# An Implementation of Boolean Circuits

Alex Scofield teruel, François-Elie Ingwer, Hugo Taile Manikom

May 2024

## Contents

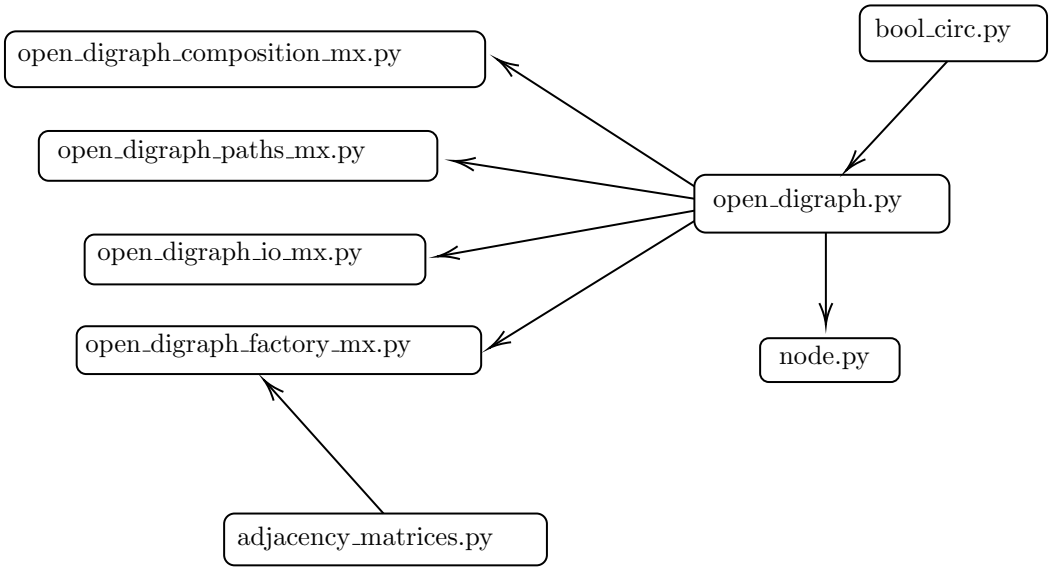


Figure 1: Dependency graph

# 1 Introduction

The following is a short document outlining some of the functionalities our team has implemented regarding boolean circuits, in Python, during the second semester of *LDD2 IM*.

The objective of the project was to, ultimately implement boolean circuits. In order to do this, we have implemented a simple Python library capable of managing graphs, and, in particular open directed graphs, of which boolean circuits are a subtype. We have attempted to render the code as user-friendly as possible, so that other developpers could hypothetically easily extend it and understand it (which is why it is heavily documented and explained), and we have tested the code during all stages of development.

# 2 Project structure

In the spirit of keeping the code as well organised and modular as possible, we have structured the project into two folders: the *modules* folder, which contains the implementation of all of the features, which itself contains a subfolder, *open_digraph_mixins*, containing mixins which the main *open_digraph.py* module inherits from, and the *tests* folder, which we will briefly describe bellow.

The advantage of this architecture is that the code is very loosely coupled. In particular, it would be easy to construct other types of graphs (distinct from open directed graphs, which have been the focus of this project) using the same *node* type. It would also be easy to construct other types of *open directed graphs*, distinct from boolean circuits, using the same *open_digraph* type. Figure 1 explicits the dependencies between the different subcomponents of the library we have developped.

All tests are included in the *tests* folder. In the earlier stages of development we followed

relatively strict TDD(*test-driven development*). Unit tests have been written using the *unittest* Python framework. In order to run all of these tests it suffices to execute the file *run_tests.py*[1].

# 3 Random Boolean Circuits

The *bool_circ* class inherits from *open_digraph*, which, through a mixin, implements random generation of graph. We have developed a simple algorithm which generates random boolean circuit using the methods defined in the parent classes. Figure 2 shows one such graph.

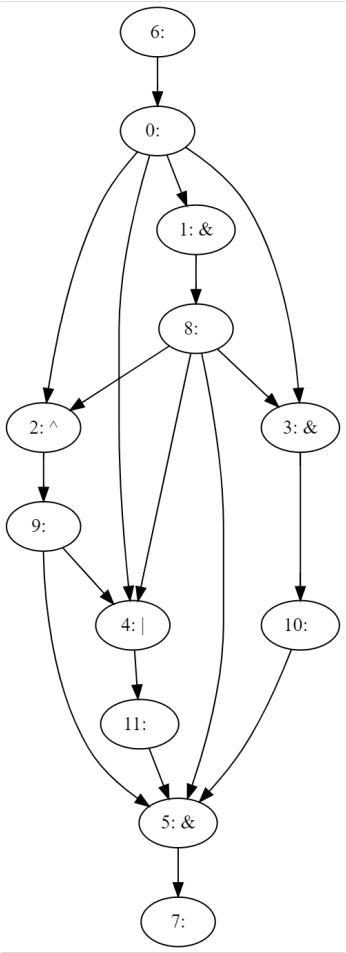


Figure 2: Randomly generated boolean circuit

Boolean circuits are constructed following a strict set of rules regarding the nodes they may possess, which are detailed out in the specification of this project.

[1]This must be done from within the *tests* folder to ensure that imports work correctly, as we have opted to use absolute imports.

# 4 Adders and Half-Adders

A particularly interesting type of boolean circuit which is well-suited to be implemented using our library is the adder (and half-adder) circuit. Given its recursive definition, it was possible to implement a function that could construct adders of any given size. Figure 3 shows two such circuits. We also began to implement a method that would evaluate a boolean circuit, however, due to time constraints we were unable to complete it.

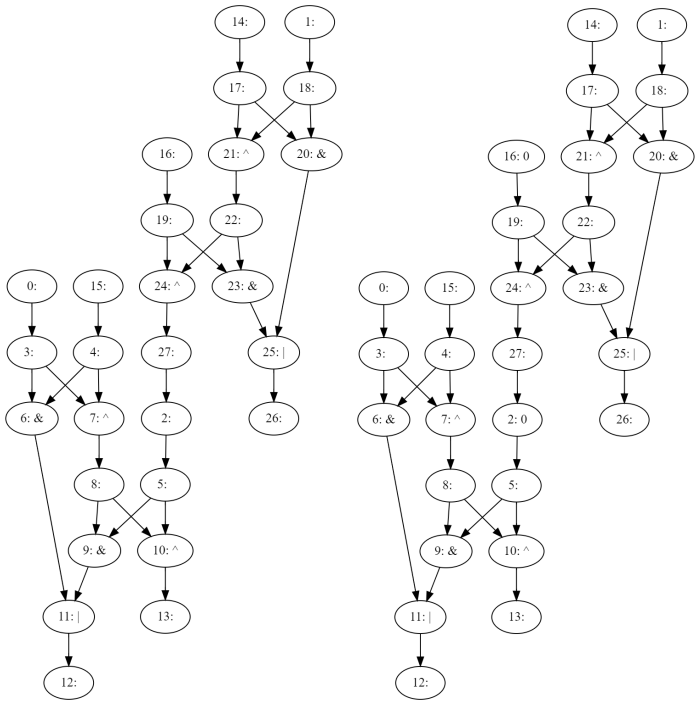


Figure 3: 3 bit adder and half-adder

The shortest path between an input and an output of an adder is 3 for the simplest case (adding just two bits together). This is distance between the carry and its corresponding output. As for bigger adders, the shortest path is fixed at 4, which is the distance between the initial carry and the first output bit. These results have been confirmed through code using *Dijkstra's algorithm*, and are presented in the *worksheet.py* file, as an example of how our library can be used in practice.

Given the recursive nature of the construction of adders and half-adders their depth is expected to be exponential as a function of their size. However, the code provided in *worksheet.py* that verifies this result experimentally does not manage to calculate the size for big adders, presumably because of the complexity of the topological sort required to do so.

# 5 Hamming Encoders

Another interesting application of boolean circuits is the implementation, through logic gates of Hamming codes. Figure 4 shows a 4 bit Hamming encoder bext to its corresponding decoder.

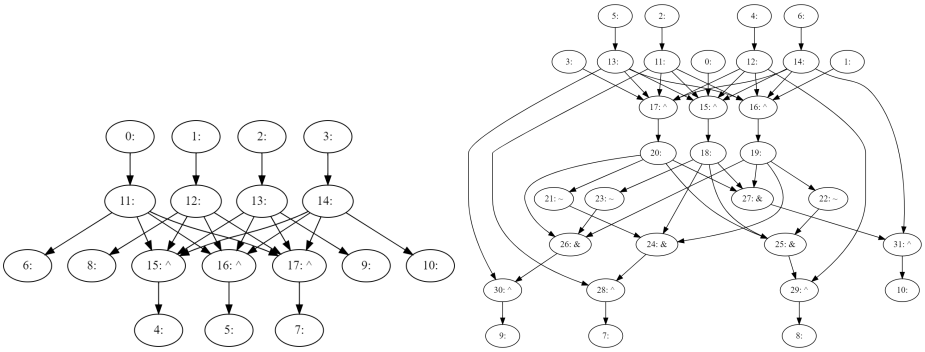


Figure 4: 4 bit Hamming encoder and decoder

When composing these two graphs (as shown in figure 5, if there is an "error" introduced (by flipping a single bit) between the encoder (the sender of a hypothetical message) and the decoder (the reciever of said message), the result is expected to be the identity graph. However, since the function required to evaluate boolean graphs has not been completely implemented, we cannot present empirical evidence of this result here.

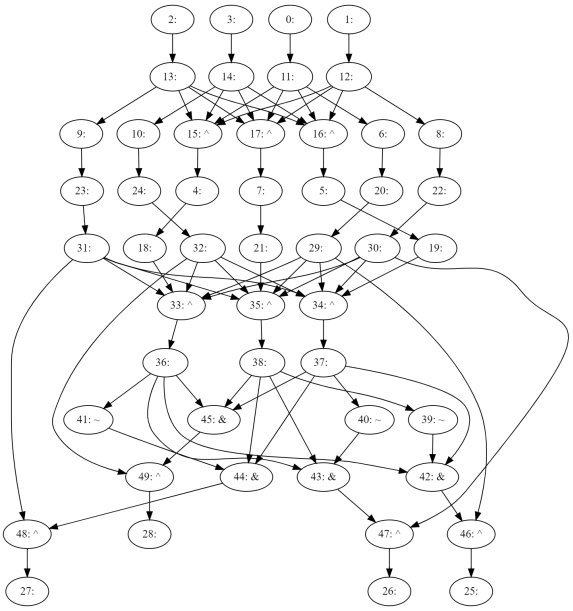


Figure 5: 4 bit Hamming decoder composed with encoder