Universitat de Barcelona

Qilimanjaro Quantum Tech

Master in Quantum Science and Technology

Reinforcement Learning based Circuit Compilation via ZX-calculus

Supervisors:
Jordi Riu and Marta P Estarellas

Author:
Jan Nogué Gómez

28 August 2023

Acknowledgments

One can not ever thank enough all the wonderful people who have selflessly offered their help and support while working on such a determinant project as a Master Thesis. This significant milestone could not have been possible without all of you.

To my family and to my soulmate Biel, for always being the stone I could rely on in moments of need, especially the last month of my master thesis.

To my partner, Gemma, for all the love and care that helped me accomplish this task. Having you by my side on this emotional roller coaster was everything I needed to succeed, and I love you for it.

To my brilliant master's fellow students, Jaime, Raúl and Dani. Meeting you was definitely the best part of the master.

Jaime, only you and I know how much you helped me at the beginning of the master's. Who would have told me that what began as a working relationship between students would flourish as one of the most sincere and trustful friendships that I could ever wish for.

To Qilimanjaro for the opportunity, but especially to Ana and Matthias for fruitful discussions and to Ameer for his advice and support.

Above all, the outcome of this work is directly proportional to the amazing supervision that I had from Marta and Jordi. I will be forever thankful for your effort and dedication.

Jordi, you are the best boss that anyone could wish for. I can only hope to become one day the outstanding researcher that you are. I am striving to continue this project with you, the best is yet to come!

Reinforcement Learning based Circuit Compilation via ZX-calculus

Jan Nogué Gómez

Supervised by: Jordi Riu and Marta P Estarellas

Qilimanjaro Quantum Tech, Comtes de Bell-Lloc, 161, 08014 Barcelona

28 August 2023

ZX-calculus is a formalism that can be used for quantum circuit compilation and optimization. We developed a Reinforcement Learning approach for enhanced circuit optimization via the ZX-diagram graph representation of the quantum circuit. The agent is trained using the well-established Proximal Policy Optimization (PPO) algorithm, and it uses Conditional Action Trees to perform Invalid Action Masking to reduce the space of actions available to the agent and speed up its training. Additionally, we also design and implement a Genetic Algorithm for the same task. Both the genetic algorithm and the most widely used ZX-calculus-based library for circuit optimization, the PyZX library, are used to benchmark our RL approach. We find our RL algorithm to be competitive against both approaches, but further exploration is required.

Keywords: quantum computing, quantum circuit optimization, machine learning, deep reinforcement learning, ZX-calculus.

Contents

Jan Nogué Gómez: jan.nogue@qilimanjaro.tech

1 Introduction

Quantum computers have the potential to outperform classical computers and solve various problems in different research fields more efficiently. In computer science, quantum search algorithms like Grover's algorithm can be used to speed up search problems significantly [Gro96]; in physics, quantum computers can be used as a tool for simulating quantum systems [BMK10]; and in chemistry, quantum computers are expected to calculate energy spectra of molecular systems much faster [LWG+10]. Quantum computers operate on quantum bits or qubits, which are the basic units of quantum information. Unlike classical bits, which can only be in one of two states (0 or 1), qubits can be in a superposition of both

states, meaning they can be 0, 1, or anything in between, allowing quantum computers to explore a larger space of possibilities to solve certain problems [RP11].
There are two main models of quantum computation: analog and digital. In the analog model, quantum computers use continuous variables and physical systems to encode and manipulate quantum information. In the digital model, quantum computers use discrete variables and logical operations to encode and manipulate quantum information. In the digital model of quantum computation, we encode the problem to solve as a sequence of quantum gates. Quantum gates are the building blocks of quantum circuits and can be classified into two types: Clifford gates and non-Clifford gates. Clifford gates are a subset of quantum gates that preserve the structure of the Pauli group, which is a set of operators that describe the behaviour of qubits. Non-Clifford gates are any quantum gates that are not in the Clifford subset. Clifford gates are easier to implement and correct than non-Clifford gates, but they are not sufficient for universal quantum computation. Therefore, we need to use both types of gates to achieve the full power of quantum computing [NC10].
Implementing quantum gates is not a trivial task, as quantum systems are very sensitive to noise and errors and any interaction can lose them lose their coherence, i.e. the ability to maintain superposition due to interactions with their environment. This phenomenon is called decoherence, and it limits the time that a quantum system can perform reliable computations [Zur03]. To overcome the challenge of decoherence, we need to find efficient ways to optimize our quantum circuits and reduce their circuit depth, i.e. the length or duration of a quantum circuit. The circuit depth depends on how many layers or steps of gates we need to apply to our qubits. The longer the circuit depth, the more prone our circuit is to errors and noise.
Therefore, we want to minimize the circuit depth as much as possible, while preserving the functionality and correctness of our circuit. This is especially important in the current era of Noisy Intermediate-Scale Quantum (NISQ) devices [Pre18], where we have access to quantum computers with chips ranging from fifty to a few hundred qubits, but with limited coherence time and high error rates.
The motivation of reducing the circuit's depth has spawned a new field of research called quantum circuit optimization, which focuses on finding algorithms for reducing the size and complexity of quantum circuits. Optimizing quantum circuits, in general, is QMA-hard[1], which means that, most likely, there exists no algorithm with polynomial runtime that returns an optimal solution for arbitrary quantum circuits. We focus on quantum circuit optimization using the ZX-calculus, a recently developed graphical language designed to simplify reasoning about quantum systems. We can transform any quantum circuit to an equivalent representation in ZX-calculus called ZX-diagram and use rules of the ZX-calculus to simplify the diagram instead of the circuit. Moreover, the ZX-calculus can be applied to different types of circuits, from NISQ devices to fault-tolerant architectures [dBH20].

2 Background on ZX-calculus

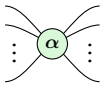
ZX-calculus is a graphical language first introduced by Bob Coecke and Ross Duncan in 2008 [CD11] to represent linear maps between qubits. We can use ZX-calculus to represent any quantum process as a 2-dimensional diagram, where *nodes* and *wires* form

[1]QMA is short for Quantum Merlin Arthur, a complexity class for quantum computers containing all problems from NP.

an undirected graph [2]. We call this graph representation of the quantum process a *ZX-diagram*. The nodes of a ZX-diagram are called *spiders*, which can either be green (Z spider) or red (Xspider). We also have two possible types of wires, Simple and Hadamard wires.[3].

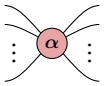
2.1 Spiders

The Z spider is usually represented in green and is a tensor constructed with the composition of the Pauli-Z eigenstates, $|0\rangle$ and $|1\rangle$.



$$:= \underbrace{|0\cdots0\rangle}_{m}\underbrace{\langle 0\cdots0|}_{n} + e^{i\alpha}\underbrace{|1\cdots1\rangle}_{m}\underbrace{\langle 1\cdots1|}_{n}, \tag{1}$$

Where m and n are the inputs/outputs of the spider. The X spider on the other hand is represented in red and is a composition of the X-Pauli eingenstates $|+\rangle$ and $|-\rangle$.



$$:= \underbrace{|+\cdots+\rangle}_{m'}\underbrace{\langle +\cdots+|}_{n'} + e^{i\alpha}\underbrace{|-\cdots-\rangle}_{m'}\underbrace{\langle -\cdots-|}_{n'}, \tag{2}$$

Note that the number of inputs need to be equal to the number of outputs, and α is an angle between 0 and 2π. Examples of special spiders can be found in Appendix A.1.

2.2 Hadamards

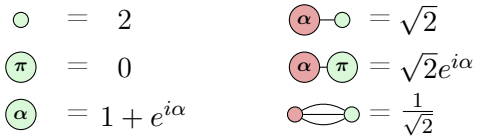
The Hadamard gate is represented as a yellow box or as a *blue wire* in ZX-diagrams

$$= \quad = \quad \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}.$$

While spiders can have any number of inputs or outputs, Hadamards, as single-qubit unitary gates, can only have one input and one output. There are two important identities related to the Hadamard gate in ZX-calculus, which can be found in Appendix A.2.2 and A.2.3.

2.2.1 Scalars

A spider with zero inputs and outputs represents a *scalar*, i.e., a single complex number. Scalars can be moved freely around the ZX-diagram, and they can be combined at any point with the spiders or other generators. Here we present the set of scalars needed to represent any complex number (Derivations in Appendix A.3)



(3)

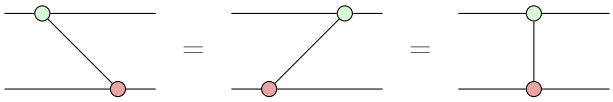
[2] An undirected graph is a mathematical structure that consists of a set of nodes (also called vertices) and a set of edges (also called links or lines) that connect pairs of nodes. The edges do not have any direction, meaning that they can be traversed in either way.

[3] A Hadamard wire is just a simple wire with a Hadamard gate.

In most papers using ZX-calculus, the non-zero scalar factors are usually dropped, for the same reason that we usually work with unnormalised quantum states, it is convenient to simplify calculations. Therefore, two diagrams are considered equal if their associated matrix is equal up to a global scalar.

2.2.2 CNOT gate

The CNOT gate is represented in ZX-diagrams with a green spider (control) connected to a red spider (target). As shown in Eq.4, we can express the CNOT gate either with the green (red) spider acting as input to the red (green) spider. Since both diagrams are equivalent, it is common to express the CNOT with a transverse wire.



(4)

2.3 Rules

In this section, we will present the set of rewrite rules that allows us to transform a ZX-diagram into another. The rules are sound, which means that they all preserve the underlying tensor representation of the diagram, e.g. they have their equivalence in the Hilbert representation. Before explaining the rules that can be applied to a spider, we will present a general set of *meta-rules* regarding ZX-diagrams.

1. Since ZX-diagrams have associated a linear map in a Hilbert space [4], there exist two possible compositions between any ZX-diagrams D_1, D_2:

   (a) **Spatial compositions**: Corresponds to the kronecker product of the tensors associated to D_1, D_2 $(D_1 \otimes D_2)$. Diagrammatically consists of placing the diagram D_2 to the right of D_1.

   (b) **Sequential composition**: Corresponds to the scalar product of the tensors associated to $D_1, D_2 (D_1 \circ D_2)$. Diagrammatically, consists of placing the diagram D_1 on top of D_2 and connecting the outputs of D_1 to the inputs of D_2. This composition is only feasible if D_1 and D_2 have the same number of inputs/outputs.

2. Any of the rewrite rules holds for green and red spiders, or for any orientation of the diagram.

3. *Only connectivity matters.* We can arbitrarily rearrange the spiders in the plane as long as the connections between them and with the inputs/outputs are preserved.[5] We can also bend as much as we want the wires, for any number of inputs and outputs.

4. We can obtain the conjugate of a ZX-diagram by negating all the phases of the spiders that form it. To obtain its transpose, we need to bend the wires to switch inputs for outputs and vice versa.

[4] This is the *standard interpretation* of ZX-diagrams.

[5] This is because the generators in a ZX-diagram represent quantum operations, which are unitary transformations that act on quantum states. Unitary transformations are defined purely in terms of their action on vectors in a Hilbert space and do not depend on any specific representation of the vectors.

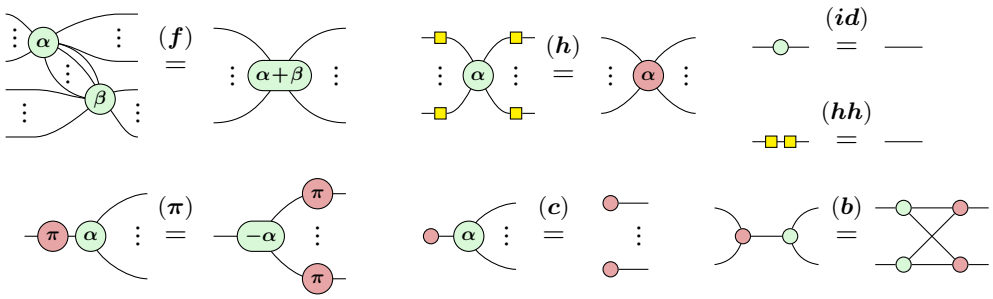


Figure 1: A summary of the ZX-calculus. Note that '...' reads as any integer $n \in \mathbb{Z}$. The letters stand respectively for spider-($\boldsymbol{f}$)usion, ($\boldsymbol{h}$)adamard, ($\boldsymbol{id}$)entity, ($\boldsymbol{hh}$)-cancellation, ($\boldsymbol{\pi}$)-commute, ($\boldsymbol{c}$)opy, and ($\boldsymbol{b}$)ialgebra. Note that these rules are only correct up to non-zero scalar.

The ZX-calculus is summarized in FIG.1. Detailed information about the rules can be found in Appendix A.2 and in [CK17, vdW20].

3 ZX-calculus for circuit optimization - state of the art

In recent years, ZX-calculus has been used for a variety of quantum circuit optimization tasks [KvdW20b, KvdW22, Kru22, SGGGC22]. There are several advantages of using ZX-diagrams for quantum circuit optimization. ZX-diagrams are not bound to the rigid structure of quantum circuits, and their transformation rules are applicable regardless of the dimension of the spider (tensor). Rules like the spider fusion or Hadamard rule are extremely powerful, as they are applicable to any spider at each step of the simplification process, and give rise to transformations that can not always be described by single or two-qubit identities. Moreover, the set of components we need to characterize a ZX-diagram is very small, only green and red spiders are required (with their corresponding angle). The basic procedure for ZX-calculus simplification consists of:

1. Transforming the quantum circuit into an equivalent ZX-diagram.
2. Simplifying the ZX-diagram using the ZX-rules
3. Extracting an equivalent quantum circuit from the ZX-diagram

Note that when we talk about ZX-diagrams for circuit optimization, we use the term *simplification* instead of optimization. It is an important distinction because the ZX-rules focus on decreasing the number of spiders and wires, but not necessarily the gates of the underlying circuit. Using ZX-calculus for circuit optimization is not a trivial task. The rules can be applied in any order, and each rule transforms the diagram, generating/eliminating nodes where other rules could be applied. In some cases, it is even better to not apply any rule even if we could, as the extracted circuit after step 3 may even have more gates than initially. There is no general strategy to tell which sequence of rules applications will yield the maximally optimized underlying circuit. It is also not possible to apply all possible combinations of rules, since the spider fusion and unfusion of a single spider have already theoretically infinite possibilities. Therefore, reducing the number of gates of a quantum circuit with ZX-calculus is an optimization problem with an infinite search space [6] where in most cases we will not be able to find the optimal solution.

[6] We will see that the search space becomes *large* if we restrict the number of rules that can be applied to a ZX-diagram.

Not only that, ZX-rules can modify the ZX-diagram in such a way that the underlying circuit structure is lost. Amongst all the rules that can be applied in a ZX-diagrams, some transform the diagram into a new diagram where circuit extraction is not possible. In this work, we will only focus on the rules that preserve the underlying circuit structure. More information about the circuit procedure can be found in [BMBdF+21].

The most prominent work for quantum circuit optimization using ZX-calculus appeared in 2020 [DKPvdW20] and it is based on a *graph theoretic* approach to simplify ZX-diagrams. Moreover, the authors developed PyZX[7], an open-source Python library to create, manipulate and simplify ZX-diagrams [KvdW20a]. Besides providing a general framework to work with ZX-diagrams, PyZX has been updated with several other optimization algorithms that have been proposed recently [SGGGC22, KvdW20b]. It is precisely one of the algorithms from the PyZX library, `full_reduce`, is the one that will be used to benchmark our results. This algorithm is selected as it is the one with the best tradeoff between computational performance and the quality of the results.

3.1 Graph-based simplifications

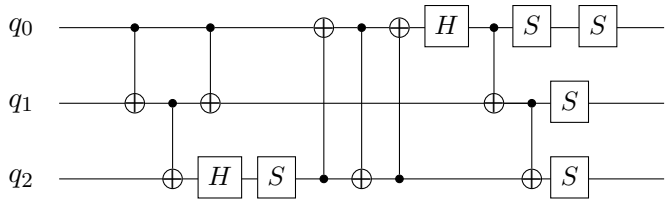
Since ZX-rules can be applied in both directions, current approaches use rules that simplify at least one spider as a way to make sure that the algorithm terminates. The core of most simplification strategies is formed by two rules from graph theory: *local complementation* and *pivoting*. However, these rules can only be applied when the ZX-diagram is in a certain state called *graph-like*. This section presents this new representation of ZX-diagrams, along with the ZX-calculus version of the rewrite rules and finally the simplification algorithm from the PyZX library, `full_reduce`.

3.1.1 Graph-like diagrams

ZX-diagrams are *graph-like* if they satisfy the following conditions:

1. All spiders are Z-spiders (*green*)
2. All connections between spiders are Hadamard wires.
3. There are no parallel Hadamard edges or self-loops.
4. Every input and output is connected to at least one spider and every spider is connected to at most one input or output.

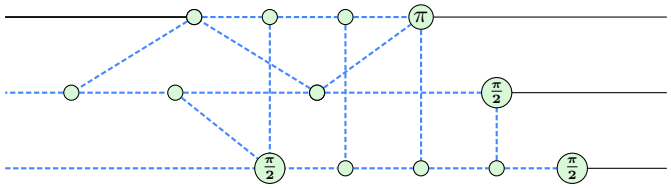
Every ZX-diagram can be transformed into an equivalent *graph-like* form using the ZX-rules presented in section 2.3 (Proof in Lemma 3.2 of [DKPvdW20]). Consider for example the following random circuit:



(5)

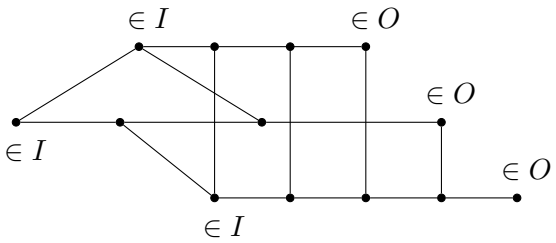
[7] https://github.com/Quantomatic/pyzx

The graph-like equivalent diagram is (intermediate steps in Appendix A.5):



(6)

The advantage of graph-like diagrams is that we capture their structure into an *open graph*. An open graph is defined by the triple set (G, I, E) where $G = (V, E)$ is an undirected graph, with V, E the sets of (vertices, edges). $(I \subseteq V, E \subseteq V)$ are the subsets of (inputs, outputs). We distinguish between *internal vertices* $\{v \in V | v \notin \{I \cup O\}\}$ and *boundary vertices* $\{v \in V | v \in \{I \cup O\}\}$. In the open graph representation of a graph-like diagram, the vertices are the spiders and the edges between spiders are the Hadamard wires. For example, the open-graph for Eq.45 is:



(7)

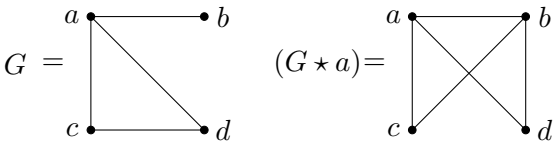
With this graph formulation of ZX-diagrams introduced, we are ready to explain the main rules of graph-theoretic simplification, *local complementation* and *pivoting*.

3.1.2 Local complementation

Consider a graph $G(V, E)$, a vertex $u \in V$ and $N(u) \in V$ the set of neighbours of u. The graph resulting of applying *local complementation* of G according to u, written as $G \star u$, is the graph G except that the neighbors of u, $N(u)$, must satisfy:

1. If two vertices $w, w' \in N(u)$ are connected by an edge in G, then w and w' are not connected by an edge in $G \star u$.
2. If two vertices $w, w' \in N(u)$ are not connected by an edge in G, then w and w' are connected by an edge in $G \star u$.

Therefore, two neighbours of u are connected in $G \star u$ if and only if they are not connected in G. Graphically:



(8)

3.1.3 Pivoting

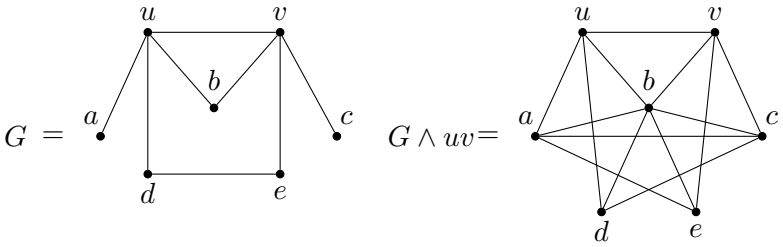
Consider the same graph $G(V, E)$ but now with the two vertices $u, v \in V$. The *pivoting* rule, denoted by $G \wedge uv$, is just the application of three local complementation:

$$G \wedge uv = G \star u \star v \star u. \tag{9}$$

To calculate the connectivity of the resulting graph, we consider the three following sets:

1. $A = N(u) \cap N(v)$, i.e. all the common neighbours of u and v.
2. $B = N(u) \setminus N(v)$, i.e. all the unique neighbours of u.
3. $C = N(v) \setminus N(u)$, i.e. all the unique neighbours of v.

In a pivoted graph $G \wedge uv$, all vertices between these three sets are connected if and only if they were not connected in G. The other connections remain untouched. Consider for example the following graph:

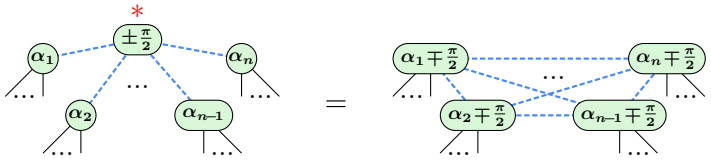


(10)

In Eq.10 we have $A = \{b\}, B = \{a, d\}, C = \{c, e\}$. Therefore, in the pivoted graph all sets are connected between them except d, e because they were connected in G.

3.1.4 Local complementation and pivoting in ZX-diagrams

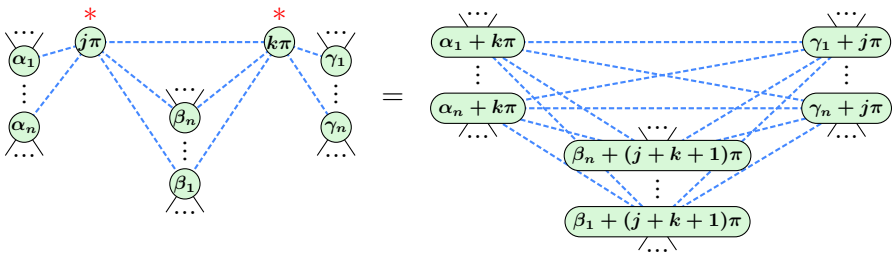
If we have a spider (marked with * in Eq.11) with a phase $\pm\pi/2$ in a graph-like diagram that is *interior*, i.e. is connected to other spiders and not connected to inputs/outputs, we can remove it from the diagram by complementing the neighbourhood of the spider and updating the phases.



(11)

With this variation of the local complementation rule, we can successfully remove *all* interior spiders with phase $\pm\pi/2$.

There also exists a variation of the pivoting rule that can be applied to a pair of *interior* connected spiders with a 0 or π phase in a graph-like diagram. On the right-hand side of Eq.12, we remove the marked spiders at the cost of performing local complementation on the subsets explained in 3.1.3.



(12)

This variation of the pivoting rule allows us to also remove *all* pair of adjacent interior spiders with phase 0 or π. Derivation of these rules can be found in Appendix B3 [DKPvdW20].

3.1.5 Interior Clifford simplification algorithm

Any ZX-diagram can be transformed into the *graph-like* form. However, if the original circuit belongs to the Clifford gate subset, we can significantly simplify the diagram using the variations of the local complementation and pivoting rules explained in Section 3.1.4 [DKPvdW20]. The simplification algorithm looks like the following:

1. Transform the diagram into a graph-like.
2. Apply local complementation to every spider with phase $\pm\pi/2$ and pivoting on every pair of spiders of phase 0 or π as long as possible.
3. Repeat step 2 until no further rule can be applied and the algorithm terminates.

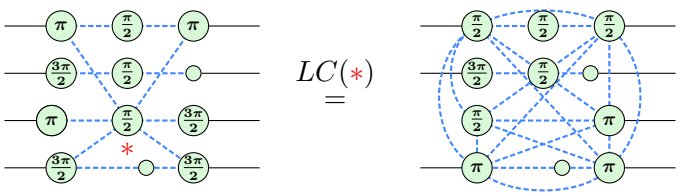
This procedure simplifies all interior spiders of phase $\pm\pi/2$ and pairs of adjacent spiders of phase 0 or π. Only boundary spiders, i.e. spiders connected to an input or an output, remain.

3.2 Simplification strategies and limitations of the PyZX Library

The PyZX library contains several simplification strategies for ZX-diagrams. Regardless of the algorithm, the library always transforms the quantum circuit into its graph-like starting point before applying a set of rules. In our case, we will benchmark our optimizer with the most powerful simplification algorithm from the library PyZX, the algorithm `full_reduce`. The algorithm `full_reduce` applies a variation of the algorithm explained in 3.1.5 with additional rules that allow simplification on boundary spiders. It is important to note that the `full_reduce` prioritizes the termination of the algorithm (reducing the number of spiders as much as possible). To apply the simplification, the `full_reduce` takes as input lists of *non-interacting* vertices for each rule and applies all the rules at once, regardless of the order. There are two main limitations in this algorithm, the first one is that the order of application of the rules matters [SGGGC22] and the second is that the non-interacting vertices reduce the space of solutions [KvdW20a].

3.2.1 The order of application of the rules matter

As mentioned in Section 3, the rules can be applied in any order and each rule modifies the graph so new rules can be applied. This generates a large space of solutions where some sequence of actions will lead to a better optimization of the underlying quantum circuit than other paths. Therefore, to obtain better results than the algorithm `full_reduce`, we need to design a strategy that applies the rules with the goal of optimizing the gates of the underlying quantum circuit. Consider for example the following diagram:



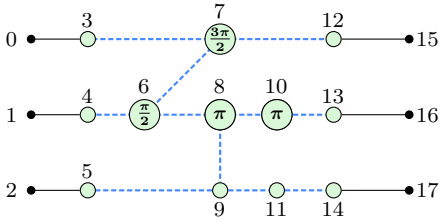
(13)

If we extract a circuit from the LHS of the equation, we obtain **21** gates. On the other hand, the circuit obtained by extracting the RHS of the equation has **35** gates. This can be understood by carefully inspection of the algorithm responsible for circuit extraction

[BMBdF+21]. As a general heuristic, applying local complementation and pivoting is only beneficial, i.e. results in a better simplification if the connectivity of the resulting graph does not increase above a certain threshold [SGGGC22].

3.2.2 Non-interacting vertices

The algorithm `full_reduce` takes as input lists of non-interacting vertices for local complementation and pivoting. Consider for example the following graph-like diagram:



(14)

Here we can apply local complementation to the nodes {6,7} and pivoting to the nodes {(8,9),(8,10),(9,11)} but note that some of these nodes are connected. Take for example that we apply local complementation to node 6: node 6 will disappear and the connections and phases of neighbouring nodes will be updated according to Eq.11. Specifically, spider 7 will update its phase to π ($(3\pi/2 - \pi/2) \bmod 2\pi = \pi$), and spider 8 will update its phase to $\pi/2$. Therefore, applying local complementation to spider 6 changes the graph such that nodes 7 and nodes (8,10) are no longer suitable for local complementation and pivoting, respectively.
Since the `full_reduce` takes a list of spider indices and applies all the rules at once, it needs to select non-interacting nodes because otherwise, it could apply a rule to a node that has been already modified by another rule. PyZX solves this problem by randomly selecting one vertex amongst the interacting nodes. In this case, if it selects vertex 6 as a candidate spider to apply local complementation, it would not select nodes 7, (8,10) as possible nodes for local complementation or pivoting. Once the `full_reduce` receives all the non-interacting nodes, the algorithm applies all the rules until no other rule can be applied.

4 Our-approach: a Reinforcement Learning algorithm

In the previous section, we identified two limitations from the `full_reduce`, the best algorithm from the PyZX library.

1. The algorithm applies all the possible rules in a random order. With this approach, we have no guarantee that the algorithm will reduce the number of gates, and if it does, most likely it will not be the optimal solution.

2. The algorithm takes as input lists of non-interacting vertices for each rule. This approach reduces the solution space by removing possible paths, i.e. feasible sequences of actions, that the algorithm will never explore.

In [SGGGC22] the authors design an *heuristic* algorithm to improve on the `full_reduce` algorithm. They focus on reducing the number of Hadamard wires as much as possible, even if that means leaving remaining actions to apply. They follow the Clifford simplification algorithm from section 3.1.5, but instead of applying local complementation and pivoting

as much as possible, they use *cost functions* for local complementation and pivoting [8] to determine which rules to use. The algorithm stops when any rule has a positive reward, i.e. all possible rules increase the number of Hadamard wires. They manage to outperform the `full_reduce` algorithm, especially for large non-Clifford circuits, at the cost of a high runtime.

As a starting point, we developed a similar strategy based on a genetic algorithm to improve the optimization on the `full_reduce` algorithm. We also managed to outperform the `full_reduce` (results in Section 5.2) but also with a high computational cost.
The computational cost of the genetic algorithm and the fact that we need to run it for each circuit that we want to optimize was a strong enough motivation to develop an approach based on Reinforcement Learning.

4.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) [SB18] is a branch of machine learning where an *Agent* learns through trial and error how to make optimal decisions in an *Environment*. Typically, RL problems are modelled as Markov Decision Processes, which may or may not be stochastic and for which the agent can receive partial or full information about the state of the environment. In our case, the Markovian property is clearly ensured, as the goodness of an action only depends on the current state of the ZX-diagram. Additionally, the environment is completely deterministic (both the local complementation and pivoting actions have non-probabilistic outcomes), and so is the information the agent receives from it. Nonetheless, the complexity of the task lies in the extremely large amount of actions available that the agent can pick from. There are several examples in the literature of successful application of RL algorithms to problems with such characteristics, perhaps the most famous one being Ref.[SHS+17], in which the agent was able to outperform the most advanced algorithms to date and even discover new optimal strategies from which humankind has learned from.
Regardless of the specific properties of the problem to be solved, any RL algorithm requires the initial agent to be trained on the problem that it needs to solve. After the training phase, which may be quite long, the agent can be tested on unseen scenarios as to evaluate its ability to generalize the learnt strategy to similar problems. An overview of the RL training loop can be found in FIG.2. In our case, the full RL training process consists of several episodes, with every episode corresponding to the optimization of a certain quantum circuit via a ZX-diagram. Each episode is divided into several steps T. At each step, the agent receives an *observation*, i.e. information about the state of the environment, s_t. For each observation, the agent outputs an *action*, a_t, which gives a reward r_t, i.e. a measure of how good the performed action is, and a new observation, s_{t+1}.
The ultimate goal of an agent is to learn a policy, i.e. a function that specifies the action that needs to be applied at every step within an episode. The optimal policy is the one that maximizes the cumulative reward received during an episode (Eq.15), which is the sum of partial rewards received after each step.

$$R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r'_t. \tag{15}$$

γ is a discount rate, typically between 0.95 and 0.99. As we will discuss further in the next sections, during the training phase, the agent needs to balance how much it exploits

[8] These cost functions measure how many Hadamard wires appear in the diagram after the application of a rule.

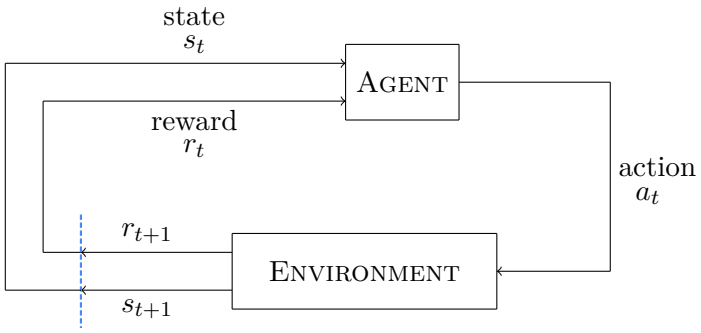


Figure 2: Reinforcement Learning training loop scheme.

a given policy, i.e., the likelihood of choosing the optimal learnt action until that point, versus how much it explores untested actions or that give suboptimal partial rewards (that may lead to a larger cumulative reward overall). We view quantum circuit optimization as a reinforcement learning problem with the following components: The environment corresponds to the graph-like representation of the circuit. From it, we extract a modified adjacency matrix of such graph, which will be fed as observation to the agent. As per the action space, it is the concatenation of available local complementation and pivoting actions as well as an additional stopping action that will terminate the episode. Note that we expect that the agent may eventually need to learn to stop applying such actions if they are not beneficial due to the process of circuit extraction. Finally, the reward will be some function that increases as the number of gates to the circuit decreases. Our approach is very much inspired by the one suggested in [FNML21] with the added benefit that we are using a much simpler action and observation space thanks to using ZX-calculus. Additionally, we add a much more sophisticated action-masking procedure, which facilitates and speeds up the training procedure of the agent. The overview of our implementation can be found in FIG.3. We next provide a detailed description of all the elements of our RL approach.

4.2 Observation space

It is crucial to provide a complete description of the ZX-diagram, so the agent is able to make the best decisions. To do this, we codify the graph-like representation in a modified version of its *adjacency matrix*. The adjacency matrix, A, is a basic representation of the graph of shape $n \times n$, where n is the number of nodes of the graph. If two nodes in the graph $\{n_i, n_j\}$ are connected, then $A_{ij} = 1$, otherwise it is 0 (note that the matrix is symmetric). For our tests, the adjacency matrix is slightly tuned in the following way:

1. The phase of a node is added in the correspondent cell in the diagonal, i.e. For a graph of n nodes, each node i has its phase in A_{ii}. If it is an input or an output, which are phaseless, then $A_{ii} = -1$. Thus, the possible values of the phases are phases are $[-1, 0, \pi/2, \pi, 3\pi/2]$ (normalised to [0,1]). Of course, this is only true for Clifford circuits.

2. The off-diagonal terms are modified to include the type of the wire, if any, connecting two nodes. A_{ij} is 1 for a Hadamard wire, -1 for a normal wire, and 0 if there is no connection.

With this codification of the observation, we are able to encode all the information that the agent needs to know about the ZX-diagram (see FIG.5 for an example of the observation). Note that, since we are using as an observation of the graph-like representation of the

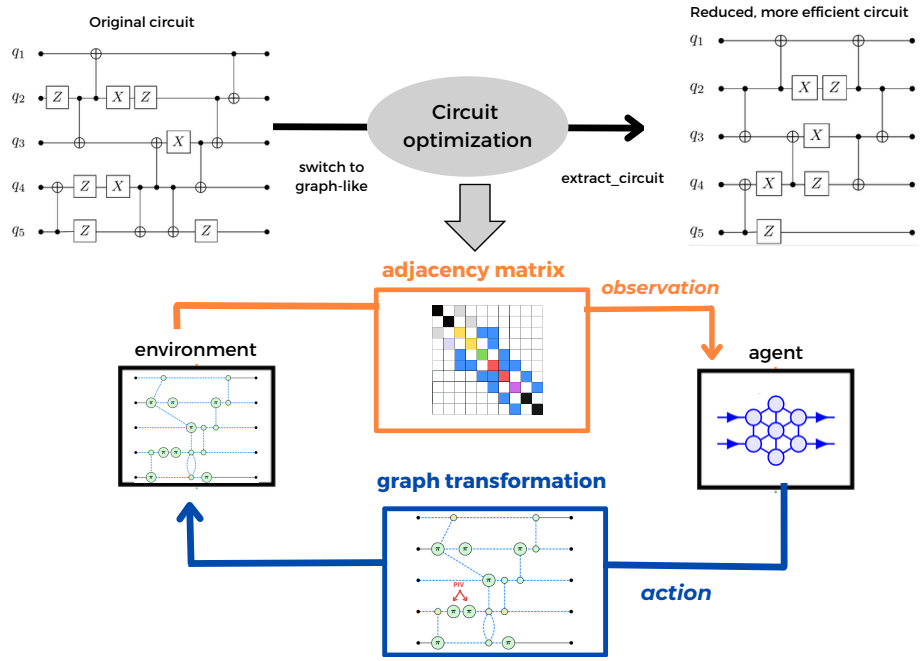


Figure 3: Overview. From a random quantum circuit, we build the equivalent graph-like ZX-diagram. With the codification of the graph-like representation of the circuit (see Section 4.2 for the details), the agent, realized by a neural network, can choose between several graph transformations (see Section 4.3) to generate another, logically equivalent ZX-diagram. This process is repeated multiple times. When the episode is done, we use the method `extract_circuit` from the PyZX library to obtain a circuit from the graph-like representation.

circuit, the size of the graph will vary depending on the topology of the initial circuit. Since the size of the observation must be constant during the training, we determine an upper bound in the number of nodes required to represent any circuit of a given dimension (for a circuit with 5 qubits and 25 gates, we arbitrarily choose 40) and pad the adjacency matrix with zeros until such upper bound is reached. Therefore, the observation for any circuit of 5x25 will be a 40x40 matrix. We also randomize the indices of the nodes for each observation to avoid overfitting.
A possible drawback of this approach is that the matrix tends to be very sparse, containing some irrelevant information, that can make the training difficult (as it adds noise to it).

4.3 Action space

The action space is the set of possible actions that the agent can take in a given environment. In our case, we define a 3-dimensional *multi-discrete* action space, i.e. we need 3 elements to fully characterize an action. The first element represents the action to apply to the circuit (whether to do nothing, apply local complementation or apply pivoting) and the second and third elements represent the nodes to which the action is applied:

1. First element of the action space: Between [0,2]. 0 means STOP, i.e. to not apply an action, 1 means LC, i.e. apply local complementation, and 2 means PIV, i.e. apply pivoting.

2. Second element of the action space: values can range between [0, N], with N the number of nodes in the adjacency matrix. It represents the first node to which the action is applied.

3. Third element of the action space: values can range between [0, N]. It represents the second node to which the action is applied.

Even though each action needs to have all three components, some of those turn out to be uninformative. For example, only one node is needed to characterize the action *local complementation* and no nodes are required for the STOP action. To avoid inconsistencies, we will force the second and third elements of an LC action to be the same and to fix these same elements to 0 for the stopping action.
Along these lines, the agent can never apply LC or PIV to some of the nodes within the graph. This is the case for input and output nodes, and their nearest neighbours, for which it is straightforward to see that they do not fulfil the required conditions detailed in Section 3.1.4. For the circuit of 5 qubits and 25 gates, these unchangeable nodes correspond to 20 out of the 40 total nodes. Therefore, we can truncate the first and second elements of the action space to have values between [0,19].

The second issue of such action space definition is that the agent can choose to apply actions to invalid nodes. Consider for example the ZX-diagram in Eq.14. For this diagram, the only feasible actions are

$$Actions = \{[0,0,0],[1,6,6],[1,7,7],[2,8,9],[2,8,10],[2,9,11]\}.$$

which correspond to combinations of the action space elements that represent the only possible actions for the ZX-diagram in Eq.14, i.e. local complementation (LC) to node 6 and 7 or pivoting (PIV) to nodes (8,9),(8,10),(9,11) and the stopping action. However, the available actions that the agent can generate combining the elements of the multi-discrete action space (A) are [A_0,A_1,A_2] such that

$$A_0 \in \{0,1,2\}.$$
$$A_1 \in \{0,1,...,19\}.$$
$$A_2 \in \{0,1,...,19\}.$$

Hence, with this action space configuration, the agent can sample invalid actions such as {[1,6,7], [1,10,10], [2,8,8]} or even actions that do not represent any identifiable action, like [1,15,19]. To avoid this, one can choose to penalize unfeasible actions through the reward function, but it is a much more efficient strategy to only allow the agent to select between actions that fulfil the constraints of the problem. This process is called *invalid action-masking* and its inner workings are described in section 4.3.2. Moreover, we note that our valid actions have a *hierarchical* structure, i.e. if the sampled from A_0 is 1 (LC), then the agent should only have as options {6,7} to sample for A_1 and {6} or {7} to sample for A_2 (depending on the sampled node from A_1). Therefore, we need to implement a strategy that forces the agent to sample between the feasible actions but takes into account the hierarchical structure of the valid actions. This can be achieved with Conditional Action Trees [BO21].

4.3.1 Conditional Action Trees

A Conditional Action Trees (CAT) is a data structure that represents the action space of an agent as a tree of nodes and edges. Within the tree, a node corresponds to an action and an edge corresponds to a condition or a dependency between actions. The root node represents a dummy node that is used to add conditions on the first layer of the hierarchical

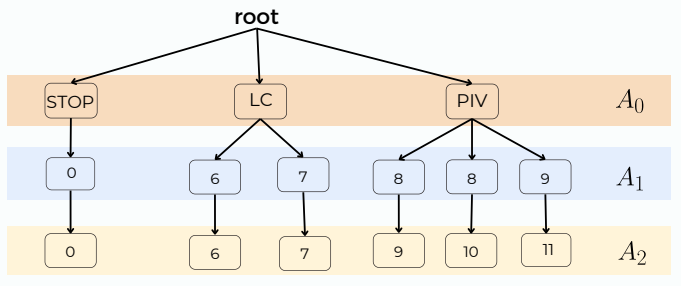


Figure 4: Conditional Action tree with the possible actions for the ZX-diagram of Eq.14.

action space (A_0), and the leaf nodes represent the final actions (A_2 in our case), see FIG.4 for an example of a CAT.
For a given state, we will construct the CAT and use it to guide our agent through the action selection process via Invalid Action Masking (IAM).

4.3.2 Invalid Action Masking (IAM)

Invalid action masking (IAM) [HO22] is a technique used to prevent agents from sampling actions that are invalid in a particular state. In Actor-Critic (AC) methods (see Section 4.5), we use a Neural Network (NN) to interpolate the probability to select each action given an input state, i.e. the policy $\pi_\theta(a|s)$. Here θ represents the parameters of the neural network. During training, the agent learns the optimal policy, i.e., it optimizes the parameters of the network such that actions that are expected to lead to larger episodic rewards for a given state are selected with the highest probability. Specifically, the NN interpolating the policy outputs an unnormalised distribution of action probabilities over all possible actions in the action space, regardless of whether they are feasible or not. Each value l_i assigned to an action i is called a *logit*. To obtain the policy $\pi_\theta(a|s)$, we need to turn these logits into a probability distribution. This is achieved by applying a *softmax* activation function to each set of logits (Eq.16).

$$\pi(a_i|s) = \frac{e^{l_i}}{\sum_j^N e^{l_j}}. \tag{16}$$

Note that, for our multi-discrete action space, we will generate 3 sets of logits corresponding to each component within the multi-discrete space (see FIG.5). Each of these logits needs to be normalized using a softmax activation separately so that the action probabilities for each component of the action space add up to one, but we want this normalization to be in accordance with the conditional action tree of the state. To achieve this, the 3 components of an action will be sampled iteratively in the following way: First, we will sample from the probability distribution of the first component and obtain either LC, PIV or STOP as output. Depending on the obtained value, all logits of the next action component that can not be selected, as per the restrictions determined by the CAT, will be converted to -∞. By doing that, once we apply the softmax function to this set of logits, these nodes will have a probability of 0 to be selected, thus ensuring that all sampled actions are feasible. As a simple example, if we consider again the CAT in FIG.4, a possible outcome of the sampling procedure could be the following: We start sampling from the probability distribution of A_0. Assuming that LC is the sampled node, the logits $\{0, 1, ..., 5, 8, ..., 19\}$ of A_1 are then masked to a value $-\infty$ such that only $\{6, 7\}$ can be selected. Finally, if node 6 is sampled as A_1, all logits of A_2 will be masked except for logit 6.

4.4 Reward function

The reward is a measure of how good the action taken by the agent is. Since in our approach we want to optimize the number of final gates of the circuit, we decided this metric to be the reward. Before and after applying an action, we use the previously mentioned built-in method of the PyZX library, `extract_circuit`, to obtain the number of gates. The reward is then calculated as the normalised difference between the number of gates before and after applying the action. The normalization coefficient is chosen to be a constant value that depends on the circuit dimension (number of qubits and initial gates) such that all episodic rewards are between -1 and 1, as it leads to more stable training.

4.5 Proximal Policy Optimization (PPO)

Following the approach described in [FNML21] we select the Proximal Policy Optimization [SWD+17] as the RL algorithm used to train the agent. The PPO belongs to the class of Actor-Critic (AC2) algorithms, which are characterized by the fact the agent architecture consists of two separate function interpolators. The first one, called *actor*, is responsible for the interpolation of the policy function, which we have thoroughly discussed above. The second one called *critic*, is in charge of assessing the expected benefit from the current policy. In the case of the PPO, this is done through the estimation of the value function $V(s)$, i.e., the cumulative reward that the agent expects to obtain from state s given its learnt policy. The actor and the critic work together to improve the agent's performance by learning from the rewards and penalties received from the environment. In our work, we use neural networks to interpolate both functions, and their parameters are optimized simultaneously using gradient-based optimization techniques, more specifically, the *Adam* optimizer.

Another important characteristic of the PPO is that the training is done *on-policy*, meaning that any update on the current policy is only dependent on observations, rewards and actions generated by this same present policy. This differs from *off-policy* methods that store experiences generated with previous policies in a memory, for them to be used in future updates.
It is important to point out that policy-based methods are very sensitive to large policy updates, which can lead to instability during training or even *catastrophic forgetting*. To solve this problem, PPO includes several parameters that can be tuned in order to restrict policy changes. Examples of such are, maximum gradient norms, and clipping parameters. The former scales all parameter gradients to a maximum value, and the latter clips the allowed change of parameters of the network such that the ratio between the new policy probabilities and the previous ones are within a range $[1-\epsilon, 1+\epsilon]$, hence removing the incentive from the current policy to go too far from the old one.

4.5.1 Loss function

As mentioned above, during the training of the agent, the parameters (weights and biases) of both the actor and critic network are updated using gradient-based optimization. In this section, we present the loss function Eq.(17) used by PPO, from which the gradients are computed.

$$L(\theta) = L^{Policy}(\theta) + \alpha L^{Value}(\theta) - \beta L^{Entropy}(\theta). \tag{17}$$

The loss function is composed of three terms:

1. The policy loss, $L^{Policy}(\theta)$, is computed as the product of the ratio of probability change between policies and the expected return of the actions. With this, the

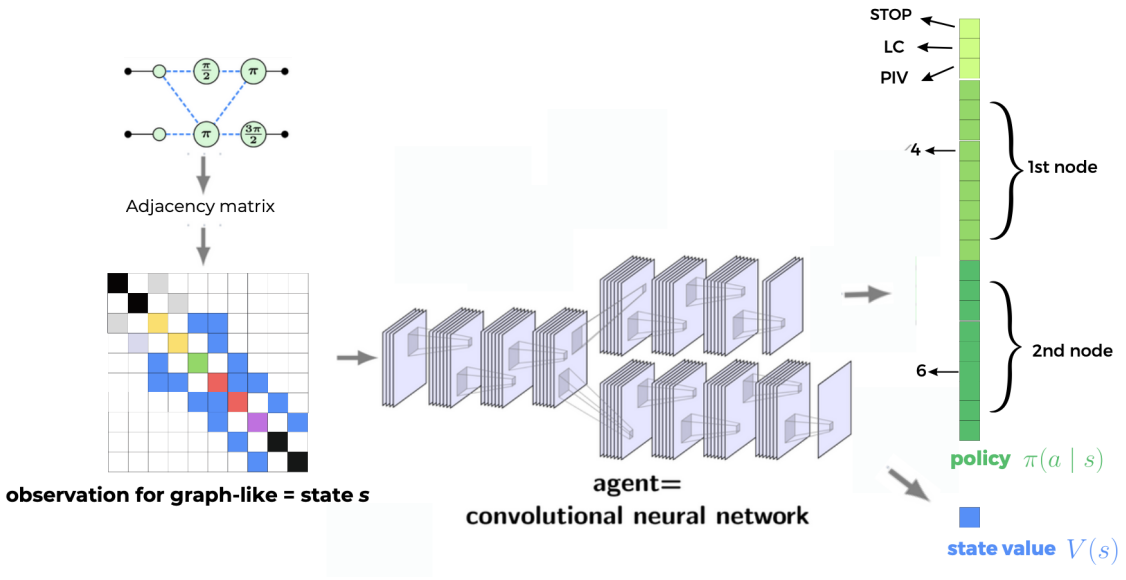


Figure 5: Deep Convolutional network architecture of our RL agent. In the observation, each possible type of node or edge is displayed in a different colour for visualisation purposes.

policy loss decreases if the new policy increases the probability of selecting actions with higher returns with respect to the previous one, and likewise, it reduces the probability of selecting non-beneficial actions.

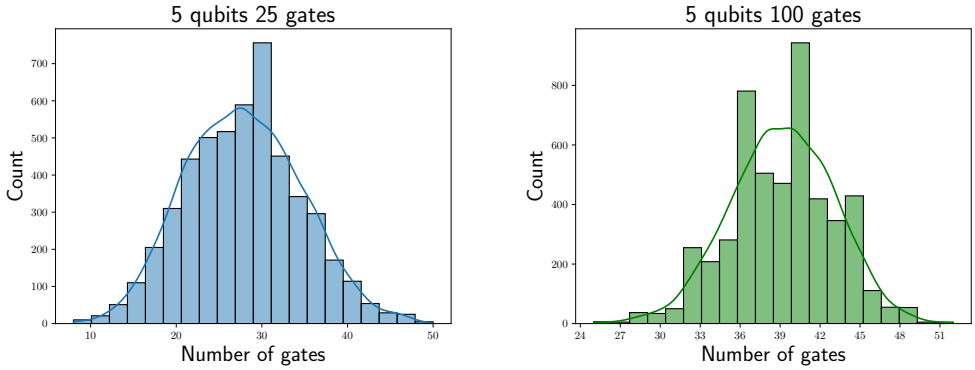
2. The value loss, $L^{Value}(\theta)$, essentially corresponds to the mean-squared error between the critic network's prediction and the actual return obtained by the agent. Hence, it is a measure of how well the critic can estimate the expected return given a state. This term can be slightly modified to include clipping restrictions if desired, so as to achieve a more stable training.

3. The entropy loss, $L^{Entropy}(\theta)$, is equal to the entropy of the logits of the policy (actor) function. Note that the entropy has an opposite sign in the loss function, meaning that the total loss function decreases as the entropy in the logits increases. Although it seems counterintuitive, this implies that during training, this term directs the agent towards parameter configurations that increase the uncertainty of the policy.

Exploration and exploitation are two conflicting goals in reinforcement learning. Exploration means trying new actions that may lead to better outcomes in the future, while exploitation means sticking to the best-known actions that maximize the immediate rewards. A good reinforcement learning agent should be able to explore enough to discover new and better actions, but also exploit enough to avoid wasting time and resources on tuning the networks to properly interpolate suboptimal actions. For the case of PPO, finding the right balance between both is done through the tuning of the hyperparameters, α and β, in the loss function. The higher the entropy coefficient, the more the agent will explore the action space and vice versa.

4.5.2 Agent Architecture

We provide a graphic representation of the agent architecture, both for the actor and critic networks, in FIG 5. At each step in an episode, the agent receives a complete description of the state s, i.e. the ZX-diagram in graph-like formalism, through the modified adjacency



(a) 5 qubits and 25 gates.

(b) 5 qubits and 100 gates.

Figure 6: Results of applying the `full_reduce` algorithm to optimize the gates of 5000 random Clifford circuits. a) Histogram of 5 qubits and 25 gates random circuits with the corresponding fit. The mean value and peak of the fitted distribution is 27.8 gates. b.) Histogram of 5 qubits and 100 gates random circuits with the corresponding fit. The mean value and peak of the fitted distribution is 39.2 gates.

matrix. This observation is then passed through several convolutional layers[9]. After that, we flatten the output and convert it into a one-dimensional vector that is then used as input to two separate Feed Forward Neural Networks: One for the policy, $\pi(a|s)$, and the other for the value function. Every action, i.e. graph transformation, is mapped uniquely into a single policy output neuron.

5 Experiments and results

In this section, we present the most relevant results obtained by the RL approach in contrast to the `full_reduce` algorithm. Additionally, we also implement a genetic algorithm to assess the capabilities and limitations of our approach against an even more sophisticated alternative.

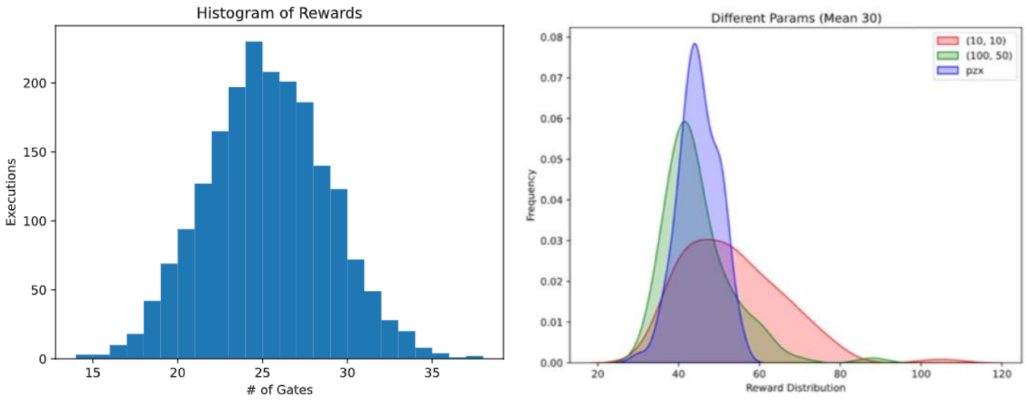
5.1 Results with the full_reduce algorithm

The `full_reduce` is particularly well-suited for Clifford circuits [DKPvdW20]. In FIG.6b we can see that the algorithm `full_reduce` performs well for large circuits, i.e. for 500 random circuits of 5 qubits and **100** gates, it obtains a mean of **39** final gates.
On the other hand, we find that the algorithm has difficulties in fine-tuning optimizations (see FIG.6a), i.e. for 500 random circuits of 5 qubits and **25** gates, it obtains a mean of **27.8** final gates (the algorithm even increases slightly the number of final gates).
Even though we would like to test our approach against the most challenging case of 5 qubits and 100 gates, due to a limited amount of computational resources, we will focus on the use case of Clifford circuits of 5 qubits and 25 gates.

5.2 Genetic algorithm

As a first step, we develop a genetic algorithm in order to check whether the results obtained by `full_reduce` algorithm are competitive. These type of algorithms have already

[9]Convolutional Neural Networks (CNN) are especially well suited for capturing spatial correlations within the input data. Since we codified our observation as a matrix where each cell resembles a pixel in an image, we expect that CNNs may perform slightly better than feed-forward neural networks.



(a) Results of the genetics algorithm for Clifford circuits of 5 qubits and 25 gates.

(b) Results of the genetics algorithm for Clifford circuits of 5 qubits and 100 gates.

Figure 7: Results. a) Histogram of final gates of the genetic (100 individuals and 50 iterations) algorithm for random Clifford circuits of 5 qubits and 25 gates. b) Distribution of final gates for the genetic algorithm: In red, 10 individuals and 10 iterations. In green 100 individuals and 50 iterations. In blue are the results of the `full_reduce`.

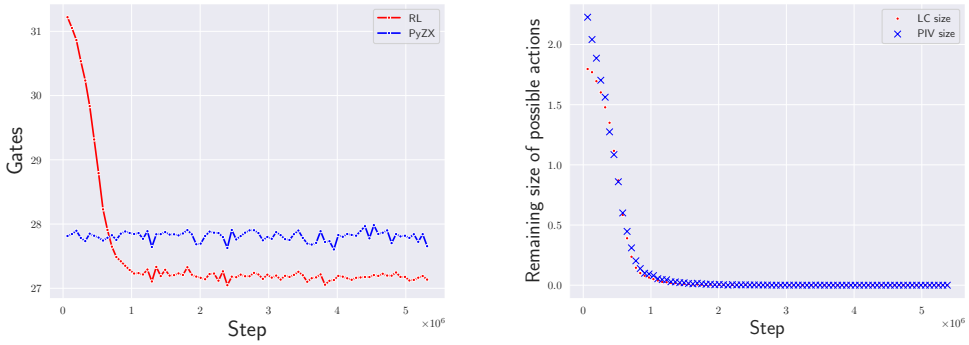
been explored for similar tasks in conjunction with ZX-Calculus [SGGGC22]. The algorithm details are described in Appendix B. Most likely our design and implementation can be more carefully refined, but we do not explore it in detail, as it is not the goal of our project.
Even without this refinement, the results of the Genetic algorithm are significantly better than the PyZX (see the results in FIG.7), hence proving that the `full_reduce` algorithm can be improved even for Clifford circuits.
We tested our algorithm with 100 individuals and 50 iterations of random Clifford circuits of 5 qubits and 25 gates (see FIG.7a) and obtained a mean value of final gates of **24.77**, whereas the `full_reduce` algorithm obtained a mean of **27.8** gates. Thus, the genetic algorithm already surpasses by approximately three gates the `full_reduce` for the case of a Clifford circuit of 5 qubits and 25 gates.

An interesting side result from the genetic algorithm applied to Clifford Circuits of 5 qubits and 25 gates is that, in the majority of the cases, the algorithm obtained a **better optimization** without **performing all the available actions.** This behaviour will allow us to understand whether the RL algorithm is performing correctly.

In FIG.7b we see that for the case of 5 qubits and 100 gates, the distribution of final gates of the genetic algorithm (100 individuals, 50 iterations, in green), with a mean of 42.25 gates, is slightly more shifted to fewer gates than the gates distribution by the `full_reduce`(in blue), with a mean 49.8 gates. Therefore, the genetic also outperforms the `full_reduce` for this type of circuit. However, as a drawback, our implemented genetic algorithm is not scalable and the computational cost for each circuit is significantly higher, typically a couple of orders of magnitude larger with respect to the `full_reduce`.

5.3 Reinforcement Learning Algorithm

We believe the aforementioned results clearly motivate the usage of Reinforcement Learning. This is because even though the training is computationally demanding, once finished, the agent is able to optimize unseen circuits very rapidly. Here lies the ultimate goal of



(a) Number of gates during the training.

(b) Remaining actions during training.

Figure 8: Results of the RL. a) Evolution of the final number of gates during training vs. the `full_reduce` from the library PyZX. b.) Evolution of the remaining local complementation and pivoting size during training.

our approach: training an agent that is nearly as fast as the `full_reduce` algorithm and obtains results of similar quality to the Genetic algorithm, for any Clifford circuit of 5 qubits and 25 gates, without individually training the agent for each circuit.

We trained our RL algorithm to optimize the gates of random Clifford circuits of 5 qubits and 25 gates using the graph representation of ZX-diagrams. In FIG.8a we compare the performance of our agent with the `full_reduce` during training. Note that our agent initially interpolates policies that obtain an average of 31 gates and slowly decreases until stabilizing around **27.2** gates, already improving the `full_reduce`, that oscillates around **27.8**. This result, although promising, is still far from the one obtained by the genetic algorithm of 24.77 gates. To understand the reason behind this performance, we refer to the results shown in FIG.8b. In this figure, we plot the remaining pivoting and local complementation actions at the end of the episode. A value of 0 implies that the agent has performed all the actions available. We can clearly see that the agent is not able to navigate policies for which not all actions are performed. This may indicate that the STOP action is not explored enough, and thus a better balance between exploration and exploitation needs to be found. Nonetheless, the agent is able to learn a more "smart" order of application of actions with respect to the one used by `full_reduce`.

Reaching the best optimization of the agent is by no means a trivial task. Besides tuning all the hyperparameters, there are infinite CNN structures to test. Finding the best combination that suits our problem is the real challenge in RL. The presented results are the best ones obtained amongst all the explored combinations of hyperparameter configurations and agent architectures, that are not discussed for brevity.
A likely reason for which our architecture is not able to improve the results further is due to the fact that we are using CNN layers. These convolutional layers are very successful in capturing 2-D or 3-D correlations between data, hence why they were used in the original paper, which dealt with gate-based circuit optimization. However, that is not the case with our diagrams, which can be high-dimensional graphs. In this regard, Graph Neural Networks [WPC+21] may be a much more natural candidate to be used both for the actor and critic networks. We leave the exploration of this type of architecture as future work.

6 Conclusions and future work

In this work, we provide an overview of the state-of-the-art quantum circuit optimization of Clifford circuits using ZX-calculus, with a specific focus on the `full_reduce` algorithm from the PyZX library. We discuss the two main limitations of the algorithm and develop two strategies to try to overcome them, although only one of those is studied in detail.

The first approach combines a genetic algorithm with the graph-formalism of ZX-diagrams. It accomplishes a 0.8% gate reduction (in mean) for the Clifford circuits of 5 qubits and 25 gates and a 61% gate reduction for the Cliffords circuits of 5 qubits and 100 gates, in both cases clearly outperforming the `full_reduce`. Nonetheless, our unrefined implementation of the genetic algorithm is computationally expensive, requiring 1 minute on average to optimize circuits of 5 qubits and 25 gates and 20 minutes on average to optimize circuits of 5 qubits and 100 gates. Since this approach does not allow transferring the knowledge from past optimizations to new circuits, we are not confident in its scalability.

As a second approach, and the main focus of the project, we develop a RL-based algorithm using the well-established PPO algorithm to try and find the right balance between the computational performance of the algorithm and the quality of the obtained optimization. The RL agent uses Conditional Action Trees and Invalid Action Masking to improve the speed of training. Both Convolutional and Feed Forward layers are used for the actor and critic networks used by the agent to act on the environment. We also design a modified adjacency matrix that stores all the relevant information that the agent needs to act, through either local complementation (LC) or pivoting (PIV) on the environment.

For Clifford circuits of 5 qubits and 25 gates, the agent actually increases the gates by 8.8 %, outperforming the `full_reduce` algorithm by 2.4% gates reduction, but far from the results obtained by the genetic approach. By plotting the remaining pivoting and local complementation actions available at the end of each episode, we see that our agent is applying all possible actions without ever stopping. We hypothesize that the reason why the agent is not reaching better optimizations is that the STOP action is not sufficiently explored, hinting at the need for increasing exploration during training. In regard to the computational performance of the agent, its training lasts for over 2 days (on a very small server), whilst each circuit optimization for the genetic algorithm requires a minute on average. The advantage of the RL approach is that, after training, the agent optimizes circuits in less than a second. Therefore, the RL approach is preferable for continuous use. Additionally, and although it is not explored in this project, another potential advantage of RL approaches is that they may allow the transferring of knowledge acquired during the training of smaller circuits to bigger ones thus significantly speeding up the training.

Finally, we also argue the fact that convolutional layers are not well-suited to capture correlations for ZX graph-like diagrams, as they are high dimensional graphs. We suggest the use of Graph Neural Network (GNN) for the actor and critic networks.

Bibliography

[BMBdF+21] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. There and back again: A circuit extraction tale. *Quantum*, 5:421, mar 2021.

[BMK10] Katherine L. Brown, William J. Munro, and Vivien M. Kendon. Using quantum computers for quantum simulation. *Entropy*, 12(11):2268–2307, nov 2010.

[BO21] Christopher Bamford and Alvaro Ovalle. Generalising discrete action spaces with conditional action trees, 2021.

[CD11] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, apr 2011.

[CK17] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[dBH20] Niel de Beaudrap and Dominic Horsman. The ZX calculus is a language for surface code lattice surgery. *Quantum*, 4:218, jan 2020.

[DKPvdW20] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum*, 4:279, jun 2020.

[FNML21] Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. Quantum circuit optimization with deep reinforcement learning, 2021.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[HO22] Shengyi Huang and Santiago Ontañ ón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, may 2022.

[Kru22] Ryan Krueger. Vanishing 2-qubit gates with non-simplification zx-rules, 2022.

[KvdW20a] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. *Electronic Proceedings in Theoretical Computer Science*, 318:229–241, may 2020.

[KvdW20b] Aleks Kissinger and John van de Wetering. Reducing the number of non-clifford gates in quantum circuits. *Physical Review A*, 102(2), aug 2020.

[KvdW22] Aleks Kissinger and John van de Wetering. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4):044001, jul 2022.

[LWG+10] B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, A. Aspuru-Guzik, and A. G. White. Towards quantum chemistry on a quantum computer. *Nature Chemistry*, 2(2):106–111, jan 2010.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[Pre18] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, aug 2018.

[RP11] Eleanor Rieffel and Wolfgang Polak. *Quantum Computing: A Gentle Introduction*. The MIT Press, 1st edition, 2011.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[SGGGC22] Korbinian Staudacher, Tobias Guggemos, Wolfgang Gehrke, and Sophia Grundner-Culemann. Reducing 2-qubit gate count for zx-calculus based quantum circuit optimization. In *Quantum Processing and Languages (QPL22)*, pages 1–17, 2022.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[vdW20] John van de Wetering. Zx-calculus for the working quantum computer scientist, 2020.

[WPC+21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, jan 2021.

[Zur03] Wojciech Hubert Zurek. Decoherence, einselection, and the quantum origins of the classical. *Reviews of Modern Physics*, 75(3):715–775, may 2003.

A ZX-calculus

A.1 Special spiders

From the definitions of *green* and *red* spiders, we can see that the linear map of Z and X spiders with only one input and one output corresponds to the well-known Z and X phase gates,

$$R_Z(\alpha) = \quad = \quad |0\rangle\langle 0| + e^{i\alpha}\,|1\rangle\langle 1| \quad = \quad \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix}. \tag{18}$$

$$R_X(\alpha) = \quad = \quad |+\rangle\langle +| + e^{i\alpha}\,|-\rangle\langle -| \quad = \quad \frac{1}{2}\begin{pmatrix} 1+e^{i\alpha} & 1-e^{i\alpha} \\ 1-e^{i\alpha} & 1+e^{i\alpha} \end{pmatrix}. \tag{19}$$

For $\alpha = \pi$ we get the the Pauli-Z and Pauli-X gate. When $\alpha = 0$, we represent the Z and X spiders as

$$\begin{aligned} &= \quad |0\cdots 0\rangle\langle 0\cdots 0| + |1\cdots 1\rangle\langle 1\cdots 1|\,. \\ &= \quad |+\cdots +\rangle\langle +\cdots +| + |-\cdots -\rangle\langle -\cdots -|\,. \end{aligned} \tag{20}$$

Note that, in addition to an empty wire, both the 1-input, 1-output Z spider and the 1-input, 1-output X spider are also the identity.

$$= \quad = \tag{21}$$

A.2 ZX-calculus rules

A.2.1 Spider fusion

The most fundamental rule of ZX-calculus is spider fusion. This rule allows us to fuse two spiders of the same color connected by one or more wires into a single spider. When the two spiders are connected, their phases are added together and the wires connecting the spiders disappear. The adding of the two phases basically generalises that two rotations of the Bloch sphere in the same direction add together, and thus it is assumed to be modulo 2π.

$$= \tag{22}$$

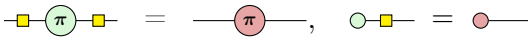
A.2.2 Identity rules (id, hh)

We obtain the identity from the 1-input, 1-output spider. In the ZX-diagrams, the identity is represented by an empty piece of wire, and it is used for concatenation between spiders (***id***). The second equality is due to the fact that the Hadamard gate is self-inverse, $HH = \mathbb{I}$ and hence two boxes in a row cancel out (***hh***).

$$\begin{array}{c|c} (\boldsymbol{id}) & (\boldsymbol{hh}) \\ = & = \end{array} \tag{23}$$

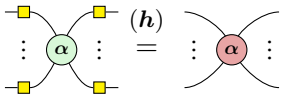
A.2.3 Hadamard rule (h)

This rule relates to the fact that conjugating the Z gate with two Hadamard gates results in an X gate: $HZH = X$. This identity allows us to change the colors of spiders when it is surrounded by Hadamards. Note that the number of inputs or outputs can also be zero. Here we present two examples,



(24)

This colour-changing identity generalises to the following:

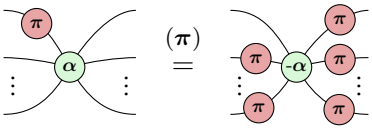


(25)

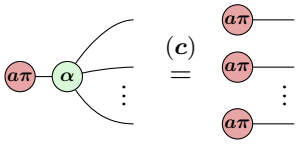
Where $\alpha \in \mathbb{R}$ can be any phase. Here we have also assumed that the spider has the same number of inputs/outputs, otherwise, the identity does not make sense.

A.2.4 The copy rule (c) and π-commutation (π)

The copy and π-commutation (Eq.38) allow us to commute red (green) spiders with phase $(0, \pi)$ through an arbitrary phase α green (red) spider.



(26)

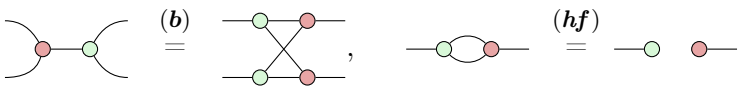


(27)

Where a is a boolean variable and it can be $a = \{0, 1\}$. It is important to note that the *copy rules* only hold when the spider being copied has 0 input and 1 output. Further details on these rules can be found in Appendix A.4.

A.2.5 The bialgebra and hopf-rule

The other known rules to prove circuit identities are the bialgebra rule (***b***) and the hopf-rule (***hf***).

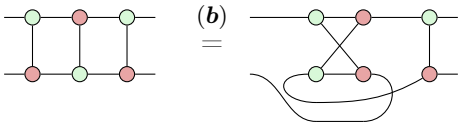


(28)

In Appendix A.2.6 we present an example of the ZX-rules application to prove that three consecutive CNOT gates make a SWAP gate.

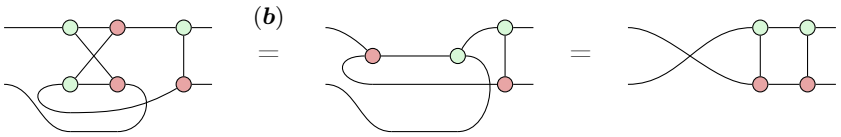
A.2.6 Examples

Using the *hopf rule* and the *bialgebra rule*, we can show that three consecutive CNOT gates make a SWAP gate [vdW20]. We start by rearranging the CNOTs so we can apply the bialgebra rule:



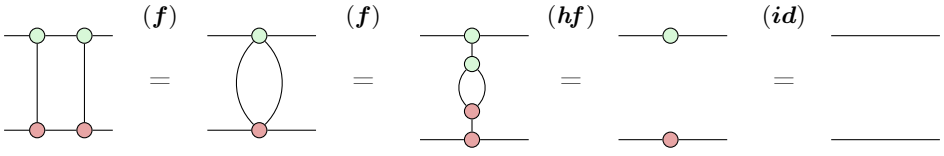
(29)

Now we apply the bialgebra rule to the first two CNOTs, and reorder the diagram



(30)

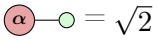
To finish the proof, we need to show that two CNOTs applied consecutively cancel each other. To do so, we fuse the spiders, apply the Hopf rule and apply the identity rule to conclude our proof



(31)

A.3 Scalars in ZX-diagrams

Here we will present the derivations of the non-trivial scalars explained in Section 2.2.1.



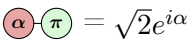
(32)

Proof. The result of this ZX-diagram can be calculated explicitly as follows

$$(\langle +| + e^{i\alpha}\langle -|) \circ \mathbb{I} \circ (|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}[(\langle 0| + \langle 1|) + e^{i\alpha}(\langle 0| - \langle 1|)] \circ (|0\rangle + |1\rangle) = \frac{2}{\sqrt{2}} = \sqrt{2}.$$

□

If instead, we have a phase-π Z-spider, we obtain



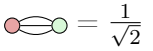
(33)

Proof. Similarly, the result of this ZX-diagram yields

$$(\langle +| + e^{i\alpha}\langle -|) \circ \mathbb{I} \circ (|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}[(\langle 0| + \langle 1|) + e^{i\alpha}(\langle 0| - \langle 1|)] \circ (|0\rangle - |1\rangle) =$$
$$= \frac{2}{\sqrt{2}}e^{i\alpha} = \sqrt{2}e^{i\alpha}.$$

□

Finally, we will see that closed loops in ZX-diagrams are also scalars (Eq.34),



(34)

Proof. We can also calculate the analytic expression of this last ZX-diagram

$$[\langle + + +| + \langle - - -|] \circ \mathbb{I} \circ [|000\rangle + |111\rangle] =$$
$$= [\langle + + +|000\rangle + \langle + + +|111\rangle + \langle - - -|000\rangle + \langle - - -|111\rangle] = \frac{2}{2\sqrt{2}} = \frac{1}{\sqrt{2}}.$$

Where we have used that

$$|\pm\pm\pm\rangle = \frac{1}{2\sqrt{2}}\left[|000\rangle \pm |111\rangle + e^{i\alpha}\sum_{|x\rangle\neq|000\rangle,|111\rangle}|x\rangle\right].$$

And $e^{i\alpha}$ is just a phase factor. □

Since we can represent any complex number as a multiplication of $\frac{1}{\sqrt{2}}$, $\sqrt{2}e^{i\alpha}$ and $1+e^{i\alpha}$[10], these three ZX-diagrams form a complete basis.

A.4 The copy rule and π-commutation

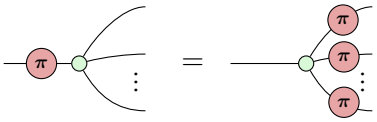
The copy rule arises from the interaction of the Pauli Z and X gates and their respective eigenbasis with spiders. Let's consider an X gate and a single input, no phase, and an arbitrary number of outputs Z-spider. If we apply the X gate to the input of the Z-spider, we have

$$|0\cdots0\rangle\langle0|\,X + |1\cdots1\rangle\langle1|\,X = |0\cdots0\rangle\langle1| + |1\cdots1\rangle\langle0|\,.$$

Hence, we see that the X gate changes $|0\rangle$ and $|1\rangle$. This last expression is equivalent to the same Z-spider we considered before, with X gates applied to each of the outputs

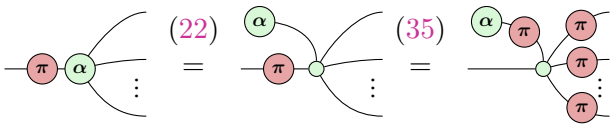
$$|0\cdots0\rangle\langle1| + |1\cdots1\rangle\langle0| = (X\otimes\cdots\otimes X)\,|1\cdots1\rangle\langle1| + (X\otimes\cdots\otimes X)\,|0\cdots0\rangle\langle0|\,.$$

In terms of diagrams, the *π-copy rule* is:



(35)

From this diagram, we can also see that the *π-copy rule* holds for any number of inputs and outputs. If instead, we have a non-zero phase Z-spider, we need to apply the spider fusion rule to unfuse them before copying

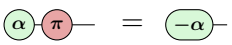


(36)

We can further simplify this last expression by applying the X gate to the non-zero phase 1-input Z-spider, $|0\rangle + e^{i\alpha}\,|1\rangle$.

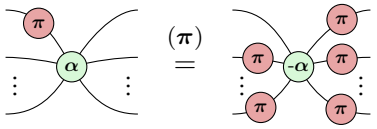
$$X(|0\rangle + e^{i\alpha}\,|1\rangle) = X\,|0\rangle + e^{i\alpha}X\,|1\rangle = |1\rangle + e^{i\alpha}\,|0\rangle = e^{i\alpha}(|0\rangle + e^{-i\alpha}\,|1\rangle).$$

In terms of diagrams, this equality is (up to a global phase):



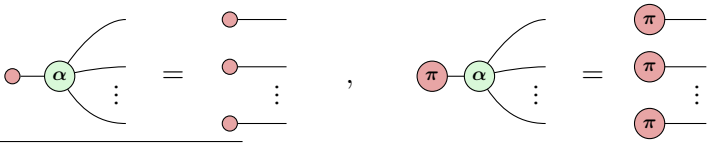
(37)

Hence, the most generic case for the copy rule is (which also holds for inversed colours)



(38)

If we now consider trying to copy the eigenstates of the Z gate through a 1-input, m-outputs, nonzero phase Z-spider, we find that



(39)

[10]Let z be a complex number with $|z| < 2$. We can choose α such that $|z| = |1+e^{i\alpha}|$. Also, for some β we also have that $\frac{z}{1+e^{i\alpha}} = e^{i\beta}$. Thus, $z = (1+e^{i\alpha})e^{i\beta} = \frac{1}{\sqrt{2}}(1+e^{i\alpha})\sqrt{2}e^{i\beta}$. For complex numbers with $|z| > 2$, we can just first rescale it by multiplying by $1/\sqrt{2}$.

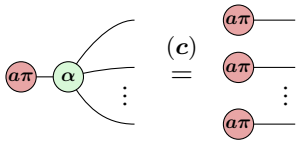
Proof.

$$(\sqrt{2}\,\langle 0|)(|0\rangle\langle 0...0| + e^{i\alpha}\,|1\rangle\langle 1...1|) \propto \langle 0...0| = \underbrace{\langle 0| \otimes ... \otimes \langle 0|}_{m}.$$

$$(\sqrt{2}\,\langle 1|)(|0\rangle\langle 0...0| + e^{i\alpha}\,|1\rangle\langle 1...1|) \propto \langle 1...1| = \underbrace{\langle 1| \otimes ... \otimes \langle 1|}_{m}.$$

□

These rules are denominated as the *state-copy rules* and can be unified using a boolean variable a that takes values $a = \{0, 1\}$. It is important to note that the *state-copy rules* only hold when the spider being copied has phase 0 or π.

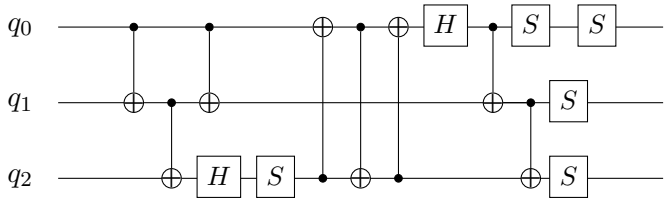


(40)

As usual, these rules hold with colours flipped and regardless of the orientation of the wires, as we can always use caps and cups to deform our diagram to our will.

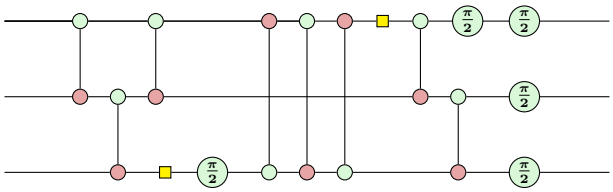
A.5 Graph-like example

Consider the following random circuit:



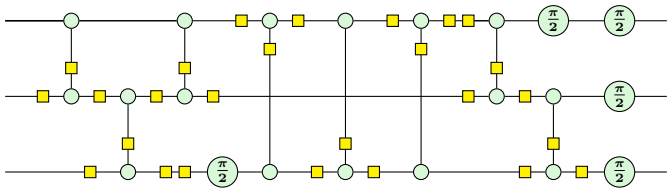
(41)

Which corresponds to the following ZX-diagram:



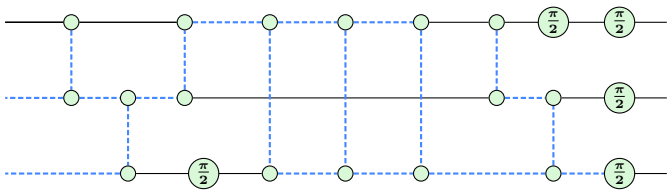
(42)

First, we turn all the spiders into green spiders using the Hadamard rule (***h***)



(43)

Now we use the identity rule (***hh***) to remove two Hadamards in a row and use the blue wire representation for the wires with Hadamards.



(44)

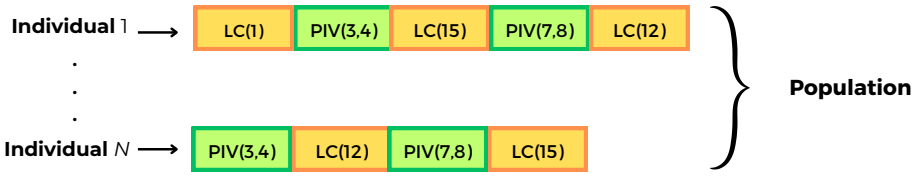
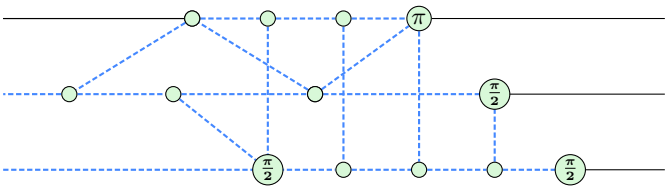


Figure 9: Scheme of the genetic algorithm. The individuals are sequences of genes, where each gene is a possible action.

We obtain a maximally fused graph using spider fusion ($\boldsymbol{f}$) (note that all edges between spiders ara Hadamard wires)



(45)

In this case, we do not have any loops or parallel Hadamard wires so we are done (we can always remove them using the rules explained in Lemma 3.2 of [DKPvdW20].

B Genetic algoritm

A genetic algorithm is a biologically-inspired algorithm that works by creating a population of possible solutions, called individuals, and evaluating how good they are at solving the problem, using a fitness function. The fitness function is a way of measuring how close an individual is to the optimal solution.
The genetic algorithm then selects some of the best individuals to create new individuals, called offspring, by combining parts of their solutions, called genes. This process is called crossover. The genetic algorithm also introduces some random changes to some of the individuals, called mutations. This process helps to explore new solutions and avoid getting stuck at a local minimum. The algorithm repeats this process of evaluation, selection, crossover, and mutation until it finds a good enough solution, or reaches a maximum number of iterations.
As mentioned, the individuals are chains of genes. In our case, each gene can be type pivoting (to nodes N_i, N_j) or local complementation (to node N_i). See FIG.9.

B.1 Mutation and offspring rules

The first-generation individual evolves through mutation and offspring rules. In our case, we have defined the following rules, see FIG.10:

1. **Cut**: The individual from generation $i+1$ is generated by cutting a random number of genes from the individual from generation i.

2. **Enlargement**: The inverse action of Cut. The individual from generation $i+1$ is generated by adding a random number of genes from the individual from generation i. We always make sure that the added genes are in fact feasible actions for our individual.

3. **Mutation**: The last individual's gene changes for another possible gene.

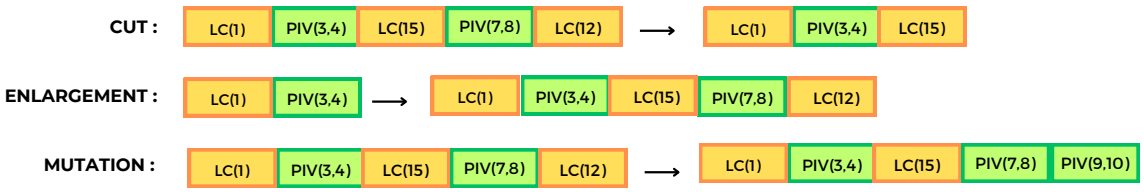


Figure 10: Mutation and offspring rules of our genetic algorithm.

B.2 Fitness function

The fitness function allows us to compare between individuals and keep the best one. For our problem, we defined the fitness function to be the difference between the initial and final number of gates. With this fitness function, we will be able to isolate the individuals whose combination of genes reaches the highest reduction of gates.