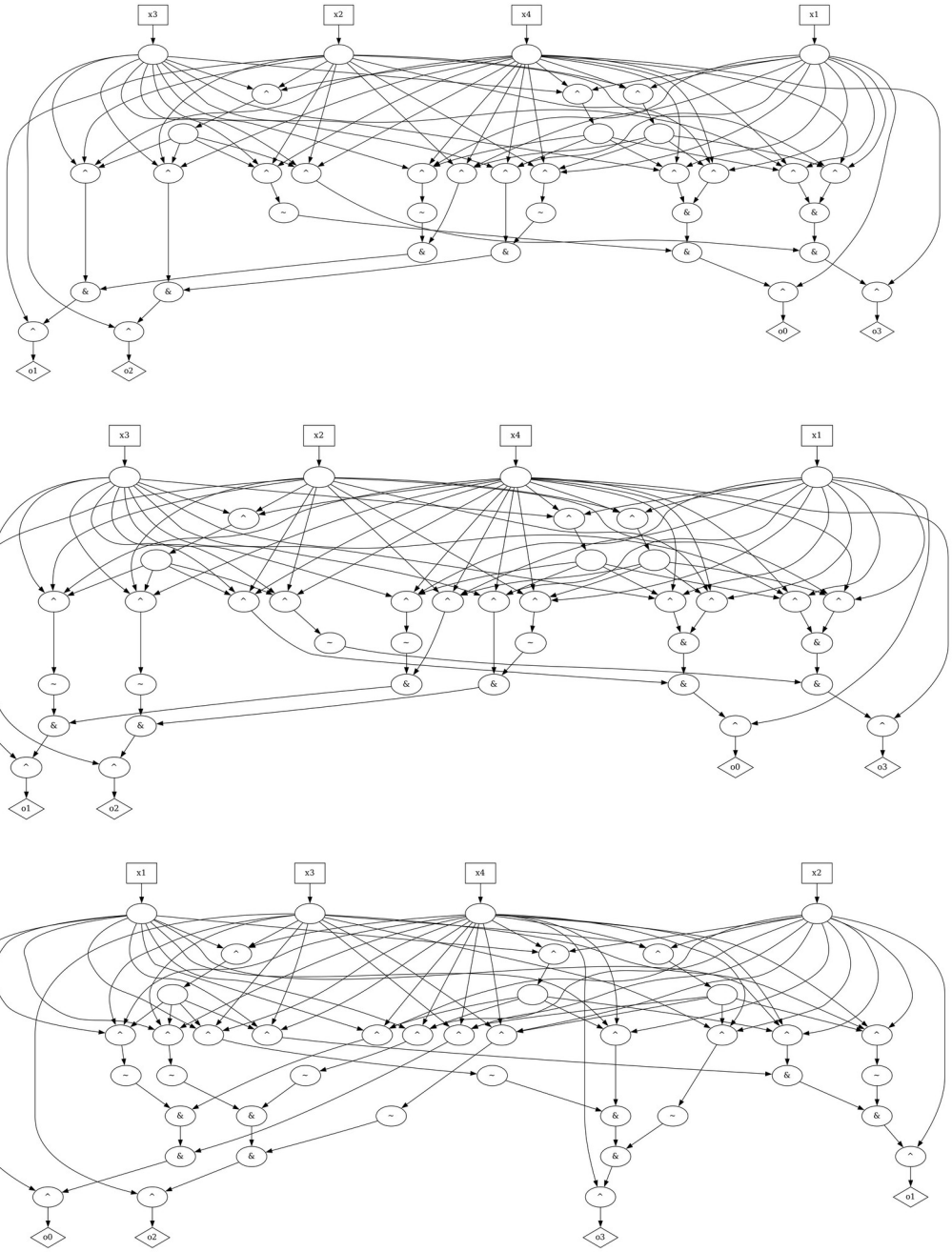
Pour finir : le plus court chemin. Lorsque nous faisons des tests pour conjecturer une relation, le résultat peut paraît surprenant : le plus court chemin entre une entrée et une sortie est toujours de 3 ! Et en réfléchissant, c'est normal : c'est le chemin depuis la retenue vers le bit le plus faible du résultat.

Par rapprot à Carry Lookahead : comme vous pouvez le voir dans notre code, il est implémenté. Seulement, l'ordre des inputs et outputs ne permet pas de pouvoir l'évaluer. En revanche, voici-les résultats des mêmes questions, sans détails :
- Nombre de porte : $u_{n+1} = u_n + u_0$ d'où $u_n = (n+1) u_0 = 53(n+1)$
- Profondeur : bizarrement on a que le premier a pour profondeur 10, puis les autres ont pour profondeur 11 (au moins ça montre l'efficacité accrue!)
- Chemin le plus court : 3, pour les mêmes raisons que Half_Adder(n)

Partie 4 : Code de Hamming

Pour tester l'encodage d'Hamming, nous avons fait très simple : on compose l'encodeur puis le décodeur, et on l'évalue sur toutes les valeurs possibles (0 à 15). Ensuite, notre méthode de test vérifie l'égalité avec l'entier de base. De plus, nous testons toutes les combinaisons possibles avec une erreur (il y en a donc 7 car nous implémentons seulement pour n = 3).
Et enfin, pour tester qu'avec deux erreurs le décodage ne fonctionne plus, nous avons tout simplement choisis arbitrairement deux erreurs et avons testé toutes les valeurs possibles (0 à 15), en remarquant qu'aucun des codes décodés ne correspondent aux entiers de base.

Voici le codeur + décodeur, avec 0, 1 et 2 erreurs :



Partie 5 : Taux de simplification

En générant mille graphes aléatoires, comportant 5 à 20 nœuds aléatoirement, de 2 liens maximum entre deux nœuds, ainsi que 3 inputs et 3 outpus, on obtient un rapport de « nombre de nœuds après simplification » sur « nombre de nœuds avant simplification » de 95%, soit en moyenne 5% d'un graphe qui est simplifiable : cela est dû à une méthode de simplification pas complète (l'exemple d'Adder0 est parlant).
Si nous avons eu plus de temps pour en implémenter d'autres, peut-être que ce pourcentage de simplification serait plus grand.