Graphical output for a quantum circuit description language

Boyer Benjamin
(supervisor: Peyras Q. Nalpon N. Chareton C.)

Technical Report, June 2024

In the context of the QBriks project, we present a novel tool for the efficient design of parameterized quantum circuits, facilitating rapid circuit creation and analysis. The core emphasis of this work lies in the development of an abstract and optimized circuit representation. Furthermore, we address the implementation of macros and loops, as well as the integration of additional tools to enhance functionality.

Dans le cadre du projet QBriks, nous présentons un nouvel outil pour la conception efficace de circuits quantiques, facilitant la création et l'analyse rapides de ces derniers. L'accent est mis sur le développement d'une représentation abstraite et optimisée des circuits. En outre, nous abordons l'implémentation des macros et des boucles, ainsi que l'intégration d'outils supplémentaires pour améliorer la fonctionnalité.

Keywords
Quantum circuit, Graphical renderering, Circuit description, Macro & Loop

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
`name.surname@lre.epita.fr` – http://www.lre.epita.fr/

Copying this document

Contents

1 Introduction

Recent advances in quantum computing are opening up new horizons for solving complex problems. However, to take full advantage of these progresses, it is essential to have working environments adapted to this field. QBricks has been developed with this in mind. QBricks is an open-source environment dedicated to the formal verification of quantum programs. The tools available in this environment include HBricks, a quantum circuit description language, to which we intend to add a tool for visualizing the circuits described. Finally, we address the issue of macros and loops within circuits, and finish with a standard output format.

Remerciements

Avant de continuer, j'aimerais adresser mes plus sincères remerciements à l'ensemble des membres du Laboratoire de Recherche de L'Epita (LRE), et plus particulièrement à mes encadrants: Quentin Peyras (LRE), Nicolas Nalpon (CEA-list) et Christophe Chareton (CEA-list) qui m'ont grandement aidé dans ce travail et sans qui je n'aurais pas eu la chance de faire le parcours recherche.

Prerequisites

Numerous reminders are given throughout the document, but it is advisable to be familiar with the following concepts: Linear algebra, Programming language (Python), Graph theory, Basics of quantum computation (a reminder is given at the beginning).

Contributions

- We present an abstract representation of quantum circuits using graphs together with an algorithm for its optimal traversal.
- A first solution for macros (implemented) and loops (non-implemented) in circuits.
- Tikz compilation of circuits

2 Preliminaries

Quantum computing leverages the principles of quantum mechanics to perform computations. The fundamental unit of quantum information is the **qubit**, which, unlike a classical bit, can exist in a superposition of states.

2.1 Qubits and Superposition

A qubit is represented as a vector in a two-dimensional complex vector space, with basis states $|0\rangle$ and $|1\rangle$. A general qubit state is:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.

2.2 Quantum Gates

Quantum gates manipulate qubits through unitary operations. Common single-qubit gates include:

- **Hadamard gate** (H):
  $$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$
  which creates superposition.
- **Pauli-X gate** (X):
  $$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
  which acts as a quantum NOT gate.

Multi-qubit systems are described by tensor products of individual qubit states. For example, a two-qubit state is:

$$|\psi\rangle = \alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$.

2.3 Quantum Circuits

Quantum circuits are composed of quantum gates applied in sequence to qubits. Each quantum gate represents a unitary operation on the state of the qubits.
A simple quantum circuit with two qubits might involve the application of a Hadamard gate followed by a CNOT gate. The circuit can be represented as:

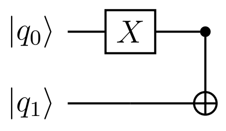


Figure 2.1: Bell state preparation circuit

As you can see, unlike algebraic forms, circuits bring clarity and structure to calculations.

2.4 Measurement

Measurement in a quantum circuit collapses the qubit states into one of the basis states. For example, measuring the entangled state $|\psi\rangle$ will result in either $|00\rangle$ or $|11\rangle$, each with probability 0.5.It is not possible to measure α and β of the $|\psi\rangle$ status separately.

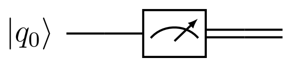


Figure 2.2: Measuring gate

Upon observing the output of the gate, it is noted that the wire is doubled. The aim is to clarify for the reader that the qubit, following measurement, transitions into a classical bit.

3 Literature review

Quantum circuit design tools can be broadly categorized into two groups: general-purpose quantum programming languages and specialized quantum circuit design tools.

General-purpose quantum programming languages, such as Qiskit, Cirq, and Q#, provide comprehensive environments for quantum computation, but they have limitations in certain areas. For instance, these languages do not typically offer well-developed macro functionality, which could be beneficial for automating repetitive tasks in quantum circuit design. Moreover, they do not support the direct representation of loops within quantum circuits, which can be a significant limitation for certain quantum algorithms that require iterative processes.

Qiskit, while benefiting from IBM's expertise and resources, is limited by the availability and capabilities of IBM's quantum hardware. Similarly, Cirq's effectiveness is tied to Google's quantum hardware, which may not be accessible or suitable for all users. Q#, while providing a robust environment for quantum program development, requires integration with Visual Studio, which may not be the preferred development environment for all users.

4 Core Research

In this chapter, we detail the work that led to the results described in the following chapter. These include:
- An abstract representation of quantum circuits implemented in Python.
- The ability to compose circuits using macros.
- The theoretical integration of loops in quantum circuits.

4.1 Abstract circuit representation

In order to properly address the problems related to the study and conception of quantum circuits, it is essential to build a solid model with which we can faithfully describe them. The question we need to ask ourselves is: in essence, what is a quantum circuit ? And more precisely, what is the minimum set of information we need to define it ?
These are fundamental questions, because once answered, they mark out the boundaries of what can be done with the chosen model.

To answer this question, it's important to break down what constitutes a quantum circuit and what distinguishes it from other types of circuit, notably classical ones.

1. **Representation of quantum information**
   - Qubits: Quantum circuits manipulate qubits, which are the basic units of quantum information. Unlike classical bits, qubits can exist in superposed states, allowing a linear combination of $|0\rangle$ and $|1\rangle$.
2. **Operations and gates**
   - Unary: Quantum gates are the elementary operations that modify the state of qubits. They are analogous to logic gates in classical circuits, but can be reversible and often linear.
   - Multi-qubit: Multi-qubit gates, such as the CNOT (controlled-NOT) gate and more complex control gates, allow the creation of intricate states, where the states of individual qubits cannot be described independently of other qubits.
3. **Measurement**
   - Measuring a qubit causes its quantum state to be projected onto one of the base states $|0\rangle$ or $|1\rangle$. This operation is probabilistic, the probabilities being determined by the amplitudes of the wave function prior to measurement.

In summary, a quantum circuit is an abstract object composed of wires that represent the lifelines of qubits. These wires consist of nodes that take the form of either qubits (typically at the start of the wire) or quantum gates.

Before proposing a model, it's important to define the needs it is intended to meet. First and foremost, we want a template that can compile the circuit it describes into other formatting languages, giving it a concrete and visual form (such as *LaTeX*).
Then we can define a few other important requirements that our model will have to satisfy, such as: **circuit composition** (with macro definition), **circuit simplification** (rewriting/optimization) and finally **visual rendering customization**. Other related functionalities can also be added.

4.1.1 Class diagram

From this, we can build a class diagram representing the general architecture of our model. We'll make sure our model is as modular as possible, and can interface with existing tools if required.

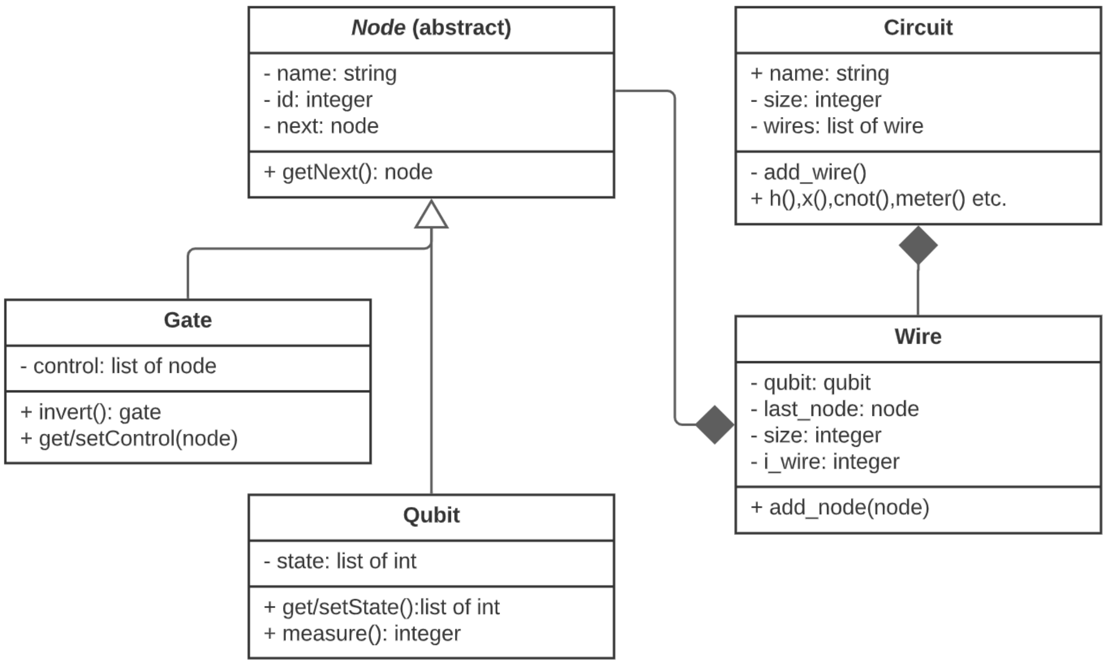


Figure 4.1: Simplified project class diagram

4.1.2 Associated graph

Once the *circuit* class has been instantiated, the object created can be seen as a directed graph. Representing a quantum circuit as a graph makes it possible to exploit well-established graph-theoretic techniques to solve complex quantum computing problems.

Definition: A directed graph G is a pair $G = (V, E)$ where V is a finite set of vertices (*nodes* in our case) and E is the set of ordered pairs $(x, y) \in V^2$ representing the graph's edges.

Definition: The degree δ of a vertex is the number of arcs incident on it. If $G = (V, E)$ is a directed graph, then $\delta(x) = Card(\{(v_1, v_2) \in E \mid v_1 = x \lor v_2 = x\})$

Graph traversal

In order to implement circuit calculations, we need to propose method for traversing the circuit graph.

In this section, v stands for any vertex of any graph, and the terms vertex and node are synonymous. We distinguish two types of vertex: those representing single gates (i.e. $\delta(v) <= 2$), which we'll call δ-*faible*[1], and those representing multiple gates (i.e. $\delta(v) > 2$), which we'll define as δ-*fort*[2].

Hereafter, we present a simple example of a quantum circuit (*fig. 4.2*) with its equivalent graph (*fig. 4.3*) representation. The δ-*faible* (respectively, δ-*strong*) nodes are depicted in gray (respectively, red). We will ensure that the graph maintains the same structure as the circuit to improve readability.

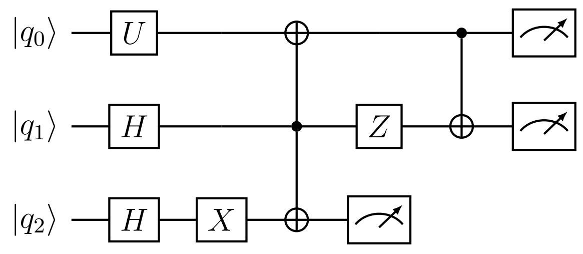


Figure 4.2: Quantum circuit 1

<=>

Gray: δ-*faible*; Red: δ-*fort*

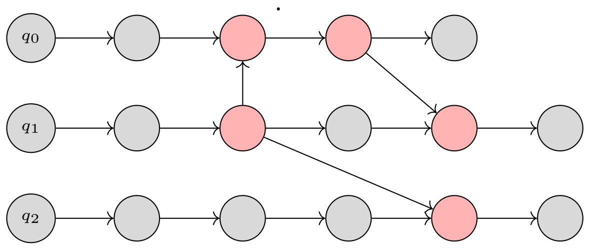


Figure 4.3: Graph of quantum circuit 1

[1] *Failble* means weak in French
[2] *Fort* means strong in French

1. **Initialization**: This step involves selecting all the nodes that represent the initial qubits of our circuit. Additionally, we prepare the qubits, their respective states, and various parameters before starting the graph traversal.

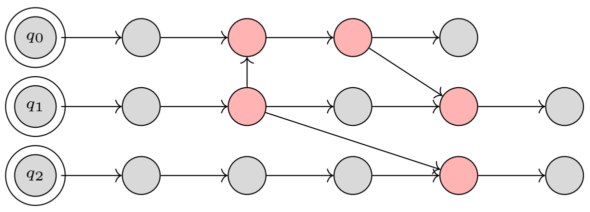


Figure 4.4: Step 0

2. **Traversal**: The graph is traversed "widthwise". Starting from the nodes selected during the initialization phase, we progress from successor to successor following two simple rules. Firstly, if the vertex is δ-*faible*, it is traversed normally. Secondly, if the vertex is δ-*strong*, it is marked with a marking vector and we wait until all its successors v_i such that $\delta^-(v_i) > 1$ are marked before traversing it.

   The second rule reflects the need to process all gates connected to the gate currently being processed. In particular, it applies to control gates for which it is necessary to know the controlled qubits before proceeding with any operation.

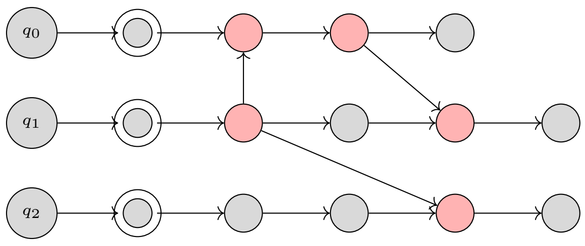


Figure 4.5: Step 1

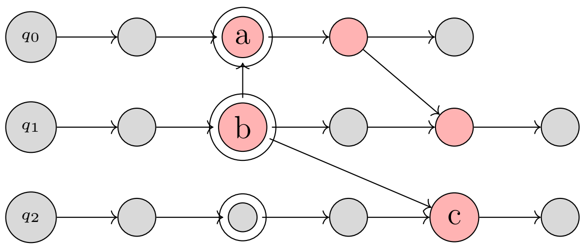


Figure 4.6: Step 2

Above right, vertices a and b will be "blocking" until vertex c is marked. In fact, $c \in succ(b)$ and $\delta^-(c) > 1$, therefore the second rule applies. This principle is repeated until all vertices have been visited. We'll leave the complete traversal to the reader in the appendix A of this document.

4.2 Advanced features

The abstract representation of quantum circuits enables us to manipulate them with ease and to construct increasingly complex operations. At this stage, we are able to perform the basic operations of a quantum circuit, such as the creation of registers, wires, qubits, as well as the use of usual and controlled gates, among others.

4.2.1 Macros

A *macro* is a predefined sequence of quantum operations that is stored under a unique name and can be reused in the composition of quantum circuits. This sequence of operations can include quantum gates, measurements and other quantum instructions.

By creating macros for frequently used sequences of quantum operations, users can simplify the description of their quantum circuit, reduce errors and improve readability of the circuit. Macros can also facilitate the modularization and reuse of quantum circuit components, making it easier for researchers and developers to collaborate and share quantum circuits.

Let M (for main) and S (for sub-circuit) be two quantum circuits. We give the function $Size(c : circuit)$, which determines the size of a circuit (i.e. the number of wires), and the function $Compose(c1 : circuit, c2 : circuit, wires : ListofInt)$, which maps the wires in the wires list of circuit $c1$ (output) to the wires in circuit $c2$ (input).
The $Compose()$ function is defined for any circuit M and S such that $Size(M) >= Size(S)$ From this, it's straightforward to connect the graphs associated with C and S together. This section is extended in chapter 5.

4.2.2 Loops

Quantum circuits are powerful representations for understanding algorithms. However, there is no official notation for describing parameter loops directly within the circuit. How do you draw a circuit with a '*while n > 0*' loop, for example ?
The notation with '...' in quantum circuits can be employed to represent sequences of repetitive operations concisely. This notation is particularly useful for describing circuits where a certain operation is applied identically several times. Loops can be used to model iterations in quantum circuits, facilitating the design and optimization of quantum algorithms.
For example, a sequence of U operations applied n times to a qubit q can be represented as :

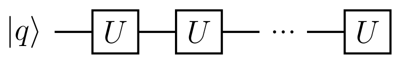


Figure 4.7: Loop example

Coin toss example

The term *coin toss* refers to a quantum operation that results in a superposition of states, in the same way that a fair coin toss can result in either heads or tails, with equal probability. In the context of quantum algorithms, a quantum coin flip is often used to create a superposition of states that can then be manipulated to solve a problem. A simple implementation of the coin toss can be realized using a *Hadamard* gate. Indeed, applied to a qubit in the $|0\rangle$ state, it creates a superposition of states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, which can be seen as a quantum coin in a superposition of heads and tails.

This algorithm is well-known in the literature. Now, let's imagine another version that would also perform a *coin toss*, but this time, the throw would repeat until a 1 is obtained. As you can understand, this version is directly related to the title of this section as it requires the addition of a *while* loop. Such an algorithm cannot be represented with conventional tools; however, the famous '...' allow us an explicit representation.

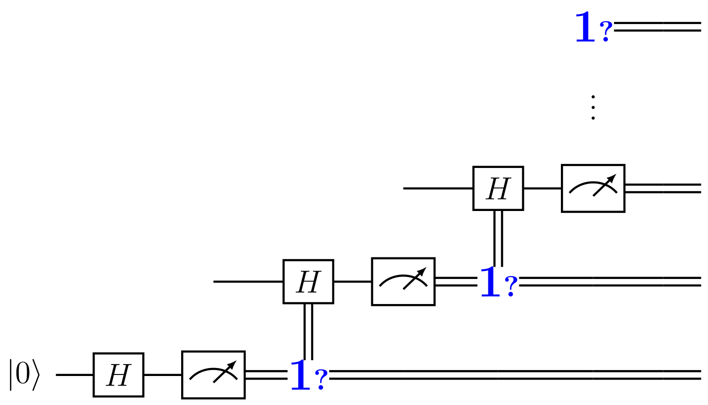


Figure 4.8: Coin toss extended

Above, a way of representing the *while* loop. Of course, this is only a preliminary approach to the problem. Moreover, to keep the diagram simple, we've used the '*1?*' syntax. Nevertheless, for a better formalism, we'd need to detail the diagram with, for example, a control gate representing the *if* instruction.

5 Results

This section presents the results of our research. We show the algorithms implemented using various examples, followed by a few comments and explanations where necessary.

5.1 Creating and manipulating circuits

First of all, to put the reader at ease, here's a typical implementation of a quantum gate (*Hadamard* in this case):

```python
def h(self, *index):
    # Hadamard gate apply on the i-th wires
    for i in index:
        wire = self.wires[i]
        if i >= self.size:
            raise IndexError("i cannot exceed " + str(self.size - 1))
        h_gate = Gate("H")
        wire.add_node(h_gate)
```

You'll notice the use of the various classes described in the section 4.1.1 of this document, and you'll also appreciate that the class model chosen offers concise and clear writing for the developer. This method comes from the *Circuit* class, so it operates on a *self* object, and takes a variable number of *index* arguments so that the gate can be applied to several wires at once.

Now, if we increase the level of abstraction and use the library's public methods, we can write this kind of algorithm:

```python
macro = Circuit(2)   # 2-wire circuit
macro.x(0)           # Apply NOT gate (X) on the 0th wire
macro.cnot(0,1)      # CNOT gate with 0th and 1st wire

main_circuit = Circuit(3)
main_circuit.h(0, 1, 2)      # Hadamard on the 0th, 1st, and 2nd wires
main_circuit.cnot(1, 0)
# Apply the sub-circuit 'macro' to the main circuit,
# mapping wires [0, 2] of 'main_circuit' to wires [0, 1] of 'macro'
main_circuit.func(c0, [0, 2])
# Measure the 0th, 1st, and 2nd wires of the main circuit
main_circuit.meter(0, 1, 2)
```

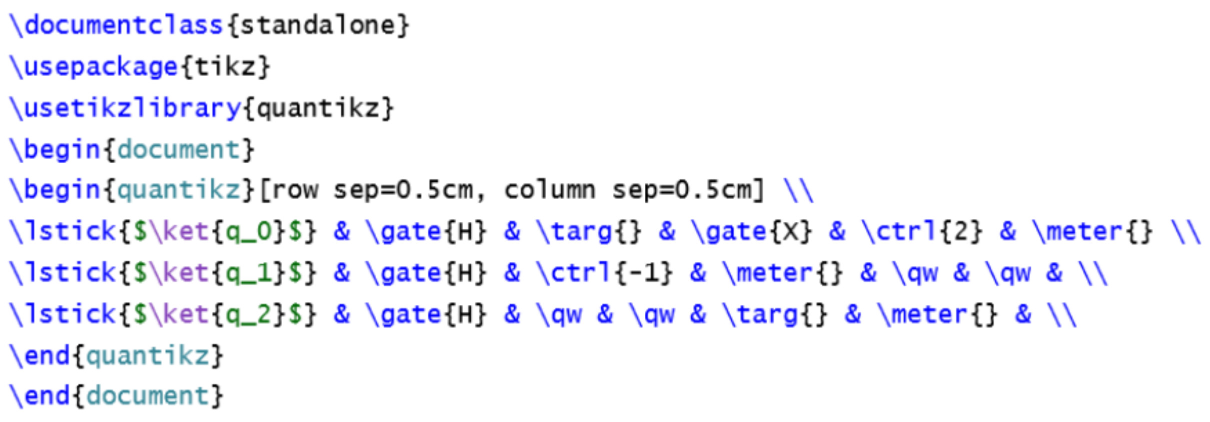
5.2 Visual renderings

For visual rendering, the *tikz()* method compiles the circuit in Tikz (a LaTeX package for schematic construction). This method implements the graph traversal described in the section 4.1.2. We chose this output format first, as it allows us to create a variety of figures, from simple diagrams to complex illustrations, while maintaining visual consistency with the document text. Its programmable syntax facilitates the automation and reuse of drawings. Moreover, as a text-based tool, Tikz favors portability and collaboration, supported by extensive documentation and an active community.

It is therefore possible to write this kind of code:

```python
print(main_circuit.tikz())
```

Which will generate this tikz code:



```latex
\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{quantikz}
\begin{document}
\begin{quantikz}[row sep=0.5cm, column sep=0.5cm] \\
\lstick{$\ket{q_0}$} & \gate{H} & \targ{} & \gate{X} & \ctrl{2} & \meter{} \\
\lstick{$\ket{q_1}$} & \gate{H} & \ctrl{-1} & \meter{} & \qw & \qw & \\
\lstick{$\ket{q_2}$} & \gate{H} & \qw & \qw & \targ{} & \meter{} & \\
\end{quantikz}
\end{document}
```

Figure 5.1: Return value of the *tikz()* method

6 Discussion

In this report, we initially proposed an abstract representation of quantum circuits. This representation is based on graph theory, with which we then developed a dedicated algorithm for the breadth-first search of the associated graph. This representation allowed us to lay the foundations for the rest of the work, however, it remains subject to improvement. Indeed, the class model must be clearer in order to be modular and allow for more advanced customization of the circuits. Finally, care must be taken to ensure that it does not lose its abstract nature and that it avoids any form of concretization by separating its implementation from that of its output formats (e.g. Tikz). We then discuss the representation of macros and loops in circuits. The macros work correctly, however, the loops need to be implemented. They constitute the most delicate part of this research work which, unfortunately, ends too soon to formalize this concept properly. Due to the infinite possibilities of loops, it will first be necessary to identify a reasonable subset to tackle before gradually expanding it. This functionality would significantly increase the possibilities in terms of quantum circuits, yet it remains unavailable in the current state of the art. Now that the foundations have been laid, a next step would be to integrate the loops into this environment as well as allowing other output formats than the one presented here (tikz).

Bibliography

[1] Arnaud Bodin, Quantum: Un peu de mathématiques pour l'informatique quantique.

[2] Michael A.Nielsen and Isaac L.Chuang, Quantum Computation and Quantum Information.

[3] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron, An Automated Deductive Verification Framework for Circuit-building Quantum Programs.

[4] Irma Avdic, University of Chicago, 2022, Quantum Coin Toss.

[5] Pablo Arrighi, Simon Perdrix, Modèles de Calcul Quantique

[6] Gérard Tisseau and Jacques Duma, 2017, Tikz pour l'impatient.

[7] Qiskit (IBM) open-source project https://github.com/QISKit

[8] Q# (Microsoft) open-source project https://docs.microsoft.com/quantum

[9] Cirq (Google) open-source project https://quantumai.google/cirq

A Appendix

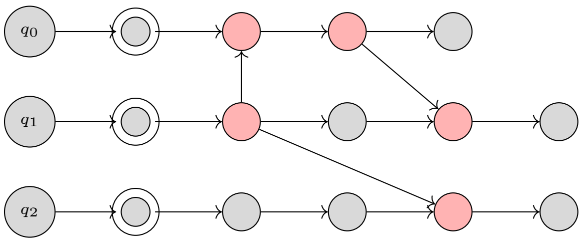


Figure A.1: Step 1

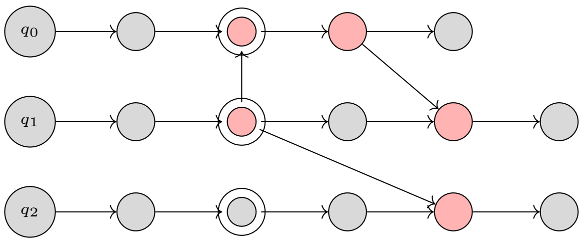


Figure A.2: Step 2

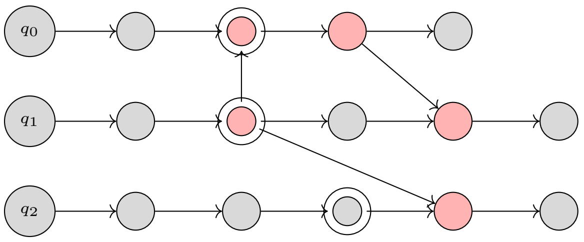


Figure A.3: Step 3

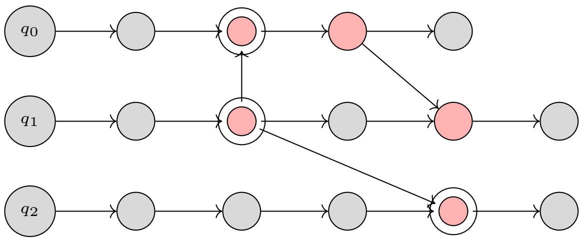


Figure A.4: Step 4

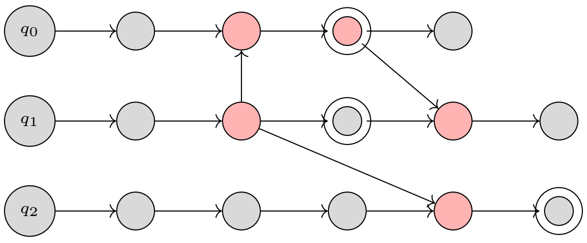


Figure A.5: Step 5

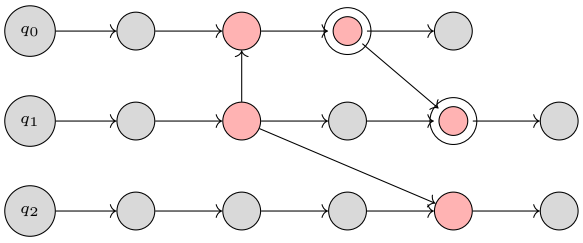


Figure A.6: Step 6

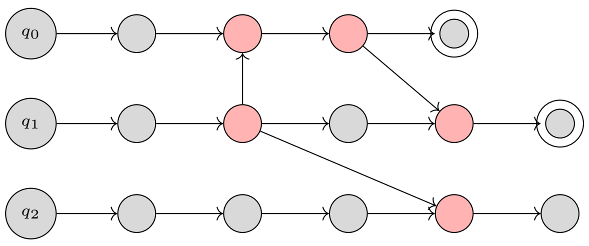


Figure A.7: Step 7