POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Mater's Degree Thesis

Design of a customizable simulation infrastructure for noisy quantum circuits

Politecnico
di Torino

Supervisors:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Prof. Giovanna Turvani

Candidate:
Simone Pont

December 2021

Summary

In the last years, quantum computing has grown in popularity becoming one of the most promising computational paradigms. Its advantage is the increase of the **density of information**, achieved by using **superposition**, and a potential speed-up in the convergence to an optimal solution through **entanglement**. To fully explore the possibilities that quantum algorithms can offer and to reliably estimate the performance of a real quantum computer in a practical scenario, a **classical quantum circuit simulator** is needed.
In this thesis, a classical simulator infrastructure, capable to simulate **noisy quantum circuits**, was developed. This infrastructure should be placed at the end of a toolchain dedicated to the creation of quantum circuits which is under development at the VLSI Lab of Politecnico di Torino. The simulator has been developed using the C++ language, which allows greater degrees of optimization in terms of computational cost. The simulation methodology is based on the Schrödinger formalism: quantum states are represented by their wave functions and unitary operators act on them. The main issue is that the **complexity grows exponentially** with the number of qubits n: according to Dirac notation, states are described by using vectors of 2^n complex amplitudes, and operators by using $2^n \times 2^n$ unitary matrices.
Two different representations of quantum states and quantum gates have been explored and implemented: the **Array based** and the **Decision Diagram (DD) based**. In the former, quantum states and gates are described using mono and bi-dimensional arrays. In the latter, they are described using graphs that try to optimize the representation by **exploiting the redundancies** of the elements of vectors and matrices. In both cases, the advantages and disadvantages in terms of performance and memory occupation have been investigated.
The implemented simulator can work with two different approaches to model the noise. The first one is the most employed in scientific literature and it is based on the use of some super-operators $\{K_1,K_2,\cdots,K_m\}$, called **Kraus operators**, acting on the state. Considering that the non-ideality phenomena described by these operators are usually applied after every gate, the computational cost and the memory occupation of the overall simulation increase. The second methodology is based on a **compact model** developed by the VLSI Lab research group at Politecnico di Torino. In this approach, the effects of **relaxation and decoherence** are described respectively by a single vector (relaxation vector) and matrix (decoherence matrix), minimizing the needed resources. They are directly applied to the density matrix describing the quantum state through the element-by-element product.
The simulator is based on a **customizable modular structure** that leads to a

versatile simulation. Configuration files can be used to change the settings of the simulator, such as the representation or the noise model that have to be used. Moreover, the circuit to be simulated can be defined by hand or by exploiting the OpenQASM 2.0 language.
In the first two chapters, the key concepts about **quantum theory** and **quantum computation** are introduced. In particular, the problems related to the classical simulation of quantum circuits are highlighted.
In the third chapter, the **structure of the implemented simulator** is reported, underling the interactions between its different modules. Then, the approach used to implement the noiseless simulation is described.
A more detailed analysis of the **Array-based** simulation and the classes used to implement it is reported in the fourth chapter. This is the standard representation to describe quantum states and gates, based on the *Eigen* libraries. Then, **Decision Diagrams** are introduced in the fifth chapter, focusing on their possible usage in the quantum circuit simulation. Their compact structure, obtained by exploiting the redundancies inside the states and gates, can limit the memory occupation. However, their performance is greatly affected by the characteristics of the considered circuit.
In the sixth chapter, the **noisy simulation** is considered and the two implemented noise models are described. The introduction of non-ideality phenomena changes the simulation approach: quantum systems must be associated with **mixed states** that are described by using **density matrices** instead of state vectors. Moreover, the noise effects must be considered during the simulation, increasing its complexity.
The seventh chapter contains the analysis of the **obtained results** concerning the simulation time and the memory usage. The different configurations are compared to identify the most suitable one for every need.
Finally, the **possible improvements** and the **future prospects** are presented in the last chapter. In particular, the possibility to integrate new and optimized modules to improve the developed infrastructure is highlighted.

Table of contents

1 State Of The Art **1**
1.1 Quantum Theory . . . 1
1.2 Quantum Computation . . . 3

2 Quantum Circuit Simulation **6**
2.1 Classical Simulation Of Quantum Circuits . . . 6

3 Simulator Structure **10**
3.1 Simulation Approach . . . 10
3.2 Simulator Behavior . . . 11
3.3 Code Structure . . . 13
3.4 *Circuit* Class . . . 16
3.4.1 Condensed-Gate Simulation . . . 21
3.5 *CircOpenQASM* Class . . . 24
3.6 Simulator Internal Library . . . 30

4 Array Representation **32**
4.1 C++ Basic Implementation . . . 33
4.1.1 *ArrayStateVector* Class . . . 33
4.1.2 *OperatorArray* Class . . . 34
4.2 Array-Based Simulation . . . 37

5 Decision-Diagram Representation **40**
5.1 Decision Diagram Theory . . . 40
5.1.1 Implemented Structure . . . 49
5.2 C++ Basic Implementation . . . 52
5.2.1 *DDStateVector* Class . . . 52
5.2.2 *DDSqMatrix* Class . . . 57
5.2.3 *OperatorDD* Class . . . 64
5.3 Improvements . . . 68

6 Noisy Simulation **76**
6.1 Noise In Simulation . . . 76
6.1.1 Standard Noise Model . . . 79
6.1.2 Compact Noise Model . . . 80
6.2 C++ Implementation . . . 82
6.2.1 *ArrayDensityMatrix* and *DDDensityMatrix* Classes . . . 83

6.2.2 *NoiseModel* Class 84
6.2.3 Noise Model Library 86
6.3 Noisy Simulator Behavior 87

7 Obtained Results 92
7.1 Tools And Benchmarks 92
7.1.1 Massif Log File 94
7.2 Simulator Validation 95
7.3 Ideal Simulation Results 97
7.3.1 “Condensed Gate” Simulation Results 107
7.4 Noisy Simulation Results 108
7.4.1 Standard Noise Model Results 109
7.4.2 Compact Noise Model Results 113
7.5 Overall Comparison 113

8 Conclusion 121
8.1 Summary 121
8.2 Possible Improvements And Future Prospects 122

Bibliography 124

List of tables

3.1 Table containing all the gates of the internal library. 31
7.1 Comparison between the quantum state probability distribution affected by numerical errors obtained with the implemented simulator and the correct one deriving from *Qiskit*. 97
7.2 Obtained results from the simulation of generic quantum circuits considering noiseless simulation using state vectors. 98
7.3 Obtained results from the simulation of generic quantum circuits considering the noiseless simulation using density matrices. 103
7.4 Obtained results from the noiseless simulation of generic quantum circuits considering the "condensed gate" optimization. 108
7.5 Obtained results from the simulation of generic quantum circuits affected by Bit-Phase Flip error. 109
7.6 Obtained results from the simulation of generic quantum circuits affected by Depolarizing errors. 112
7.7 Obtained results from the simulation of generic quantum circuits affected by decoherence and relaxation errors described by the compact model analyzed in Section 6.1.2. 113

List of figures

1.1 Example of the graphical representation of a generic quantum circuit. 4
2.1 Comparison between a circuit with “separated” and “merged” gates. . 8
3.1 General behavior of the simulator. 13
3.2 General structure of the simulator’s code. The classes are reported in the rectangles and the implemented library in the hexagon. The dependencies between them are indicated by arrows. 14
3.3 Directory tree containing the simulator files. 16
3.4 Algorithm used to reorder a quantum state applying SWAPs operations to it. 19
3.5 Algorithm used for the basic noiseless simulation of a generic quantum circuit. 20
3.6 Example of the overlap between two gates after having applied the needed SWAPs. 22
3.7 Algorithm used for the optimized condensed-gate simulation of a generic quantum circuit. 23
3.8 General algorithm used to read an OpenQASM 2.0 file and generate the related circuit. 28
4.1 Algorithm used to apply the operator to a generic reordered state vector. 36
4.2 Detail of the noiseless Array-based simulator structure. 38
4.3 Simulator behavior during the noiseless Array-based simulation. . . . 39
5.1 Basic structure of a Decision Diagram representing a generic monodimensional array. 41
5.2 Comparison between the basic structure of a DD and the one that exploits the redundancies inside the elements of the represented monodimensional array. 42
5.3 Comparison between the structure of a DD with and without edge weights. 43
5.4 Optimal structure of a Decision Diagram representing a generic monodimensional array. 44
5.5 Decision Diagram used to represent a three qubits state vector. 46
5.6 Decision Diagram representation of a three qubits state vector with highlighted the path used to access the amplitude $\frac{-1}{\sqrt{2}}$. 47
5.7 Decision Diagram used to represent the two qubits matrix associated to the CH gate. 48

5.8 Decision Diagram representation of the matrix associated to the CH gate with highlighted the path used to access the element $\frac{1}{\sqrt{2}}$. 49
5.9 Decision Diagram used by the simulator to represent a three qubits state vector. 50
5.10 Comparison between the DD representation of the matrix associated to the CH gate used by the simulator and the one with the minimum number of nodes. 51
5.11 General algorithm for the creation of a DD state vector. 54
5.12 Algorithm for the creation of the bottom level of the DD state vector. 55
5.13 Algorithm for the creation of the intermediate levels of the DD state vector. 56
5.14 Comparison between the tree structure of the *DDStateVec* and the *DDSqMatrix*. 58
5.15 Comparison between the tree structure of the *DDSqMatrix* without and with the termination nodes. 59
5.16 General algorithm for the creation of a DD representing a generic matrix. 60
5.17 Algorithm for the creation of the bottom level of a matrix's DD. . . . 61
5.18 Algorithm to fill a bottom level node accessing the correct 2x2 submatrix. 62
5.19 Algorithm for the creation of the upper levels of a matrix's DD. . . . 63
5.20 DD representation of two matrices and the related tensor product. . . 66
5.21 Algorithm for the calculation of a tensor product between two matrices represented using Decision Diagrams. 67
5.22 General algorithm for the calculation of the recursive row-column product between two matrices represented using Decision Diagrams. . 70
5.23 Algorithm for the calculation of the intermediate levels of the recursive row-column product between two matrices represented using Decision Diagrams. 72
5.24 Algorithm for the calculation of the last level of the recursive row-column product between two matrices represented using Decision Diagrams. 73
5.25 Algorithm used to set (reset) a certain qubit of the quantum state. . 75
6.1 General algorithm used to simulate a noisy circuit. 89
6.2 Simulator behavior during the noisy Array-based simulation. 90
6.3 Detail of the noisy Array-based simulator structure. 91
7.1 Example of the graphical representation of the memory usage trend during the program execution. On the horizontal and vertical axes are reported respectively the progression of the program and the allocated memory. 94

7.2 Simple quantum circuit used to analyze the effects of numerical errors inside the simulator. 96
7.3 Simulation time of the QFT and IQFT circuits depending on the considered number of qubits in the case of noiseless simulation with state vectors. 99
7.4 Memory occupation of the QFT and IQFT circuits depending on the considered number of qubits in the case of noiseless simulation with state vectors. 100
7.5 Trend of memory usage in the noiseless simulation of the *iqft_8* circuit using the Array-based state vectors. 102
7.6 Trend of memory usage in the noiseless simulation of the *iqft_8* circuit using the DD-based state vectors. 102
7.7 Comparison between the simulation time of the QFT and IQFT noiseless circuits using state vectors and density matrices. 104
7.8 Comparison between the memory occupation of the QFT and IQFT noiseless circuits using state vectors and density matrices. 105
7.9 Trend of the memory usage in the noiseless simulation of the *adder_medium* circuit using the Array-based density matrices. 106
7.10 Trend of the memory usage in the noiseless simulation of the *adder_medium* circuit using the DD-based density matrices. 107
7.11 Trend of the memory usage in the noisy simulation of the *qft_9* circuit using the Array-based representation. 110
7.12 Trend of the memory usage in the noisy simulation of the *qft_9* circuit using the DD-based representation. 111
7.13 Comparison between the simulation time of small circuits considering different possible configurations. 115
7.14 Comparison between the simulation time of intermediate circuits considering different possible configurations. 116
7.15 Comparison between the average memory occupation of small circuits considering different possible configurations. 118
7.16 Comparison between the average memory occupation of intermediate circuits considering different possible configurations. 119

Chapter 1

State Of The Art

In this chapter, the basic quantum computing theory is introduced. The mathematical knowledge of complex numbers and complex vector spaces is required to fully understand its content. The initial chapters of [1] and [2] can be used to acquire this knowledge.

1.1 Quantum Theory

In the last years, quantum computing has grown in popularity becoming one of the most promising computational paradigm. Indeed, it can improve the density of information and the computational power of a certain system. Its possible applications are multiple and include several fields, such as machine learning, physics simulations, algorithm optimizations and quantum chemistry [3, 4, 5].
Classical digital computer operates using bits as basic unit of information. In contrast, quantum computers are based on the use of qubits, i.e. two-level quantum-mechanical systems that belongs to $\mathbb{C}^2$ complex Hilbert vector spaces. While a bit can assume only two possible states (0 and 1), a qubit can be in a coherent superposition of both. Considering the Dirac notation [2] and using the orthogonal states $|0\rangle = [1\ 0]^\mathsf{T}$ and $|1\rangle = [0\ 1]^\mathsf{T}$ as basis, the pure state $|\phi\rangle$ of a qubit is a linear combination of them:

$$|\phi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{1.1}$$

where $\alpha \in \mathbb{C}$ and $\beta \in \mathbb{C}$ are known as complex amplitudes (or probability amplitudes) and satisfy $\alpha\alpha^* + \beta\beta^* = 1$. A graphical representation of the state of a single qubit can be obtained by using the Bloch sphere [2].
If n qubits are combined in an unique system, the state $|\varphi\rangle$ of the latter is a weighted superposition of the 2^n orthogonal basis vectors $|x_i\rangle$ of the corresponding Hilbert

space [2]:

$$|\varphi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |x_i\rangle, \text{ with } \alpha_i \in \mathbb{C} \text{ and } \sum_{i=0}^{2^n-1} \alpha_i \alpha_i^* = 1. \tag{1.2}$$

A generic basis $\{|x_i\rangle\}$ for the n-qubit systems can be expressed as $\{|b_{n-1}b_{n-2}\cdots b_1b_0\rangle\}$ where b_{n-1},b_{n-2}, $\cdots$,b_1,$b_0 \in \{0, 1\}$. For example, in the case of a two-qubit system ($n = 2$), a valid basis is $\{|00\rangle,|01\rangle,|10\rangle,|11\rangle\}$. It important to underline that the proposed basis is not the only one that can be used. In an n-qubit system, any basis of the $\mathbb{C}^n$ space can be employed. However, the considered one is the most used in the quantum computing notation and it will be adopted from now on.

In a multi-qubit system, some superpositions can be represented more compactly as tensor products of smaller states. An example is the state $\frac{1}{\sqrt{2}}(|01\rangle + |11\rangle)$ that can be expressed as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |1\rangle$. However, not all the possible states can be described in this way because the qubits influence each other. The states that cannot be decomposed to a tensor product between smaller states are called "entangled" (for more details refer to chapter 3.1.3 of [2]). Superposition and entanglement are the two main reasons for the increase of information density in a quantum system compared to a classical one [6].

Another important aspect of quantum systems that must be analyzed is their behavior during measurements. The measurement operation modifies the quantum state of the system and collapses it to a specific basis state with a certain probability (for more details refer to chapter 3.1.4 of [2]). The result of the measurement is not deterministic and depends on the amplitudes associated with every state. In particular, measuring a single qubit in the state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ changes its state to one of the two bases adopted for the measurement. Considering to work with the standard basis ($|0\rangle$ and $|1\rangle$) the measurement outcome is $|0\rangle$ with probability $\alpha\alpha^*$ and $|1\rangle$ with probability $\beta\beta^*$. In both cases, the final state is collapsed and the superposition is lost. The measurement in a multi-qubit system can be implemented by simply measuring one by one all the qubits of the system. In this situation, the entanglement is important because the outcome of a certain qubit is affected by the measurement of another one in an entangled state. For example, supposing to have the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ and measure the first qubit to be 0 (1), the measurement of the second qubit will definitely be 0 (1).

1.2 Quantum Computation

Now that the main concepts of the basic quantum theory have been introduced, it is important to describe how they can be applied to quantum computation. A generic n-qubit system is usually called *quantum register* and is used to store information about the evolution of the state through a quantum circuit. As in classical logical computation, the circuits are composed of different gates (called quantum gates or operators) that modify the quantum state. These quantum gates are reversible and described by unitary matrices that define their effects on the quantum state [7]. Considering that the biggest part of the boolean logic gates is not reversible, the basic quantum gates have generally a different behavior compared to them. However, all the classical gates can be emulated by using quantum gates. The effect of a generic quantum gate described by the matrix U on a certain quantum register in the state $|\varphi\rangle$ is obtained by doing the product between the matrix U and the vector $|\varphi\rangle$ (containing the probability amplitudes of the state), as reported in Equation (1.3):

$$|\varphi'\rangle = U|\varphi\rangle, \tag{1.3}$$

where $|\varphi'\rangle$ is the vector describing the state after the application of the gate.
The basic quantum gates operate with a limited amount of qubits, usually less than three, the most important ones are reported in Table 3.1. However, bigger gates acting on multiple qubits simultaneously can be generated by combining simple gates. This can be done using the Kronecker product (or tensor product) between the matrices representing the different gates that have to be combined. Multiple consecutive quantum gates compose a quantum circuit and collections of quantum circuits are used to execute more sophisticated quantum algorithms. In every algorithm, some final or intermediate measurements are then performed to retrieve the needed information from the quantum register. Moreover, multiple quantum algorithms can be concatenated and combined (also with the classical computation) to solve complex problems.
To summarize, in the abstract level quantum computation is composed by:

- One or multiple quantum registers used to store the information about the quantum system;

- Multiple quantum gates that compose the quantum circuits in charge to execute the quantum algorithm;
- Multiple measurements used to extract the needed information from the registers.

Figure 1.1 reports the commonly used graphical representation of a simple quantum circuit.

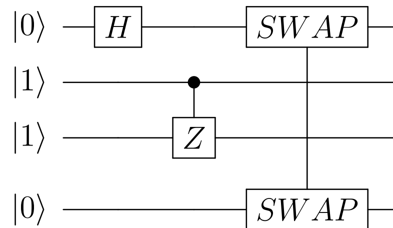


Figure 1.1: Example of the graphical representation of a generic quantum circuit.

On the left part, the initial quantum state is reported using the Qiskit notation [8]: the qubits are ordered from the least to the most significant one, starting from the top. Each qubit of the state is associated with a horizontal line where the operators related to that qubit are placed. After the initial state, all gates that compose the circuit are reported ordered from left to right. Each single-qubit gate is represented by a square containing its identifier (refer to Table 3.1 to know the employed identifiers) and it is placed on the line corresponding to its target qubit. In the case of controlled gate, the identifier is again positioned on the line related to the target qubit, while a black dot is placed on all the lines corresponding to the control qubits. These dots are connected with the gate symbol by means of a vertical line. Finally, in the case of multiple target gates, an identifier is placed on all the lines related to the targets qubits and they are connected using a vertical line.

As in the classical world, multiple architectures, compilers, and specific languages for quantum computing have been proposed during the years. However, today's most employed language to describe quantum circuits is the Open QASM [9]. It is a simple text language developed by IBM with syntax elements of C and assembly. This language can represent universal physical circuits starting from simple one-qubit and

two-qubits built-in gates. Indeed, it is always possible to decompose a multi-qubit operator into universal gates acting on one or two qubits [10]. Different versions and variants of this language are present, however, the implemented simulator refers to the Open QASM 2.0.

Chapter 2

Quantum Circuit Simulation

In this chapter, the methodology employed to simulate quantum circuits by using classical computers is described underlying the motivations behind its study and implementation.

2.1 Classical Simulation Of Quantum Circuits

Since the availability and the gate fidelity of real quantum hardware are today limited, simulators running on classical computers are needed to fully explore the possibilities that quantum algorithms can offer. These simulators are used to evaluate the possible applications of quantum computers and to estimate the reachable quantum speed-up. Moreover, simulation is needed to obtain the theoretical evolution of the quantum state through the circuit, which is used to verify the correct behavior of real quantum devices. The results obtained using simulation are compared to the ones obtained with the quantum hardware to check their correctness and identify possible errors. This is fundamental also for the development of error-correcting codes [11]. Another important advantage of the classical simulation is that the system is always under control and accessible. Differently from real quantum devices, all the probability amplitudes of the quantum state can be determined explicitly. The data can be accessed in a deterministic way and not only through measurements. Finally, the classical simulation can also consider the non-ideality phenomena inside the quantum circuits and estimate their effects on the system (a more detailed description is reported in Chapter 6). Summarizing, a classical quantum circuit simulator is helpful to:

- study the behavior of quantum devices without the need of having the access to them;

- evaluate and improve the performance of quantum algorithms and architectures;
- evaluate the speed-up obtained with quantum hardware compared to the classical one;
- verify the correct behavior of the developed quantum devices comparing their outcomes with the simulated ones;
- study and develop error-correcting codes;
- have always access to the complete information about the system;
- consider the non-idealities of the real hardware and study their effects on the system.

For all these reasons, the development of a classical simulator is today very important. Multiple simulation techniques were studied and developed, almost all of them are based on one of the three core paradigms of quantum mechanics [12]:

- **Schrödinger formalism:** it represents quantum states by their wave functions, using vectors with 2^n complex amplitudes (Dirac notation). Operators are described with unitary matrices and combined using tensor products. They are used to modify the states.
- **Feynman formalism:** it computes probabilities of individual outcomes without computing intermediate quantum states. Superposition is not considered during simulation: only one basis state is analyzed at a time and then the results are combined [13, 14].
- **Heisenberg formalism:** it considers the evolution of the operators and not of the states [15].

The most used is the Schrödinger formalism because it is straightforward, provides full information of all the possible outcomes, and can be easily parallelized on distributed architectures to improve its efficiency. This formalism is the one used also in the simulator presented in this work. The main problem of all these approaches is related to the exponential increase of complexity with the growth of the considered

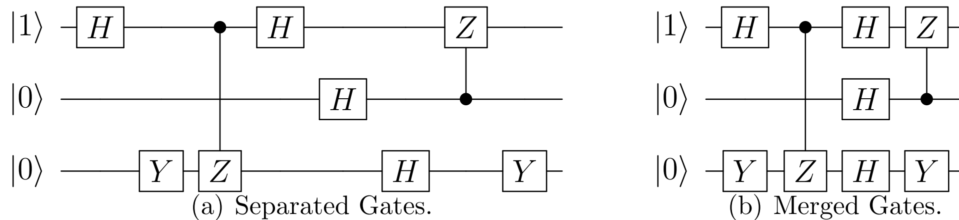
number n of qubits. Indeed, the states must be described by storing 2^n complex amplitudes and the operators must be applied to them using $2^n \times 2^n$ unitary matrices. Clearly, optimizations are needed to limit the problem and have a reasonable simulation [16, 17, 14]. Some of them are also based on the utilization of a different and more compact representation of states and operators [18, 19]. The most popular quantum computing frameworks based on the Schrödinger simulation are: ProjectQ [20], Microsoft Azure [21], IBM Qiskit [8].
As mentioned in Chapter 1, unitary matrices are applied to the quantum states to describe the effects of quantum gates. These matrices are obtained by calculating the tensor product between smaller matrices associated with the different qubits of the system, in particular:

- the qubits related to a certain operator are associated with the matrix describing it;
- the qubits without any operator are associated with identity matrices.

Considering, for example, the circuit represented in Figure 1.1, the first operator is the Hadamard gate (H). The matrix describing the effects of that operator on the quantum state is calculated as:

$$U = I \otimes I \otimes I \otimes H \tag{2.1}$$

Usually, a single gate is applied at every simulation step, so a single operator matrix and multiple identities are considered. However, there is the possibility to merge multiple non-overlapping gates and consider their effects together. Figure 2.1 reports the comparison between a simple circuit where the gates are considered one by one separately and the same circuit with multiple gates applied at the same time.



(a) Separated Gates. (b) Merged Gates.

Figure 2.1: Comparison between a circuit with “separated” and “merged” gates.

The equivalence of these two circuits can be proved by using the property expressed in Equation (2.2), referring for the first merging of Figure 2.1:

$$(I \otimes I \otimes H) \cdot (Y \otimes I \otimes I) = Y \otimes I \otimes H, \tag{2.2}$$

where $\otimes$ indicates the tensor product, I is the identity matrix, H the Hadamard gate and Y the Pauli-Y gate. This equivalence will be the basic principle of the implemented optimization of the simulation described in Section 3.4.1.
To conclude this chapter, it is important to say that the implemented simulator relies on two different representations, both based on the Schrödinger formalism:

- **Array based representation:** it is the standard approach in which mono and bi-dimensional arrays are used to store the state vectors and the matrices of operators (Chapter 4).
- **Decision Diagram based representation:** it is a different approach where states and operators are stored using graphs (Chapter 5). Its purpose is to have a more compact representation in case of redundant elements on the states and gates.

Chapter 3

Simulator Structure

In this chapter, the general structure of the simulator and the key principles of its behavior are described and analyzed.

3.1 Simulation Approach

Before starting the description of the simulator's structure and its implementation, it is necessary to clarify the approach used for the simulation of a generic quantum circuit. The main purpose of the simulation is to consider an initial quantum state and calculate its evolution through the different quantum gates of the circuit. First of all, the circuit must be created considering all the gates and their related target(s) and control(s) qubits. After that, a valid initial state must be considered to start the simulation. During the simulation, the complete circuit is crossed and the gates are applied one by one to the quantum state. Every time a gate is applied, the quantum state is partially or entirely modified depending on the gate's characteristics. In the end, after having applied all the circuit's gates, the quantum state reaches its final value and the simulation is completed.
The simulation is based on the Schrödinger formalism: it represents quantum states by their wave functions and quantum gates by unitary matrices. The modified state after a certain gate is obtained doing the product between the original state (before the gate) and the matrix related to that gate. This matrix must have the same dimension of the state and can be obtained doing Kronecker products between the matrices associated with the different qubits. In particular, the qubits related to the gate are associated with the gate's matrix while the others are associated with identity matrices.

This approach is quite simple and straightforward but has two main problems:

- **Long simulation time:** all the gates must be applied one by one to the state (multiple products between the state and the gates' matrices);

- **Large amount of needed memory:** the states and the gates' matrices have exponential growth with respect to the number of qubits.

To limit these problems and have a reasonable simulation in terms of time and memory occupation, some optimizations are performed and two different representations for quantum states and gates are investigated: the Array based and the Decision-Diagram based.
A simulation approach similar to the described one is used also in the case of noisy simulations. The main differences are related to the description used to define quantum states and the methodology used to apply noisy quantum gates. For a more detailed and accurate analysis of the noisy simulation, refer to the related Chapter 6.

3.2 Simulator Behavior

The implemented simulator is based on the simulation approach described in the previous section and can work in four different configurations:

- Ideal simulation with Array based representation;

- Ideal simulation with Decision-Diagram based representation;

- Noisy simulation with Array based representation;

- Noisy simulation with Decision-Diagram based representation.

Two different representations for quantum states and gates can be used: the Array based and the Decision-Diagram (DD) based. Their implementation and the theory behind them will be described in detail in the related Chapter 4 and Chapter 5. For now, it is sufficient to say that the vectors and the matrices are represented using mono and bi-dimensional arrays in the former and using graphs in the latter. Moreover, the noisy configurations differ from the ideal ones because in them the

quantum states are described using density matrices. For this reason, the simulation using these configurations is generally slower.
During the creation of the circuit and its simulation, a single configuration must be chosen, and it is not possible to use a mixture of them. For example, it is not possible to have a circuit represented using the Array based representation and simulate it using a state represented by Decision-Diagrams. This choice was taken because, even if the "mixed simulation" could be theoretically implemented, it would practically imply a big worsening of performance with no real advantages.
After having chosen the wanted configuration for the simulator and having set all the related parameters (see the documentation [22] for more detail about them), the circuit can be created by hand or by reading an OpenQASM 2.0 [9] file. To do this, the two classes *Circuit* and *CircOpenQASM* are employed: the first is used to create and manage a generic quantum circuit starting from its gates, the second is used to read an OpenQASM 2.0 file and generate the related circuit (as a *Circuit* object). In both cases, during the creation of the circuit, the gates are represented as objects of the *OperatorDD* or the *OperatorArray* classes, depending on the chosen representation. After that, an initial state must be defined before starting the simulation, and, for this reason, other four classes are employed, one for each configuration:

- *ArrayStateVec* in case of ideal simulation with Array based representation;
- *DDStateVec* in case of ideal simulation with Decision-Diagram based representation;
- *ArrayDensityMatrix* in case of noisy simulation with Array based representation;
- *DDDensityMatrix* in case of noisy simulation with Decision-Diagram based representation.

Finally, the simulation can be run by directly using the functions of the *CircOpenQASM* and *Circuit* classes. If no errors are encountered, the simulation proceeds until the end of the circuit. At this point, the final quantum state is available and can be measured and/or used as starting state for another simulation. The

described procedure can be applied also to multiple circuits, defined as multiple *CircOpenQASM* or *Circuit* objects, but they cannot be simulated in parallel at the same time. This means that if more than one circuit is present, the previous steps must be applied sequentially to every circuit one by one: the simulation of the next circuit cannot start until the previous one is terminated. It is important to underline that the final state obtained after the simulation of a certain circuit can be used as the initial state for another simulation of the same circuit or another one, as long as the configuration remains the same.

The described simulator's behavior is represented and summarized in the Figure 3.1.

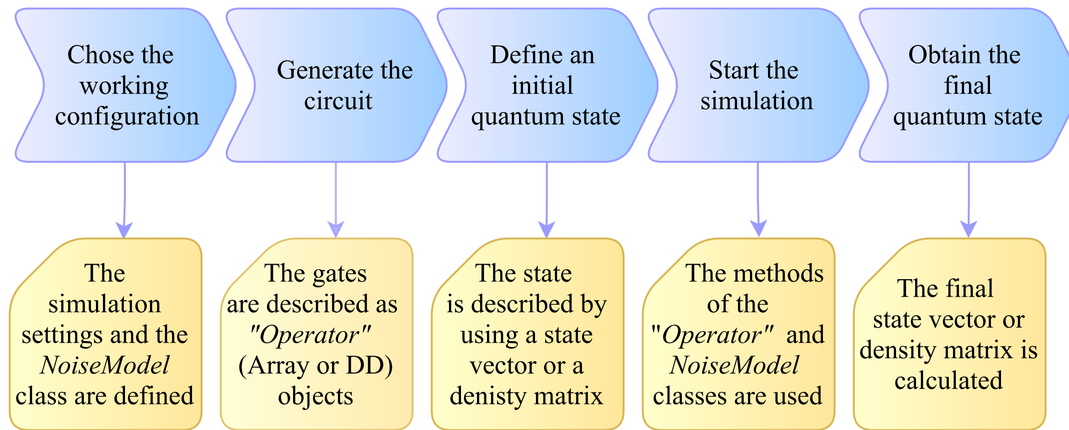


Figure 3.1: General behavior of the simulator.

3.3 Code Structure

The C++ implementation of the simulator is based on different classes that reciprocally interact with each other. They form a tree structure that has at its top the more abstract classes which do not depend on the chosen configuration and that are used to generate the circuit and manage the simulation. Under them, other more specific classes are defined to describe the quantum states and the quantum gates in all the possible configurations. Inside these classes, all the low-level calculations are implemented using different algorithms depending on the used representation. In all cases, some well-known libraries are used to simplify data management. The described tree structure is reported in Figure 3.2, where the arrows indicate the

classes' dependencies.

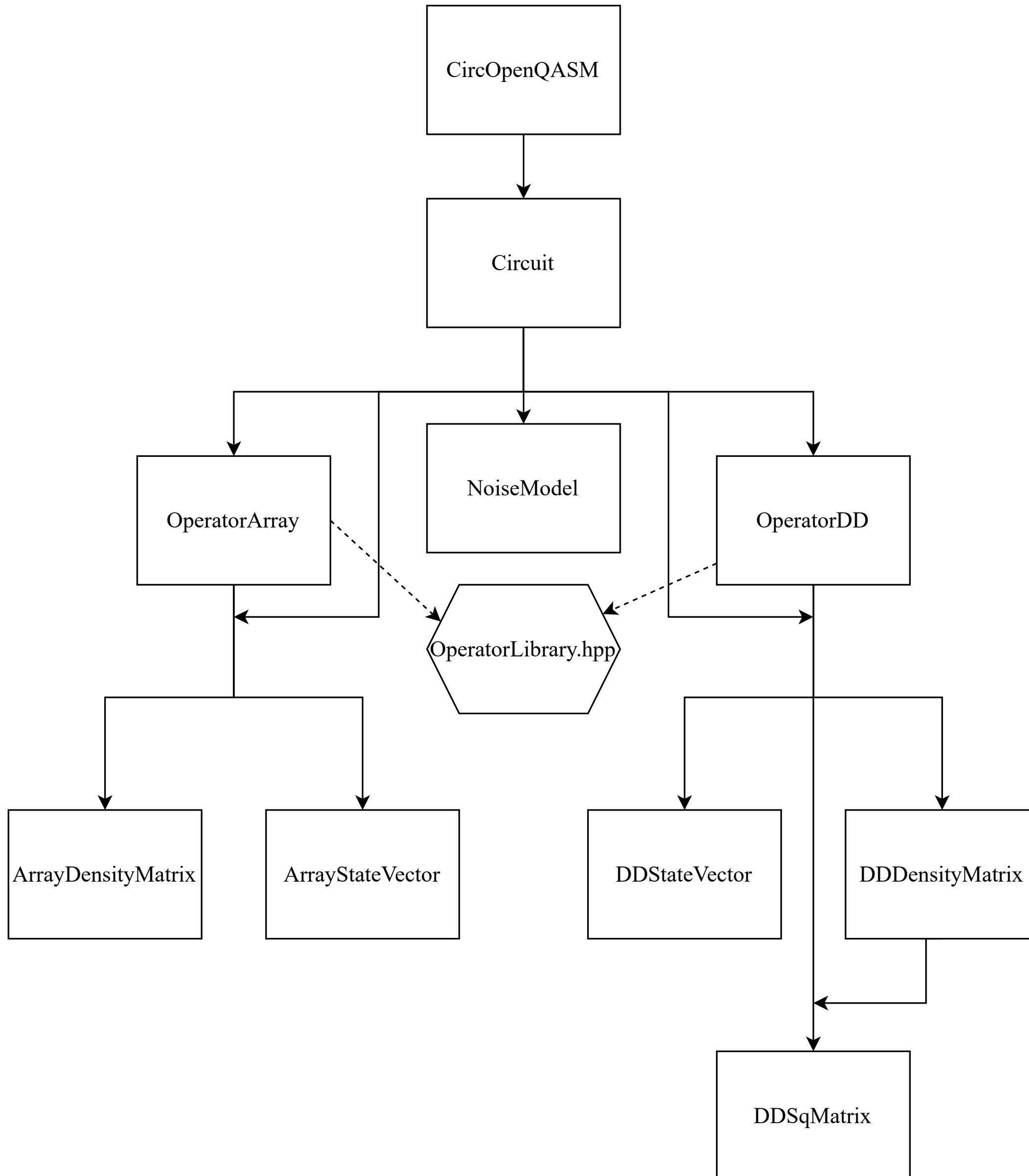


Figure 3.2: General structure of the simulator's code. The classes are reported in the rectangles and the implemented library in the hexagon. The dependencies between them are indicated by arrows.

It is important to say that the *Eigen* libraries [23] are used in all the classes related to both the Array and the Decision-Diagram based representations. In the

Array-classes, they are used to describe vectors and matrices, while in the DD-classes they are used only to manage the complex numbers. Indeed, in those situations, other specific data structures are implemented and used (refer to the related Chapter 5 for more information about them). Another important aspect that has to be underlined is the fact that the file *OperatorLibrary.hpp* is shared between the two representations. This is because it contains some functions related to the library operators that are representation-independent and can be used by both the *OperatorArray* and the *OperatorDD* classes.
All the files and the related classes will be analyzed in detail in the following. However, to have a better understanding of how the simulator is implemented, it is necessary to give an initial brief description of them. Starting from the top of the tree:

- ***CircOpenQASM***: class used to read an OpenQASM 2.0 file, generate the related circuit and simulate it;
- ***Circuit***: class used to describe and simulate a generic quantum circuit using the Array or the DD representation;
- ***NoiseModel***: class used to define the noise model in case of non-ideal simulation;
- ***OperatorArray* and *OperatorDD***: classes used to describe a generic quantum gate storing information about its matrix, its targets, its controls and its name (in case of library operator);
- ***ArrayStateVec* and *DDStateVec***: classes used to describe a generic quantum state through its state vector in case of ideal simulation;
- ***ArrayDensityMatrix* and *DDDensityMatrix***: classes used to describe a generic quantum state through its density matrix in case of noisy simulation;
- ***DDSqMatrix***: class used to represent a generic square matrix using the Decision-Diagrams.

To have a successful compilation and obtain a valid executable, all the needed libraries must be installed and all the source files must be placed in the correct directories. In particular, a directory has to be chosen to place all the representation-independent files and inside it, two sub-directories called *Array-based* and *DD-based* must be created. They must contain all the files related to the two different representations. An example of the correct organization is reported in the Figure 3.3.

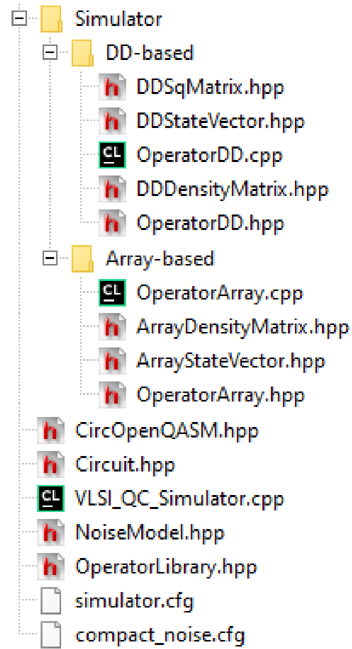


Figure 3.3: Directory tree containing the simulator files.

For more information about the compilation procedure and the usage of the *Makefile*, refer to the simulator's documentation.

3.4 *Circuit* Class

This class is used to create, manage and simulate a generic quantum circuit. Two template parameters define which representation is used by the class to describe quantum states and operators. In this way, it can be used for the ideal and noisy simulations with both the Array and DD based representations. It is based on an

std::vector containing all the circuit's gates described as *OperatorArray* or *OperatorDD* objects. As mentioned before, all the gates must use the same representation and mixed circuits cannot be generated. The gates are stored following the order they are applied to the initial quantum state, in particular, the first applied gate is placed in the first position of the vector. The final dimension of the vector depends on the number of quantum gates that compose the circuit and is controlled dynamically during the creation of the circuit. It is important to underline that this vector can be composed of every type of quantum gate but the simulation can be run only in the case of one or two-qubits gates. This is due to the simulation algorithms that work considering a maximum of two qubits related to each gate. Even if this is a strong limitation, it must be taken into account that the bigger gates can be generally decomposed into one and two-qubits gates allowing their simulation [10]. This decomposition is not automatically done by the *Circuit* class but must be done before the creation of the circuit. An example is present in the class *CircOpenQASM* for the management of the Toffoli gate: in that situation, the gate is decomposed into multiple one and two-qubits gates that emulate its behavior (see the related Section 3.5 for more details).
Various public and private methods are used to implement the different circuit's functionalities, in particular, they can be divided into:

- methods to create and modify the circuit;
- methods to obtain information about the circuit;
- methods to set and execute the simulation.

Most of the mentioned methods have a trivial implementation, however, some of them need to be analyzed in more detail to better understand their behavior. They are the ones used to add a certain gate to the circuit and the ones used to simulate the circuit. To have more flexibility during the creation of the circuit, a gate can be added in three different ways:

- the gate is provided as an *OperatorArray* or *OperatorDD* object;
- the gate's matrix and the related target(s) and control(s) are provided;

- the name of a library gate, its target(s), control(s) and parameter(s) are provided.

In the first case, the gate can be directly pushed into the vector, while in the other two a new *OperatorArray* or *OperatorDD* object has to be created starting from the provided data and then added to the vector.
Regarding the simulation methods, there is the possibility to have a complete or partial simulation of the circuit and, in both cases, the changes of the quantum state can be shown after the application of every gate or only at the end of the simulation. In this way, the user has complete control of the simulation and can retrieve all the information about the evolution of the quantum state crossing the circuit gate by gate. Moreover, in the case of ideal simulation, an optimized algorithm is implemented to speed up the simulation (described in Section 3.4.1).
The basic simulation follows the approach described in the previous Section 3.1: all gates are applied one by one to the quantum state in the correct order. In the case of noisy simulation, the noise is applied to the state depending on the considered model (see the related Chapter 6 for more details). To apply a single gate to the state the related methods of the *OperatorArray* or *OperatorDD* are used (refer to Section 4.1.2 and Section 5.2.3 for more details). However, to have a correct behavior of these methods, the quantum state must be reordered before the application of the gate. In particular, all the qubits related to the gate must be in consecutive positions and, in the case of controlled gates, the target must be placed in the least significant position. From now on, the situation of a controlled two-qubits gate will be considered, however, the behavior is similar for a gate with two targets. A simple algorithm is used to reorder the quantum state moving the control qubit until it reaches the correct position. When necessary, SWAPs operations are iteratively used to swap two near qubits until the wanted order is reached, as reported in Figure 3.4.

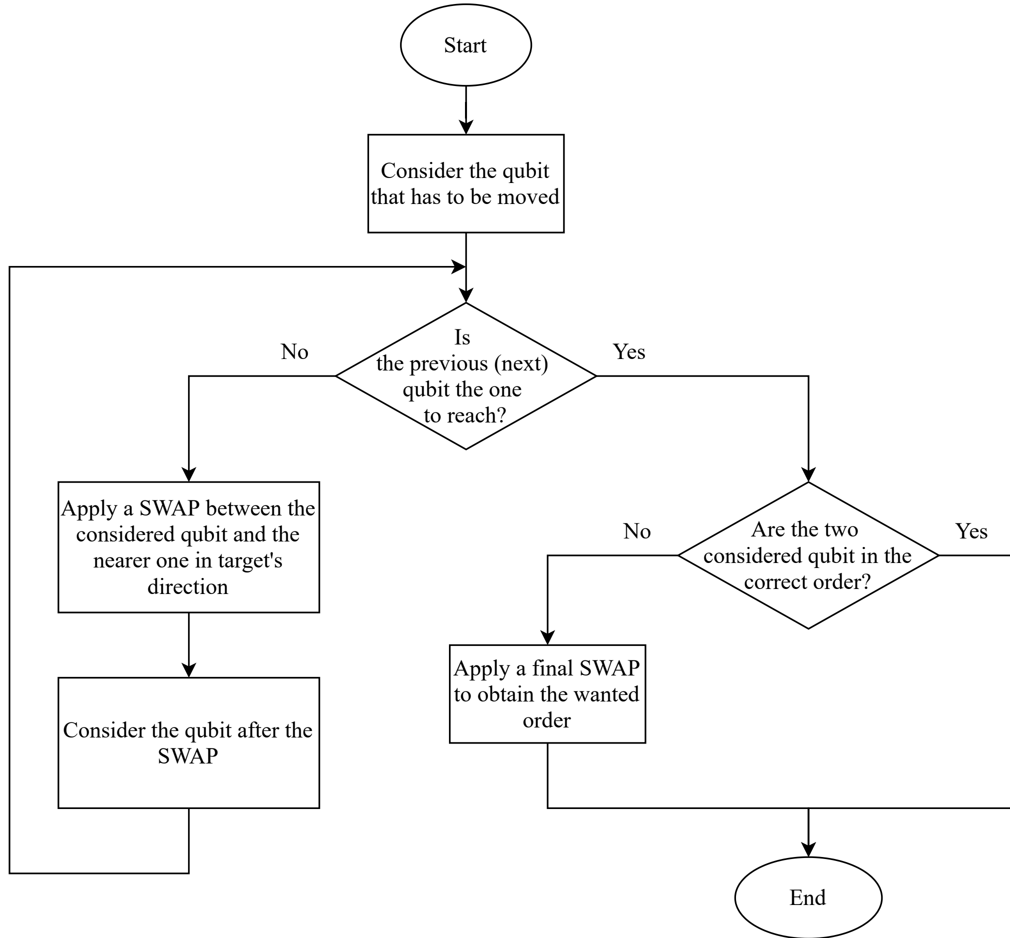


Figure 3.4: Algorithm used to reorder a quantum state applying SWAPs operations to it.

Once the reordering is complete, the quantum gate can be correctly applied to the quantum state.Then, the state must return in its original order: the control qubit must be moved again until it reaches its original position. The same algorithm is used to do this, but, in this case, the SWAPs are applied in the opposite order.
This procedure is not computationally negligible because multiple SWAPs are applied to the quantum state. Considering for example a 5 qubits circuit and a CX gate with the control on qubit 0 and the target on qubit 4 a total of 8 SWAPs are needed only to apply that single gate. This procedure must be done for each controlled gate with no-consecutive target and control qubits, greatly affecting the total simulation time.

The described algorithm for the basic noiseless simulation is summarized in Figure 3.5.

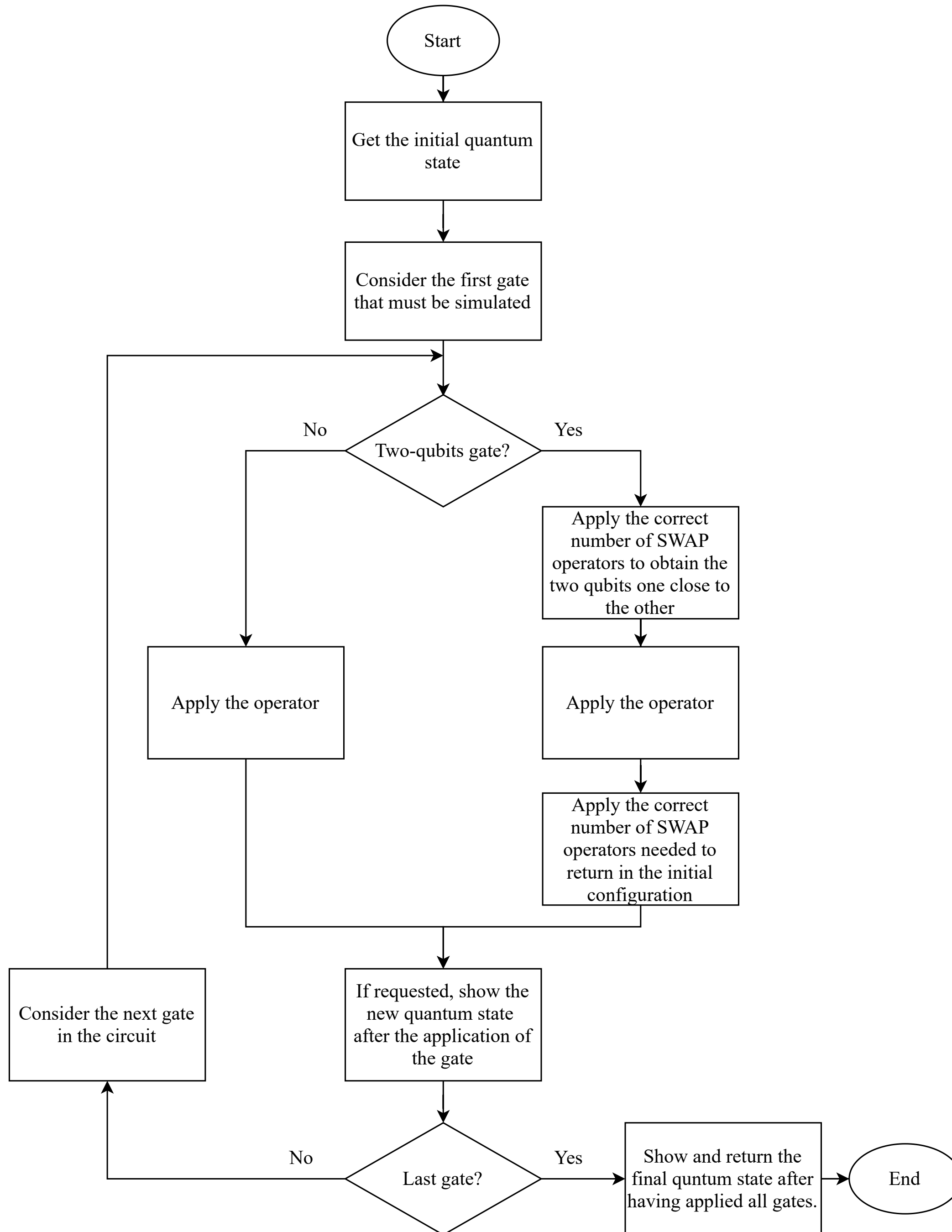


Figure 3.5: Algorithm used for the basic noiseless simulation of a generic quantum circuit.

In the case of partial simulation of the circuit, the algorithm is the same but the starting and ending gates are chosen by the user and can be different from the first and the last gate of the circuit. When the noise is considered during the simulation, it is applied to the quantum state accordingly to the defined model. Usually it is considered just after the application of a certain gate, however a more detailed discussion is reported in Section 6.3.

3.4.1 Condensed-Gate Simulation

The key concept of this optimized simulation is to group some not overlapping and consecutive gates and apply them together to the quantum state. In this way, the number of products needed to complete the simulation is lowered and so is the computational cost and the execution time. To do this, during the creation of the circuit, the gates must be divided into different groups. Each group is composed of consecutive non-overlapping gates that can be "condensed" during the simulation. An example of the group creation is reported in Figure 2.1.

To correctly manage the group division, a boolean vector is used to define if a certain gate is at the end of a group or not. Once the circuit is completed and all the groups are correctly created, the simulation can be launched. The circuit is crossed gate by gate but the quantum state is updated only at the end of a group and not at every new gate. Also, the Kronecker products needed to generate the final matrix to be applied to the state are done only at the end of a group considering multiple gate matrices together and not only one. The qubits related to all the gates contained in the group are associated with the corresponding gate's matrices and identity matrices are associated only with the free qubits.
Like in the case of basic simulation, SWAPs are needed to correctly reorder the state before and after the application of a group of gates. For this reason, during the group creation, a hypothetical overlap between two gates occurs also when they share a qubit near to their targets. This is because after the reordering that qubit will be occupied by a control and so the real overlap will be present. The Figure 3.6 can be used to better understand what is a hypothetical overlap.

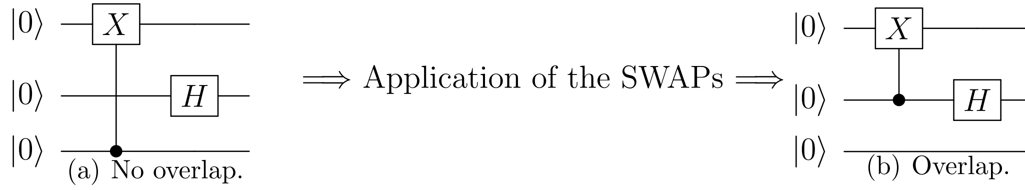


Figure 3.6: Example of the overlap between two gates after having applied the needed SWAPs.

It happens because, during the simulation, the target qubit of every gate cannot be moved except if swapped with the related control. This is because the target is used as a reference for the application of the gate and must remain in the original position to obtain the correct Kronecker product.
The Figure 3.7 reports the complete described algorithm used for the condensed-gate simulation. The "qubits vector" represents the vector used to store the matrices associated with the different qubits of the circuit. At the beginning of every block it is initialized defining every element as an identity. Then, only the matrices associated to the qubits related to the gates of the considered block are modified. Finally, the elements of this vector are used to obtain the final matrix by calculating the Kronecker product between them.

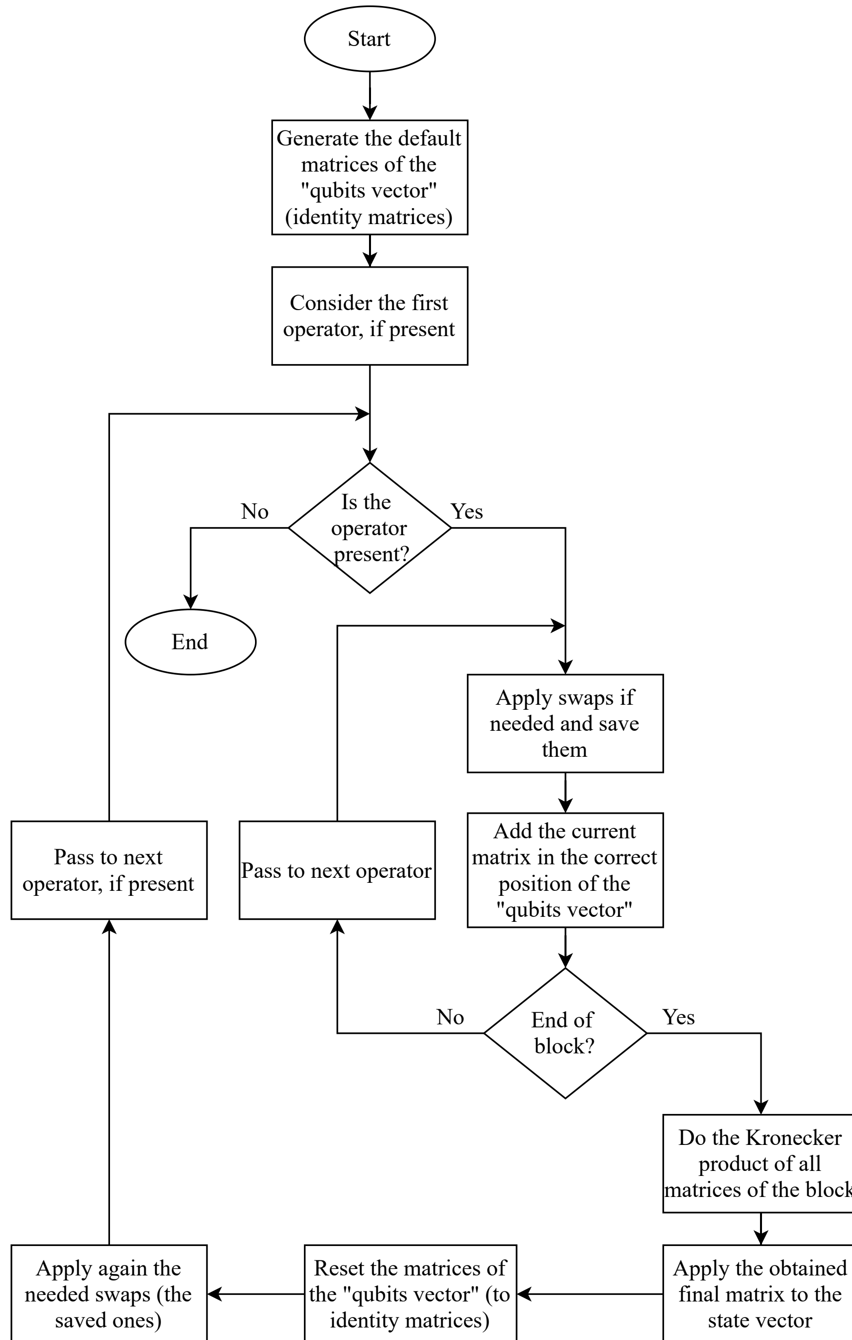


Figure 3.7: Algorithm used for the optimized condensed-gate simulation of a generic quantum circuit.

To conclude the description of this optimized simulation approach it must be underlined that this algorithm has an important drawback: the quantum state is

not updated after every gate but only after a complete group of gates. This means that some information about the evolution of the state through the circuit is lost. Moreover, this approach cannot be used in the case of noisy simulations because in them the quantum state must be updated after every gate, considering also the noise errors. For all these reasons, the described optimization can be enabled or not during the initial configuration of the simulator (see the related documentation for more details [22]).

3.5 *CircOpenQASM* Class

The class is used to automatically create a circuit reading an OpenQASM 2.0 file, simulate it, and obtain information about all the related registers and gates. Before starting the description of its internal structure and the implemented algorithms, it is necessary to specify that this class has some limitations. In particular, the target file must satisfy some conditions:

- It must be an OpenQASM 2.0 file with no errors and a correct syntax. The most common syntax errors are detected and signaled causing the controlled termination of the file reading. Instead, more complex errors inside the statements, such as the instantiation of an unknown gate, are usually managed by ignoring the statement that contains the error but sometimes can make the program crash.
- All the comments must start in a new line: it is not possible to have a comment in the same line of a statement;
- All the lines must contain a single statement;
- Custom gates (both unitary and opaque) cannot be declared or instantiated;
- Only the gates belonging to the internal library (see Section 3.6 for more details) can be instantiated with the corresponding identifier;
- The expressions used in the parametrized gates must be simple and cannot use parentheses;

- Only circuits with one qreg and one creg can be simulated (even if the data structure could handle more registers, the algorithms cannot do this);
- The *barrier* and *if* statement are not considered;

These conditions restrict the number of OpenQASM files that can be used but reduce also the complexity and the time needed to implement the algorithms to read them. This choice was made because, due to the limited amount of time, the priority was given to the development and optimization of the simulator kernel instead of this class. However, even with these limitations, the class allows automatizing the creation of the circuit, improving also the testing and profiling phases.
Internally, the class is composed of two different std::vectors containing the list of the circuit's cregs and qregs and by a *Circuit* object containing the generated quantum circuit. Moreover, some flags used to set the simulation parameters are stored, together with the location of the file to be read and its name. Two similar structs are defined to manage the cregs and the qregs; they are composed by:

- A string containing the name of the register;
- An integer value containing the dimension of the register (defined as the number of register's bits or qubits);
- An integer value defining the offset of the register: it is helpful to correctly handle multiple registers during simulation (not implemented);
- A variable used to store the content of the register.

To correctly run the simulation, the information contained in all the qregs should be combined to generate the complete quantum state that is capable of correctly handling the entanglement between the different qubits. The offset value can be used to do this correctly. However, in the current implementation, it is not used because the simulation algorithms work only with a single creg and qreg. This is mainly due to the fact that the classes used to store the content of the qregs (*ArrayStateVector*, *DDStateVector*, *ArrayDensityMatrix*, *DDDensityMatrix*) do not have methods to combine their information. In conclusion, even if the data structure is capable of correctly handling the presence of multiple qregs, the basic classes used to define

them have not this functionality and the simulation algorithms are limited to work with only one qreg.
Some public and private methods are present in the class to read the file, modify and simulate the circuit, configure the simulation parameters and access the data. Many of them have a trivial implementation that is mainly based on the functions of lower-level classes. For example, the simulation is run directly using the methods of the *Circuit* object containing the read quantum circuit. A more detailed analysis is necessary only for the methods related to the file reading. This is useful also to better understand the previously described limitations of this class.
The reading of the file is done line by line until its end. First of all, the first non-comment line is detected and its correctness is checked (it must contain the information about the OpenQASM version). Then, the reading continues deleting the initial and final whitespaces of each line and ignoring the comments. Each line is parsed to detect the statement type and the related parameters and/or arguments. After that, the effect described by the statement is immediately applied to the circuit except for the *barrier* and *if* that are correctly detected but nothing is done and for the *measure* that can be postponed depending on the settings (a more detailed description on how the postponed measurements are managed is reported at the end of this section). This behavior clearly does not allow the identification of a custom gate declaration. This is because, to correctly identify and analyze a gate declaration statement, multiple lines should be read and analyzed altogether. This is the main reason why the gate declaration is not managed by this class. Obviously, if the declaration of custom gates is not available, also the instantiation of these gates is not possible. It is important to say that, even if the custom gates are not completely managed, not only the two built-in gates of the OpenQASM 2.0 (U and CX) can be used. In particular, all the gates related to the internal library of the simulator (see Section 3.6) are automatically managed without the need for their declarations in the file. This behavior is implemented to have the possibility of using a bigger variety of gates without increasing the complexity of the algorithms needed to detect and translate the declaration statements. It is fundamental that the names used to instantiate these gates are unambiguous and equal to the ones used in the internal library (Section 3.6). In this way, the Quantum Experience standard header [9] is correctly handled by this class. It is a library of basic gates from which almost all

quantum circuits can be derived. To conclude, it must be said that the CCX gate (Toffoli gate) can be used even if it is not present in the simulator library. This is because, when detected, it is automatically decomposed into multiple library gates that emulate its behavior. The choice to handle the CCX gate even if the simulator can work with only single or two-qubits gates was made for two main reasons:

- The CCX is a common gate and its correct management is important not to further limit the class;
- The CCX gate is part of the Quantum Experience standard header.

The drawback of this approach to handling the CCX gate is that its simulation is longer: multiple gates are used and they must be applied one by one to the state. For this reason, no other gates are handled by decomposing them.
In the Figure 3.8, the described algorithm to read an OpenQASM 2.0 file is reported.

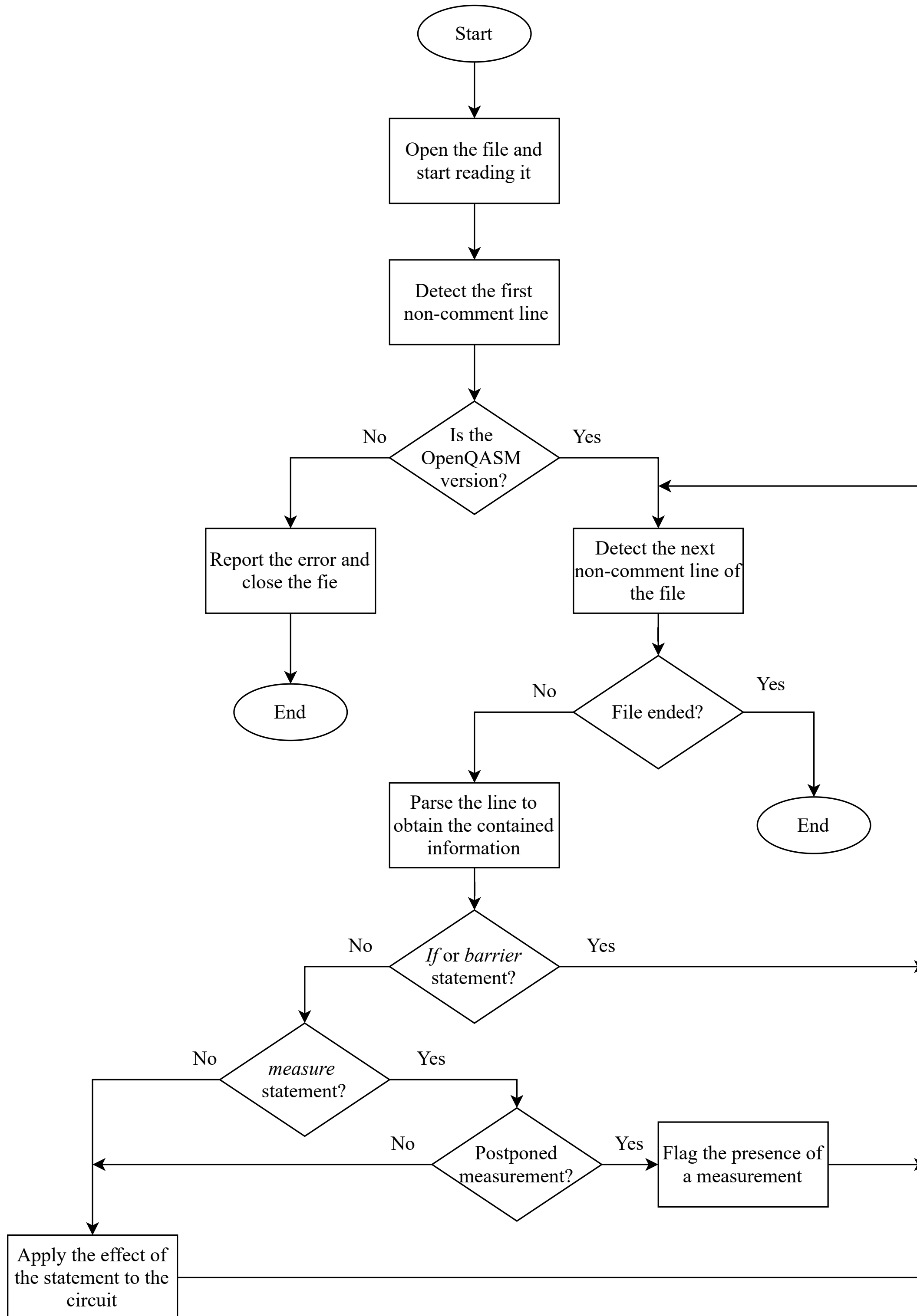


Figure 3.8: General algorithm used to read an OpenQASM 2.0 file and generate the related circuit.

The line parsing is done by creating a std::vector that contains all the information related to the statement: its name, its arguments and its parameters. These data are then used to modify the circuit accordingly to the statement type, in particular:

- ***include* statement:** the method to read a file is called again to read the target file before going on;
- **Register declaration:** a new creg or qreg struct is created, filled with the related information and added to the related vector;
- **Gate application:** a new gate is added to the *Circuit* object (only in case of library gates);
- **Measurement:** if the measurements are postponed the current statement is stored, if not the simulation is run and the registers are updated with the result of the measurement;
- ***reset* statement:** the target qubit(s) is (are) prepared in $|0\rangle$;
- ***if*, *barrier* statements:** the detection of the statement is reported but nothing is done;
- **Wrong statement:** an error message is sent and the statement is ignored.

In the case of a postponed measurement, the simulation is not automatically done when a *measure* statement is found. In this way, the file can be completely read and the circuit generated without having intermediate simulations. All the encountered *measure* statements and their position are stored and used only when the final measurement and simulation are requested. In both cases of postponed measurements or not, the creg and the qreg are updated by the result of the measure. In particular, the bits of the creg will contain the result of the measurement and the qubits of the qreg are forced in the related state. Each qubit involved in the measurement is set to its $|1\rangle$ state or reset to its $|0\rangle$ state, depending on the measurement's result. This can be obtained applying to the considered qubit the matrix $|1\rangle\langle 1|$ or the matrix $|0\rangle\langle 0|$ respectively. However, in the DD representation, a specific and optimized algorithm is used to modify the graph crossing it only once (more details can be

found in the related Section 5.3). After that, the quantum state must be normalized to obtain the collapsed state after the measure:

- In case of the state represented using a state vector the normalization is obtained dividing each element by the l^2 norm of the state;
- In case of the state represented using a density matrix the normalization is obtained dividing each element by the Frobenius norm of the state;

It is important to notice that the division by the norm can be seen also as the scalar multiplication between the state and the reciprocal of the norm. This property is used in the DD representation where the scalar multiplication can be implemented in a very efficient way (see the related Section 5.3 for more details).

3.6 Simulator Internal Library

Before starting a more detailed description of the lower-level classes of the simulator and the two used representations, it is necessary to briefly describe the used gate library. This library contains some of the most common one and two-qubits gates reported in Table 3.1. The library is contained in the header file *OperatorLibrary.hpp* and is composed by three main functions:

- ***PrintLibrary*:** to print the list of library gates;
- ***IsLibOp*:** to check if a certain gate is part of the library;
- ***DefineOperator*:** to define an *OperatorArray* or *OperatorDD* object containing the library gate.

It is important to underline that this library does not directly implement the gates but it is used to generate the related *OperatorArray* or *OperatorDD* objects. In particular, it uses some functions of those two classes to generate the optimized representation of the library gate when necessary. In this way, even if the internal description of the gate matrix is different in the two representations, the general management of library gates is the same.

Identifier	Description
I	Identity operator (single qubit)
H	Hadamard operator
X	Pauli-X operator
Y	Pauli-Y operator
Z	Pauli-Z operator
SQX	Square-root of NOT operator
P	Phase shift operator
S	Square-root of Z operator
SDG	Conjugate of the square-root of Z operator
T	Square-root of S operator
TDG	Conjugate of the square-root of S operator
U1	Single qubit rotation on Z-axis (same as P)
U2	Single qubit rotation on $X + Z$ axis
U3	Single qubit generic rotation gate with 3 Euler angles
RX	Rotation-X operator
RY	Rotation-Y operator
RZ	Rotation-Z operator
RXX	Parametric 2-qubit X⊗X interaction (rotation about XX)
RYY	Parametric 2-qubit Y⊗Y interaction (rotation about YY)
RZZ	Parametric 2-qubit Z⊗Z interaction (rotation about ZZ)
CH	Controlled Hadamard operator
CX	Controlled not operator
CY	Controlled Pauli-Y operator
CZ	Controlled Pauli-Z operator
CP	Controlled Phase shift operator
CU1	Controlled single qubit rotation on Z axis (same as CP)
CU3	Controlled single qubit generic rotation on with 3 Euler angles
CRZ	Controlled rotation-Z operator
SWAP	Swap operator
SQSWAP	Square-root of swap operator

Table 3.1: Table containing all the gates of the internal library.

Chapter 4

Array Representation

In this chapter, the standard representation for state vectors and matrices and its basic C++ implementation is described, underlining the related advantages and limitations. This representation, usually known as Array-based representation, describes the state vectors and the matrices of operators using 1D or 2D arrays whose elements are complex numbers. These arrays follow the Dirac notation and for this reason, there is an exponential increase of their sizes with respect to the increasing of the number of qubits n. In particular:

- The state vectors are represented by using an array of 2^n entries, known as probability amplitudes.
- The operators are represented by using $2^n \times 2^n$ unitary matrices.

The evident exponential growth of the data is the biggest problem related to the Array representation, as it produces an exponential increase of both memory occupation and simulation time. Considering, for example, to work with a quite simple circuit composed of 16 qubits ($n = 16$) the state vectors would be composed by $2^{16} = 65536$ probability amplitudes. During the simulation, the effect of each gate on the entire quantum state would be represented by a matrix with $2^{16} \times 2^{16} \simeq 4 \cdot 10^9$ entries that must be multiplied by the state vector. It is easy to understand how the computational cost of these operations is very high, resulting in a very slow execution in classical machines. Regarding the memory occupation, the situation could be even worse because each entry (for both the state vectors and the matrices of operators) is a complex number that occupies at least 32 bits (2 floating-point numbers are needed). In this situation a generic matrix could reach the dimension of $2^{32} \cdot 4B = (4 \cdot 2^{32})B = 16GB$!

Despite this problem, the Array representation is today used with some optimization in many simulators [8, 21, 16]. This is because it can be implemented very

easily and many algorithms used for matrix calculations were already investigated, optimized, and grouped in well-known libraries [23] [24, 25] in the past. Moreover, the dimensional growth of both the state vectors and matrices of operators does not depend on the circuit characteristics (differently from the DD representation) but only on the number of considered qubits.

4.1 C++ Basic Implementation

The simulation structure based on the Array representation was not completely implemented from scratch but uses some free libraries that offer optimized and well-tested algorithms to work with vectors and matrices. Two classes, called *ArrayStateVec* and *OperatorArray*, are present: the former is used to describe a quantum state, while the latter defines a generic quantum operator and its related information. In both cases the *Eigen* libraries [23] were chosen for the vectors and matrices management.

4.1.1 *ArrayStateVector* Class

This template class is used to generate and manage a vector describing a certain quantum state. The state is stored in memory using the Array representation and several private and public methods are used to implement all the needed functionalities. In particular, they can be divided into four categories:

- methods to generate and modify the state vector;
- methods to combine different state vectors together and apply operations to them;
- methods to access the information stored in the state vector;
- methods for the overloading of operands.

In addition to them, multiple constructors are implemented to have bigger flexibility in the creation of objects.
Most of the mentioned methods are based directly on functions provided by *Eigen*

libraries, so that their implementation is well tested and optimized. Only few of them are implemented from scratch. They are mainly the setters and getters and, as their structure is very simple, it is not necessary to analyze them.
To store the probability amplitudes of the state vector, a *Dense matrix* of the *Eigen* libraries composed by only one row and n_q columns is used. This data structure is considered, instead of the standard array, to better interface this class with the *OperatorArray* objects that describe the circuit gates and that are based on the *Eigen* libraries too.
This class seems useless as it could be replaced almost entirely by the *Eigen* libraries, without the need to encapsulate their data structure and functions inside a new class. However, the class is used to maintain the same interface that is present in the *DDSateVector* class where a state vector is stored using the DD representation. In this way a generic external structure can work with these two classes interchangeably, using the same methods. This is a key point of the simulator structure that, as mentioned before, can work equally with both the Array and the DD representations.

4.1.2 *OperatorArray* Class

This class is used to define, modify and manage a generic operator using the Array representation. Each object of this class represents a single operator (or gate) of the circuit and contains information about the related matrix (represented as a *Dense matrix* of the *Eigen* libraries), its target(s), its control(s), and its parameters if any. A string is also used to store an identifier for the commonly used operators ("library operators"). In this way the related matrix can be left empty and retrieved from the internal library (Section 3.6) only when needed, leading to a good memory saving. This approach is also useful to reduce duplicates: if two or more "library gates" are present, only their identifier is duplicated in each *OperatorArray* object, while the associated matrix is stored once in the internal library and it is then retrieved by each object only when it is needed.
Many public and private methods are used to manage the operator and to create the correct interface with the external structure. In particular, there are:

- methods to create and modify the operator using the Array representation for its matrix;

- methods to create the operators related to important quantum gates ("library operators");
- methods to combine different operators and apply operations to them;
- methods to access the information stored in an operator;
- methods for the overloading of the operands.

Furthermore, multiple constructors are implemented to have bigger flexibility.
Also in this case, almost all methods are based on the functions of the *Eigen* libraries or have a trivial implementation. Only few of them, related to the application of the operator to a certain quantum state, need a better analysis. They can be divided in two categories:

- The operator is applied to a state vector representing only target+control qubits.
- The operator is applied to a generic state vector representing more qubits than the needed ones.

In the former, the operator is directly applied to the state vector and the final state is returned as a result. This can be performed without any issue because the state vector and the matrix describing the operator are related to the same number of qubits. The situation is different in the second case where the state vector represents more qubits than the ones associated with the operator. In this case, the Kronecker product is needed to obtain the final unitary matrix that has to be applied to the state, as described in Section 3.1. The algorithm related to this second version is reported in Figure 4.1.

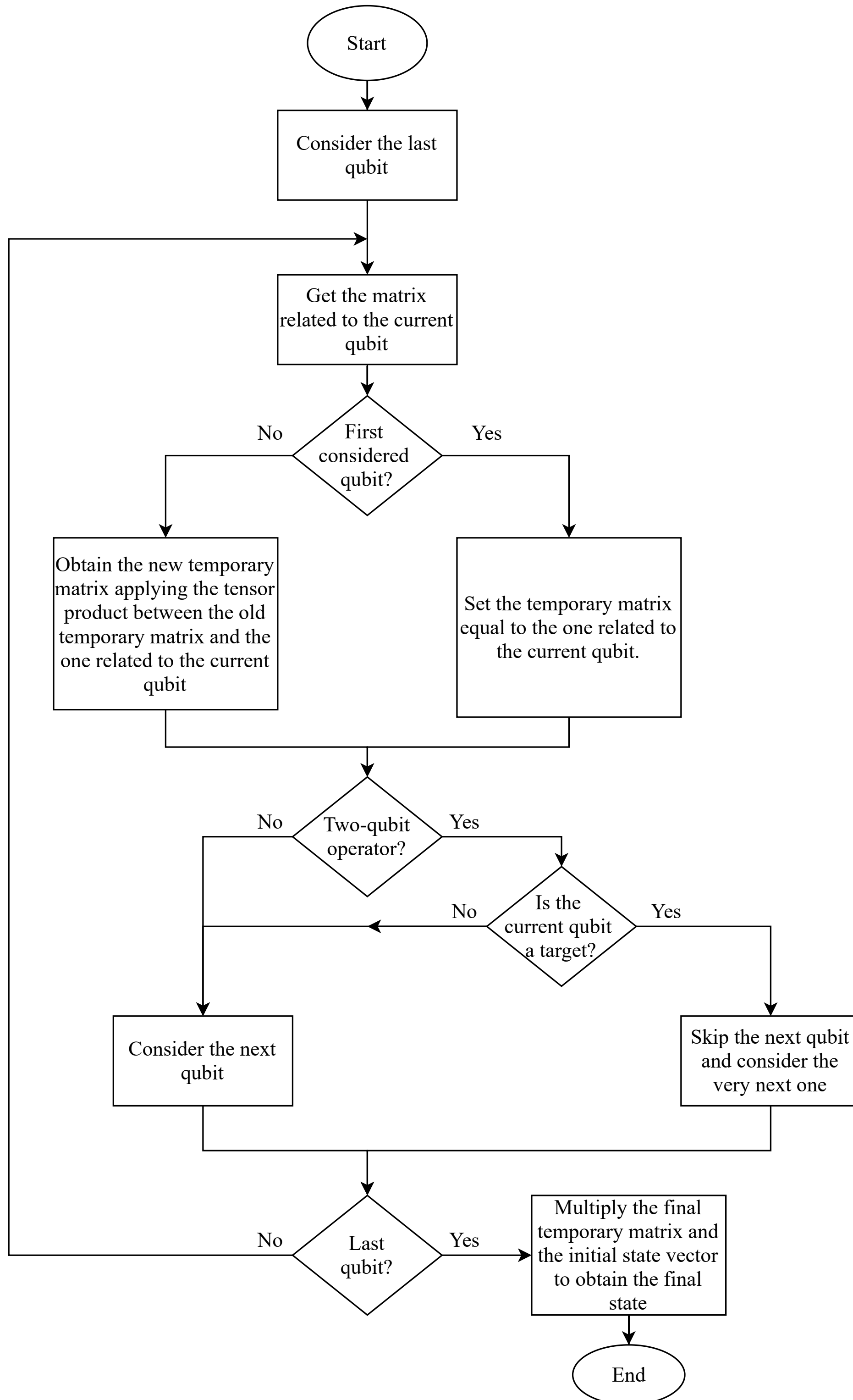


Figure 4.1: Algorithm used to apply the operator to a generic reordered state vector.

The state vector must be first reordered to have the target and the controls close to each other. This reordering is not performed directly by the methods of the *OperatorArray* class, which work considering an already reordered vector, but must be performed in advance. For this purpose, the previously described algorithm, based on the application of consecutive SWAP operations (reported on Figure 3.4) of the *Circuit* class (Section 3.4), can be used.
Finally, in case of a noisy simulation, the procedure to apply the operator to a certain quantum state is almost the same. The only difference is that the states are described using matrices, therefore the final multiplication between the temporary matrix and the state vector is substituted by the formula reported in Equation (6.5) (see Chapter 6 for more details).

4.2 Array-Based Simulation

Before moving to the analysis of the Decision-Diagram representation, it is important to summarize the general behavior and the utility of the described classes for the Array-based representation. Figure 4.2 reports in more details the parts of the simulator structure related to the noiseless Array-based representation.

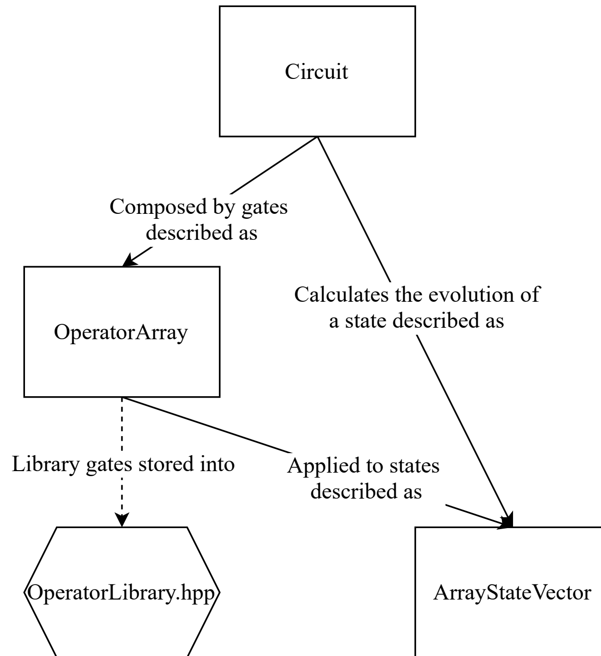


Figure 4.2: Detail of the noiseless Array-based simulator structure.

The *Circuit* class uses many *OperatorArray* objects to describe the gates of the quantum circuit. The quantum states are instead represented by using *ArrayStateVector* objects. During the simulation, an *ArrayStateVector* object is used to store the evolution of the quantum state through the circuit (see Section 3.1 for more details on the simulation approach). Every time a new gate is encountered, it is applied to the quantum state using the methods of the *OperatorArray* object, already described. The obtained result, after the application of the gate, is an *ArrayStateVector* representing the new quantum state.
The simulation proceeds in this way until all gates are applied and the final quantum state is evaluated and stored in the related *ArrayStateVector* object. The described behavior during the simulation process is reported and summarized in Figure 4.3.

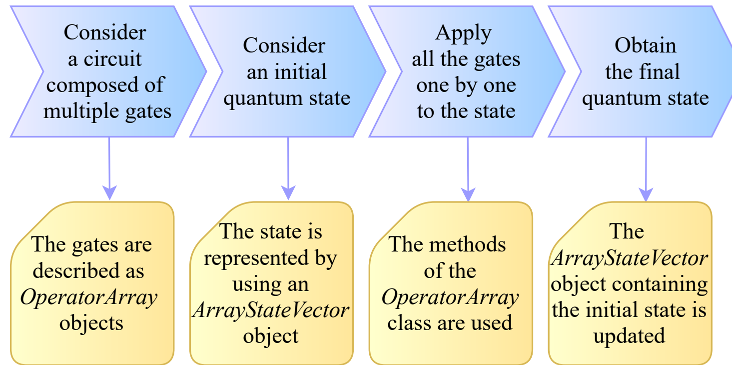


Figure 4.3: Simulator behavior during the noiseless Array-based simulation.

It is important to clarify that, in case of condensed-gate simulation (Section 3.4.1), the simulation proceeds quite differently but the interactions between the *OperatorArray* and *ArrayStateVector* classes remain the same.
The behavior here discussed is maintained also by the Decision-Diagram representation, however in this last case gates are described using *OperatorDD* objects (Section 5.2.3) and states using *DDStateVector* objects (Section 5.2.1).

Chapter 5

Decision-Diagram Representation

In this chapter, the key principles of the Decision-Diagram (DD) representation and the possible advantages in terms of memory usage when it is adopted are described and analyzed. The C++ implementation is also discussed.

5.1 Decision Diagram Theory

A Decision-Diagram (DD) is an acyclic, direct, unidirectional, tree-structured graph that can be used to store the information contained inside a generic n-dimensional array. In its simplest form, every node has a single entering edge and multiple exiting edges. The graph starts from a single root node and the data are stored inside multiple leaf nodes. In Figure 5.1 an example of a DD used to represent a mono-dimensional array is reported. The shown DD is a simple binary tree where each node has two children. To access a single data, the tree is crossed starting from the root until a leaf is reached. Every time a node is encountered, one of the two paths exiting from it has to be taken. The path to follow depends on which data has to be retrieved from the graph. For example, considering again the tree in Figure 5.1, the path to access the value '0' located on the right part of the graph is the following: *right edge* at the root level (level 0), *right edge* at the intermediate level (level 1), *left edge* at the bottom level (level 2).

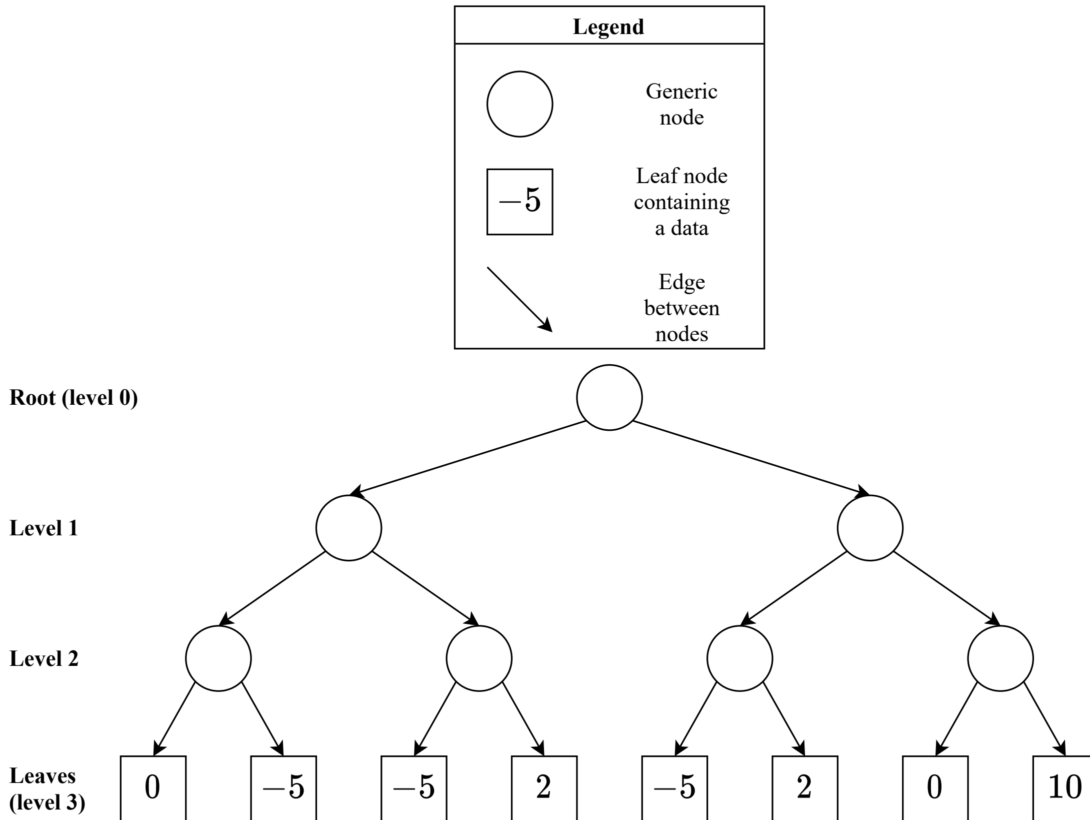


Figure 5.1: Basic structure of a Decision Diagram representing a generic monodimensional array.

The advantage of this representation is that it can minimize its size by exploiting the redundancies between the elements of the represented array. To do this, the duplicated paths are merged and each node can have more than one entering edge. Figure 5.2 reports the comparison between the simplest DD structure and the one that exploits redundancies.

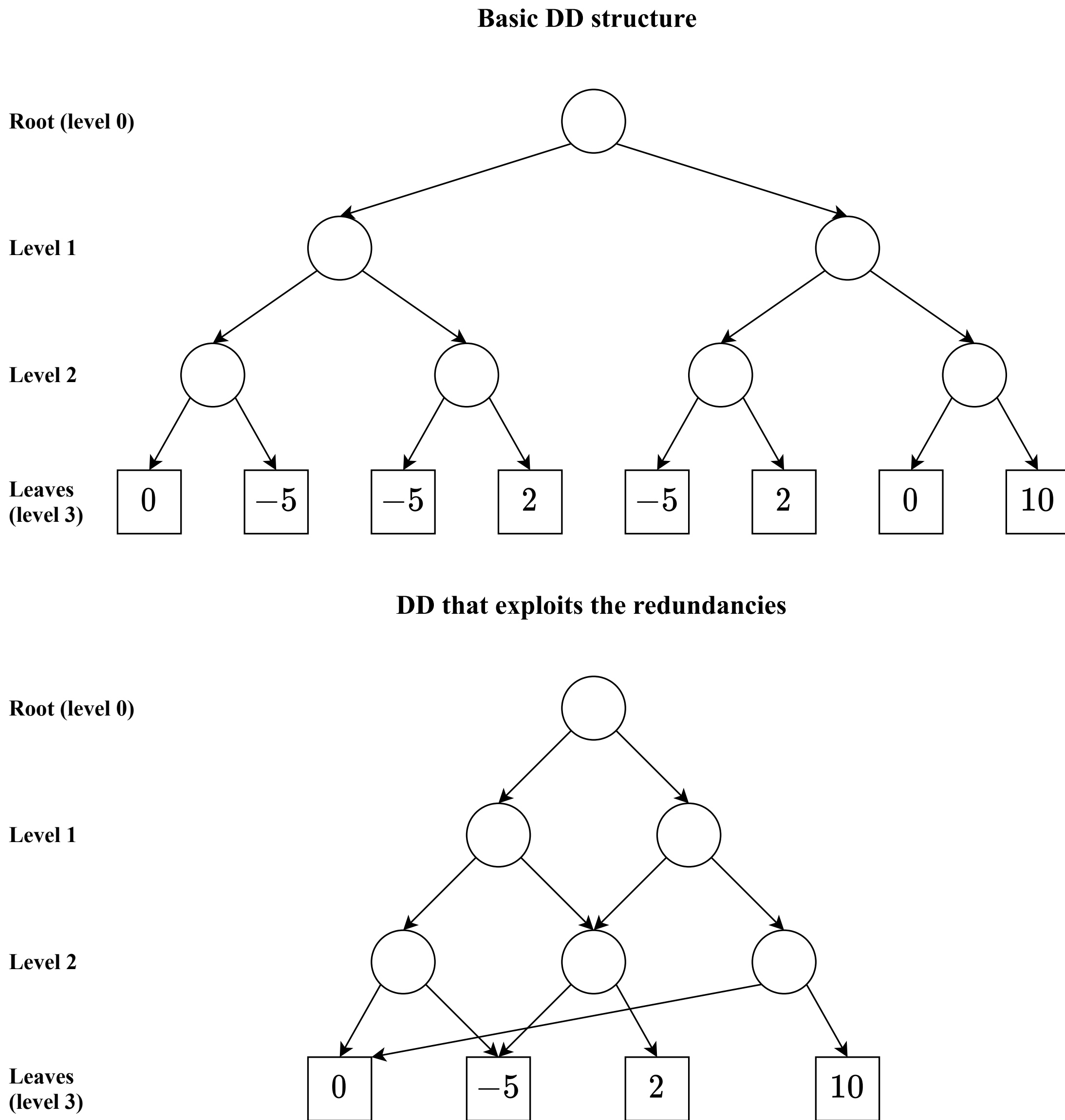


Figure 5.2: Comparison between the basic structure of a DD and the one that exploits the redundancies inside the elements of the represented mono-dimensional array.

As it can be observed, the second structure reduces a lot the tree size, without losing any information. To access the desired data, the tree is again crossed from the root to the leaves, but this time different paths can share the same nodes and edges. The time needed to cross all the tree and to retrieve a data is almost the

same, however, the memory occupation is greatly reduced.
The tree size can be reduced again by using weighted edges that allow the exploitation of more redundancies. The paths terminating with values that have the same divisors can be partially merged. A comparison between the previous tree and the one with weighted edges is reported in Figure 5.3.

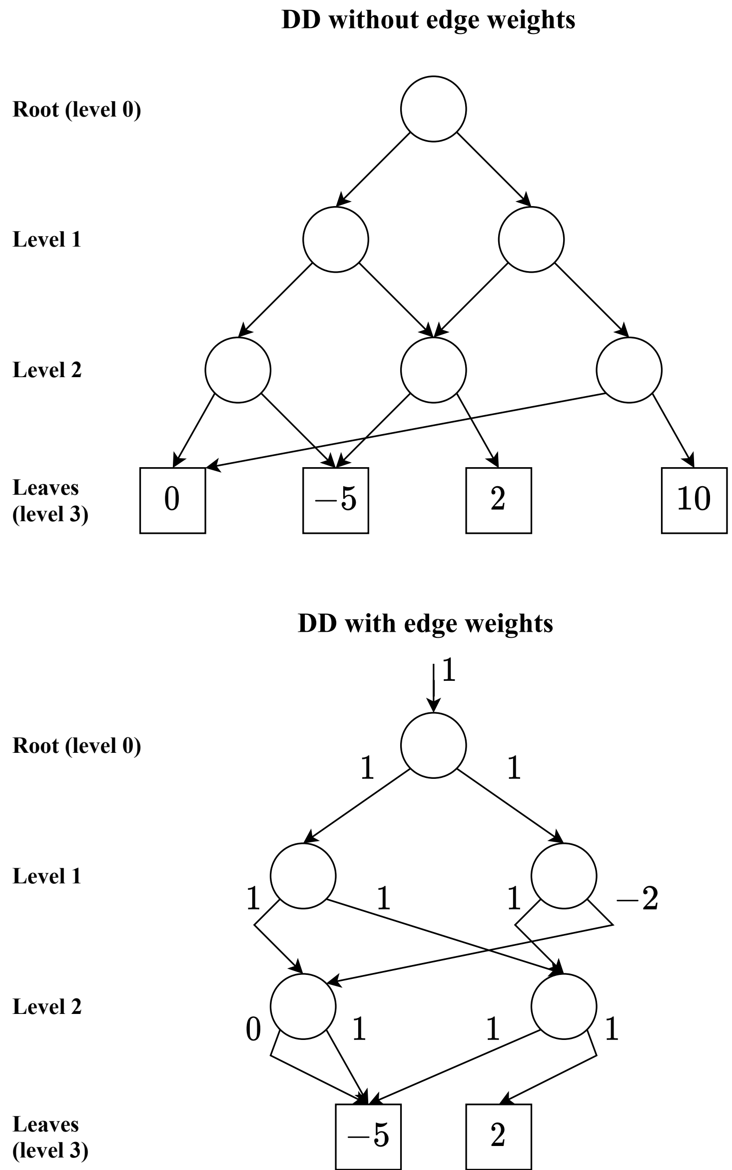


Figure 5.3: Comparison between the structure of a DD with and without edge weights.

The number of nodes of this last structure is greatly reduced compared to the initial one. The data are no more stored only on the leaf nodes but also on the edges. In particular, the tree must be crossed as before and the final value related to a certain path is obtained by multiplying the weights of all the considered edges. For example, considering the path already used in Figure 5.1 to access the data '0', the final value is calculated as: $1 \cdot 1 \cdot (-2) \cdot 0 \cdot 5 = 0$.
The structure with weighted edges is surely the one that has the minimum number of nodes. However, it is not always true that also memory occupation is minimized. This is because the edges, that before were simply pointers, have now a weight associated with them. In most cases, the consequence is a worsening in terms of memory usage. The issue can be partially solved considering to store only the meaningful weights: the not-unitary ones. Moreover, the management of the edges with a null weight can be optimized. If these edges are traversed during the access to a certain datum, the calculated final value is surely zero. For this reason, it is not necessary to continue crossing the tree after having encountered one of them: all the edges with a null weight can become null pointers and there is no need to store any weight. The optimized version of the tree structure with weighted edges is reported in Figure 5.4.

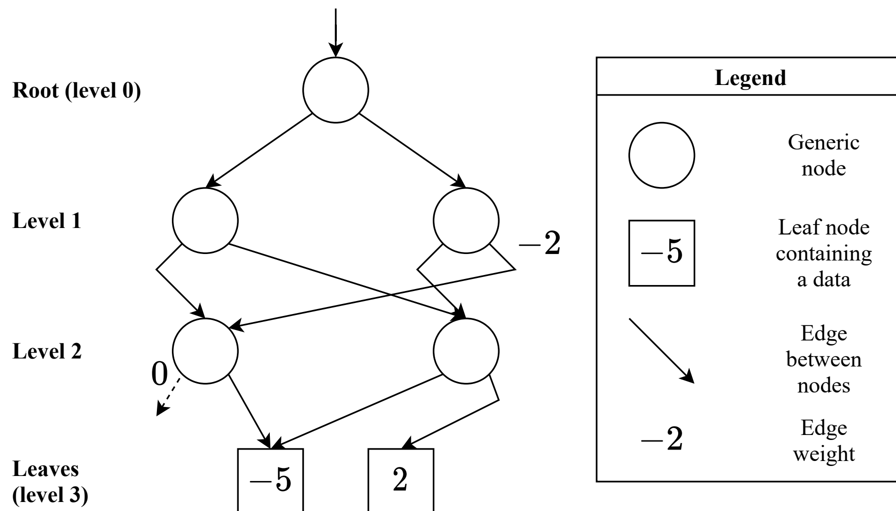


Figure 5.4: Optimal structure of a Decision Diagram representing a generic mono-dimensional array.

When the Decision Diagram is used to represent an n-dimensional array instead of a mono-dimensional one, the tree structure remains almost the same. The only difference is in the number of edges exiting from each node. In particular, a DD representing an n-dimensional array is composed of nodes with 2^n exiting edges. Thus, each new level of the tree has a bigger growth. However, more redundancies can be present in the stored data and so more optimizations can be implemented. Before describing how these structures can be used to store the information about the vectors and matrices needed to represent quantum states and gates, it is important to underline also the related drawbacks:

- The time required to access a certain value stored inside them is generally higher than the one needed in case of the standard n-dimensional array structure. This is because the wanted data must be retrieved by crossing all the graph and not only by accessing a specific memory location.
- They are useful only in case of redundancies inside the represented n-dimensional array. If the redundancies are few or they are not present at all, these data structures are generally worse than the standard ones also in terms of memory occupation.
- The operations of creation and the modification of the graph are usually computationally expensive.

The application of Decision-Diagrams in the field of quantum circuit simulation is deeply investigated in [12]. They can be used to store the information related to the state vectors and the matrices of the operators. In the first case, a binary tree is used to store the probability amplitudes of the quantum state. Every level of the tree is associated to a certain qubit starting from the most significant one that is related to the root node. The structure of the graph can be any among those described before (Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4). However, to optimize the representation, the one with weighted edges (Figure 5.4) is usually adopted. All nodes can have multiple entering edges arriving from the nodes of the upper level and have always two exiting edges that point to the nodes belonging to the lower level. In the optimal structure, the bottom level nodes, that are associated with the least significant qubit, point to a single termination with a unitary weight. This is

because the information about the final value is entirely stored inside the weights of the edges. Figure 5.5 reports an example of the described tree structure to store the probability amplitudes of a simple three-qubits quantum system represented by the following vector:

$$\varphi = \left[0,0,\tfrac{1}{2},0,\tfrac{1}{2},0,\tfrac{-1}{\sqrt{2}},0\right]^T.$$

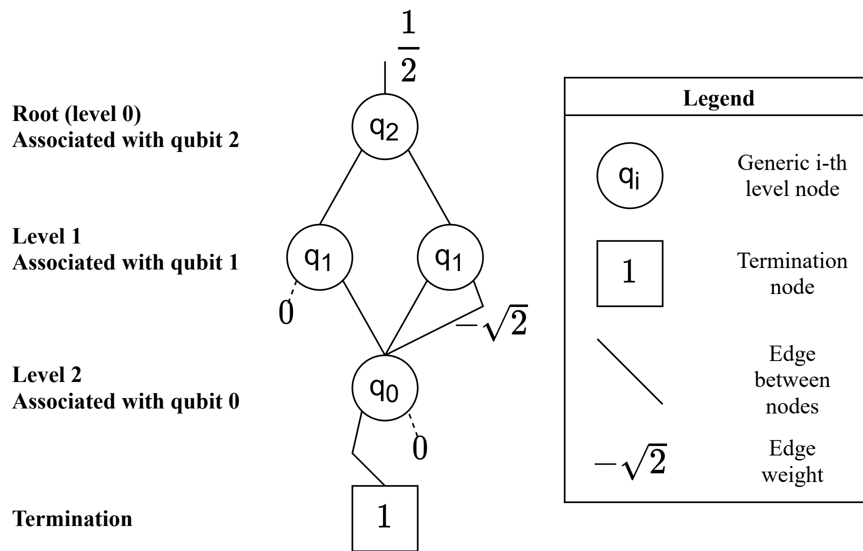


Figure 5.5: Decision Diagram used to represent a three qubits state vector.

To retrieve the amplitude associated to a certain state the tree must be crossed level by level, following a path that depends on that state. At each level, a single node is traversed. If the qubit associated with that level is set to $|0\rangle$ ($|1\rangle$) in the considered state, the left (right) edge exiting from the node of that level has to be taken to access the next level. Figure 5.6 reports again the DD representation of the vector mentioned before but in this case the path to reach the amplitude $\frac{-1}{\sqrt{2}}$, associated to the state $|110\rangle$, is highlighted:

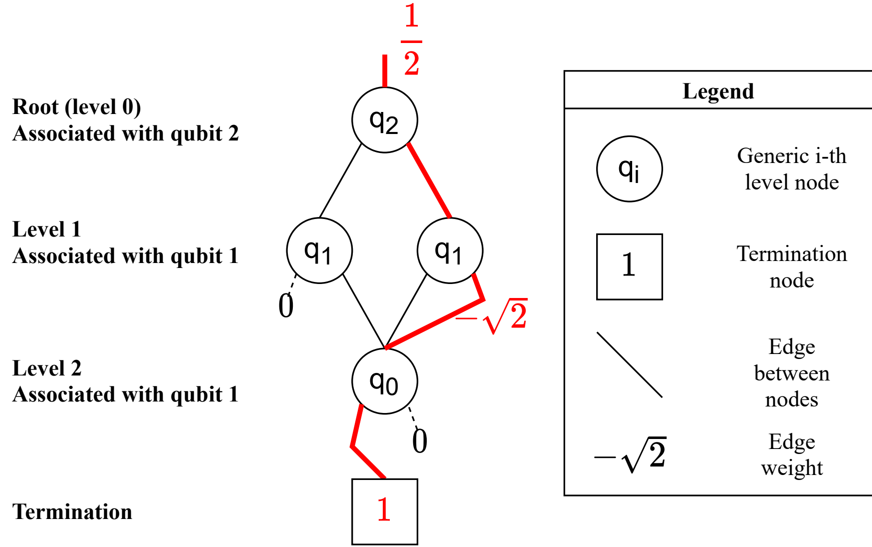


Figure 5.6: Decision Diagram representation of a three qubits state vector with highlighted the path used to access the amplitude $\frac{-1}{\sqrt{2}}$.

As expected, the followed path is: *right edge* at the top level (associated to q_0), *right edge* at the intermediate level (associated to q_1), *left edge* at the bottom level (associated to q_2). The related final value, obtained multiplying the termination by the weights of the traversed edges, is equal to: $\frac{1}{2} \cdot 1 \cdot -\sqrt{2} \cdot 1 \cdot 1 = \frac{-1}{\sqrt{2}}$.
A similar structure is used to store the operator matrices that define the gates of the circuit. The tree is again structured in different levels, each associated with the qubits of the operator. The difference is that, in this case, every node has four (and not only two) exiting edges that point to the bottom level. This is because the matrix is a bi-dimensional data structure and not a mono-dimensional one like the vector. Figure 5.7 reports the DD structure used to represent the matrix associated to a Controlled-Hadamard (CH) gate, in particular:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}.$$

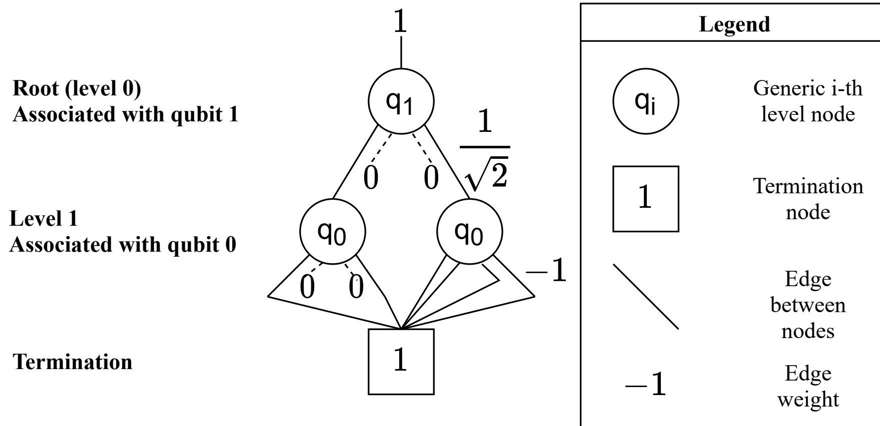


Figure 5.7: Decision Diagram used to represent the two qubits matrix associated to the CH gate.

The approach to cross the graph and retrieve the wanted data is similar to the one described before. However, this time each node has four possible exiting edges pointing to four nodes of the lower level. The choice is done considering the binary value of the row and column addresses of the element that must be accessed. Each level is associated with a bit of the two addresses starting from their most significant bits (MSBs). At every level, a new bit is considered for both addresses and the path is chosen considering the following rule:

- if both the row and column bits are at '0' the leftmost path is taken;
- if the row is at '0' and the column bit is at '1' the central-left path is taken;
- if the row is at '1' and the column bit is at '0' the central-right path is taken;
- if both the row and column bits are at '1' the rightmost path is taken.

The proposed approach for matrix access is only a practical and simplified rule that works but has no connection with the theory behind quantum operators. To have a more detailed explanation on how the DD is crossed considering that the represented matrix describes a quantum operator, refer to chapter 5.1.2 of [12].
Figure 5.8 reports again the DD representation of the CH gate and highlights the path used to reach the element $\frac{1}{\sqrt{2}}$ positioned in the row 2 (binary value 10) and column 3 (binary value 11):

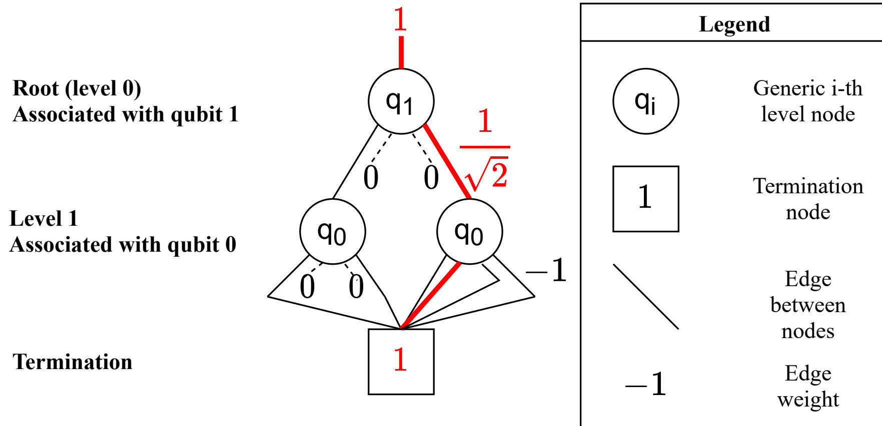


Figure 5.8: Decision Diagram representation of the matrix associated to the CH gate with highlighted the path used to access the element $\frac{1}{\sqrt{2}}$.

As described before, the path to follow depends on the binary values of the row and column addresses (starting from the MSBs): *rightmost edge* at the top level (both the row and column bits are at '1'), *central-left edge* at the bottom level (the row bit is at '0' and the column bit is at '1').

5.1.1 Implemented Structure

To conclude this analysis, it is necessary to describe the structure that is internally used by the simulator to represent vectors and matrices. The one chosen to represent state vectors is a mixture between the tree with weighted edges and the one without them reported in Figure 5.4 and Figure 5.3. It is characterized by edge weights but it does not completely exploit its benefits. In particular, only the paths that are exactly the same are merged. Instead, the ones with only a partial similarity are left separated. This representation has a very simple implementation but has some drawbacks inherited from both the two starting structures:

- The presence of a weight associated with the edges increases a lot the total memory occupation.
- The tree is not completely minimized because only part of the redundancies that can be exploited using weighted edges are effectively considered.

Figure 5.9 reports an example of the DD resulting from the representation of the following vector:

$$\varphi = \left[0,0,\tfrac{1}{2},0,\tfrac{1}{2},0,\tfrac{-1}{\sqrt{2}},0\right]^T.$$

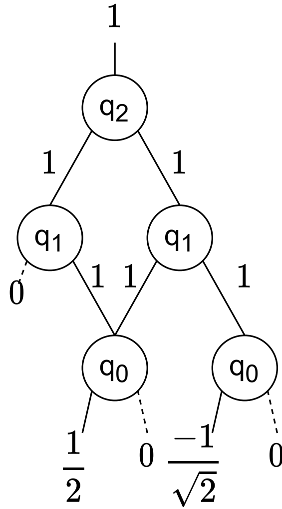


Figure 5.9: Decision Diagram used by the simulator to represent a three qubits state vector.

No improvements were applied to this structure because priority was given to the definition and the optimization of the DD structure representing matrices, which is fundamental when noise is considered. Indeed, in the case of noisy simulation, both quantum gates and quantum states are described using matrices (see Chapter 6 for more details). The chosen structure is similar to the one reported in Figure 5.4 where only the meaningful weights are stored. However, fewer redundancies are exploited and only the paths that are exactly the same are merged. Figure 5.10 reports the comparison between the optimized tree described before (Figure 5.7) and the one used by the simulator:

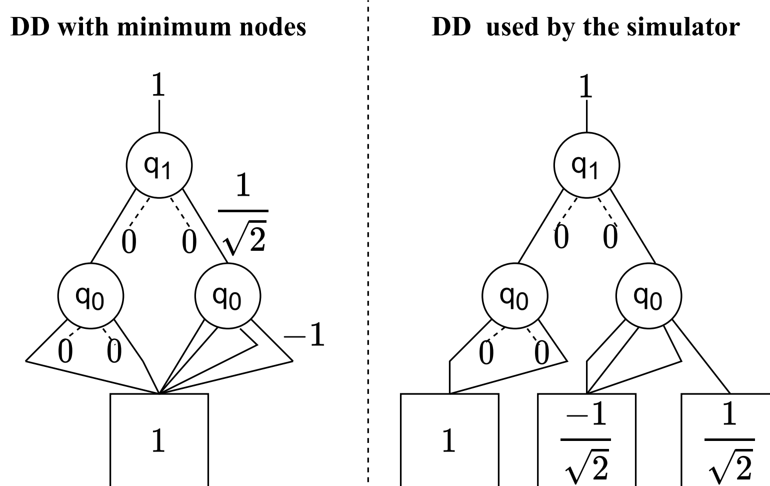


Figure 5.10: Comparison between the DD representation of the matrix associated to the CH gate used by the simulator and the one with the minimum number of nodes.

The main difference is that the total number of nodes (and so the memory occupation) is slightly higher in the structure used by the simulator. However, the creation of the tree is computationally cheaper because fewer redundancies have to be investigated. The result is a balanced structure that can be generated in a reasonable time and has a good memory occupation.
Finally, it must be also considered that the computational cost of the algorithms used to combine different Decision Diagrams generally depends on their size. For this reason, the structure with the minimal number of nodes is the optimal one when the DDs are rarely generated and often combined. This happens to the matrix representing the quantum state (density matrix) during the noisy simulation (see the related Chapter 6 for more details). Indeed, it is generated only once but combined multiple times to calculate the evolution of the state through the circuit. In this case, the structure that minimizes the number of nodes is generally better. However, due to the limited amount of time, an optimized structure for this purpose was not implemented and the balanced one described before is again used.

5.2 C++ Basic Implementation

The simulation structure based on the Decision-Diagram representation is implemented starting from two basic classes: *DDStateVector* and *DDSqMatrix*. The former is used to generate and manage a vector representing a generic quantum state while the latter is used to create a square matrix that can be used to represent both a quantum gate and a density matrix, anyway, the two classes are based on the DD representation. Next, another class, called *OperatorDD*, was implemented to describe a generic quantum operator in which the matrix is represented as a *DDSqMatrix* object.
These three classes, which contain the main differences with respect to the Array based simulation, can be considered the core of the Decision-Diagram simulation. For this reason, a more detailed analysis of them is reported, describing the key algorithms and underlying the advantages and disadvantages in their application.

5.2.1 *DDStateVector* Class

It is a template class used to generate and manage a vector to describe a certain quantum state. The state is stored in memory using the Decision-Diagram representation and all the related operations are implemented in accordance. As mentioned before, the representation is implemented using a binary tree that has as many levels as the number of the qubits of the state. Each node of the tree is defined as a struct containing different information:

- two pointers to the node's children;
- two values representing the weights of the two child's edges;
- a boolean flag used by some algorithms to detect if the node has been traversed or not.

All the termination nodes have null children and the final weights corresponding to the related amplitude of the state. The class stores only a variable containing an initial weight common to state amplitudes and the pointer to the top node (called also "head") from which all the others can be consecutively accessed traversing the

tree. Moreover, various private and public methods are used to implement all the needed functionalities, in particular they can be divided into:

- methods to create and modify the DD representation of the state vector;
- methods to combine different DD state vectors together and apply operations to them;
- methods to access the information stored in the DD state vector;
- methods for the overloading of the operands.

In addition to them, multiple constructors are implemented to have bigger flexibility in the creation of the objects.
Many of the mentioned methods have a trivial implementation and they will not be described. However, few of them consider more complex algorithms that must be analyzed. This happens in particular for the methods used to generate the Decision-Diagram representation of a certain state vector and for the ones used to combine them. In most cases, the generation of a new DD state vector is based on the creation of the boolean tree starting from the bottom level and raising up until its head. In this way, the tree is generated level by level considering the already existing children nodes and creating the related parents. The general algorithm for the creation of a generic DD state vector is reported in Figure 5.11:

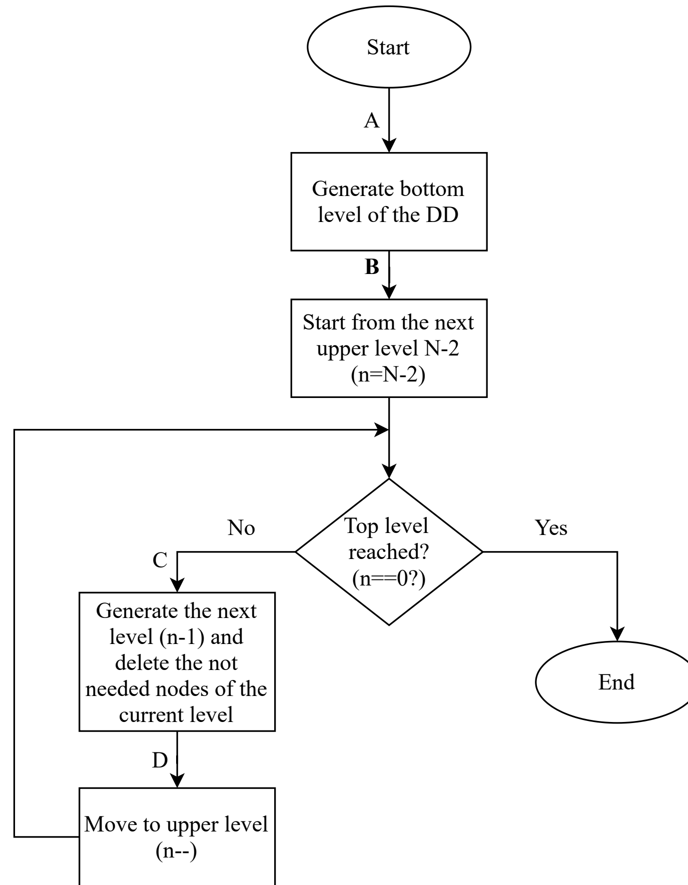


Figure 5.11: General algorithm for the creation of a DD state vector.

The discussed algorithm represents a basic version, without any optimization, and the obtained DD does not completely exploit the benefits of weighted edges. However, also in this primary implementation, the two blocks between the edges A-B and C-D are quite complex. They are the algorithms used to generate a single level of the tree (the bottom or the intermediate levels) and they are respectively expanded in Figure 5.12 and Figure 5.13:

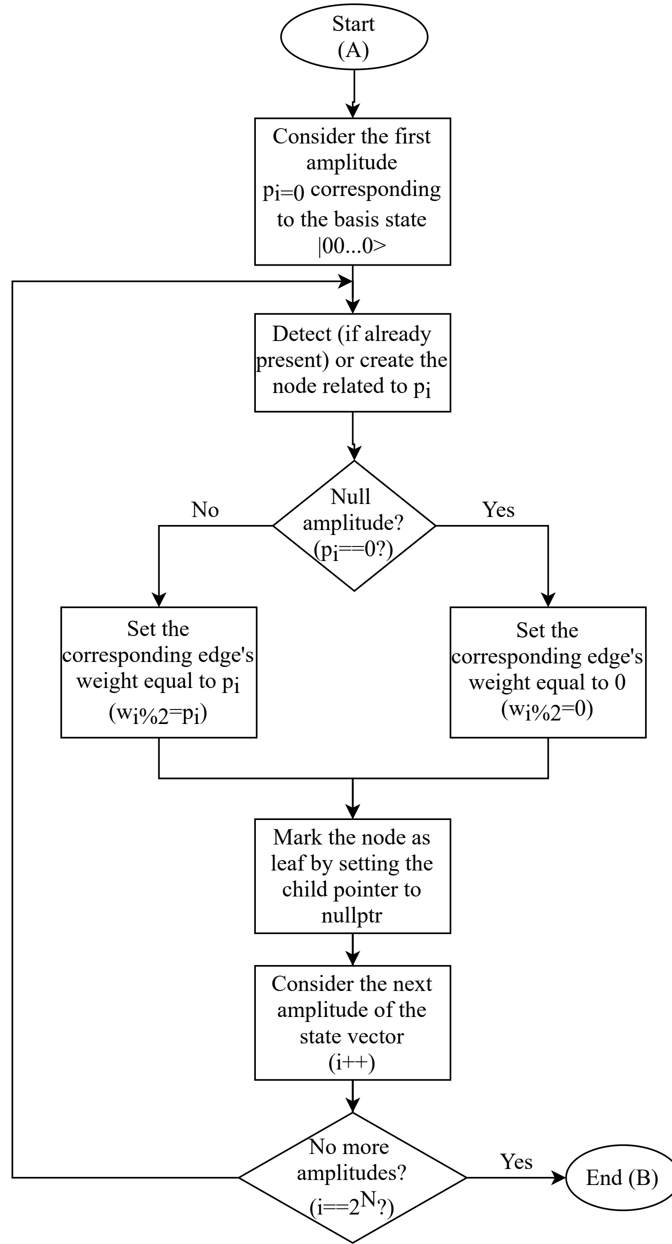


Figure 5.12: Algorithm for the creation of the bottom level of the DD state vector.

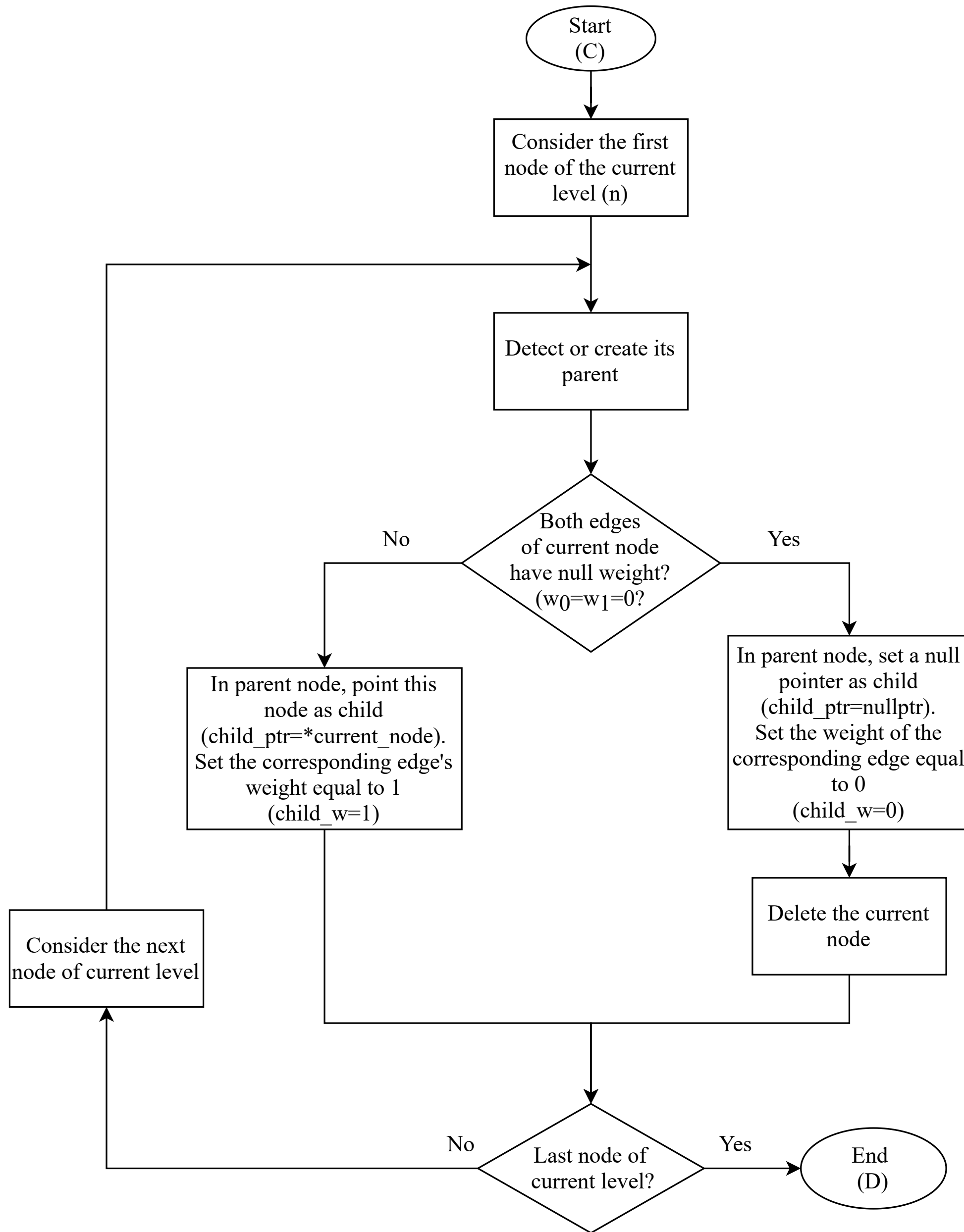


Figure 5.13: Algorithm for the creation of the intermediate levels of the DD state vector.

In both the algorithms there is a block used to detect or create a parent node when necessary. Its implementation is similar in the two algorithms and its correct behavior is important for the creation of an optimized tree with the minimum number of nodes. Another important aspect of these algorithms is the capability to

auto-detect and delete the useless node during the creation of the tree limiting the tree size and so the memory usage.

5.2.2 *DDSqMatrix* Class

This template class is used to generate and manage a square matrix, which dimension is of a power of two, using the Decision-Diagram representation. In this case, the representation is implemented using a tree in which each node can have up to four children. Each node of the tree is defined as a struct containing different information:

- four pointers to the children of a node;
- four boolean flags representing which edge has a not-unitary weight;
- an std::vector with up to four elements containing all the not-unitary edge weights;
- a boolean flag used to detect if the node has been traversed or not when the tree is crossed;
- a boolean flag signaling if the node is a termination or not.

The struct is quite different from the one characterizing the DD state vector. This is because the generated tree will be generally bigger and so some modifications are needed to limit the memory occupation for each node. The main difference is that in this case not all the weights are always stored but only the not-unitary ones. In this way, a small overhead is present, due to the need of storing the four boolean flags. However, considering that most of the edge weights are usually unitary, the final result is a reasonable memory saving. Another difference regards the management of the termination nodes. In the *DDStateVec* class the termination nodes are the bottom level nodes: each of them has two null children and two valid weights representing the amplitudes of the related state. Instead, in this class, the termination nodes are additional nodes with null children and only one weight that are pointed by the nodes in the last level of the tree. Each termination node has a different weight and so multiple bottom level nodes can point to the same

termination. In this way, if the matrix has identical elements, the final tree has a bigger number of nodes but a lower memory occupation because the final weights are stored only once. The Figure 5.14 reports the two different tree structures considering to work with two qubits:

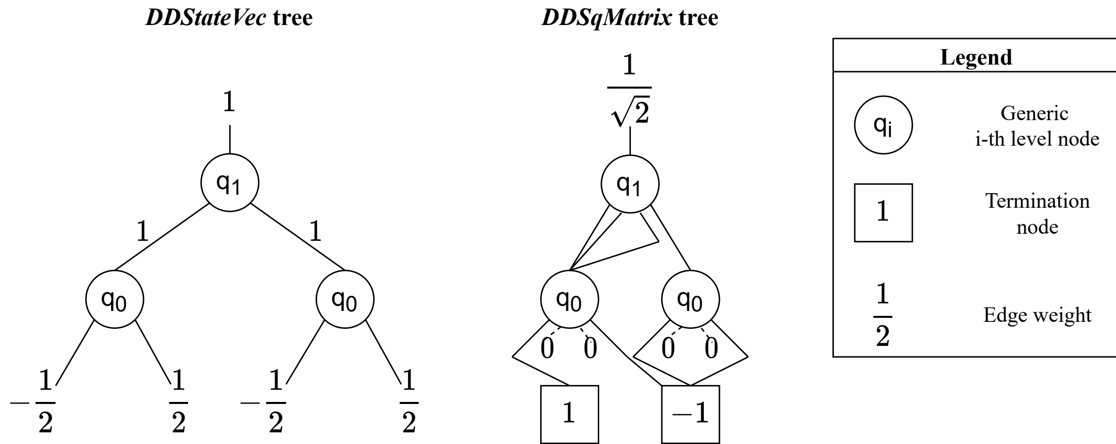


Figure 5.14: Comparison between the tree structure of the *DDStateVec* and the *DDSqMatrix*.

In the *DDSqMatrix* structure one more final level of termination nodes is needed. However, these terminations can be shared between different nodes of the upper level and the related weight is stored only once. Instead, in the case of *DDStateVec*, the absence of termination nodes leads to a repetition of the final weights and so to a bigger memory occupation. It must be also said that the dashed edges with the associated weight equal to '0' represent null pointers that do not occupy memory. A better comparison is reported in Figure 5.15 where the two approaches are applied to the same matrix:

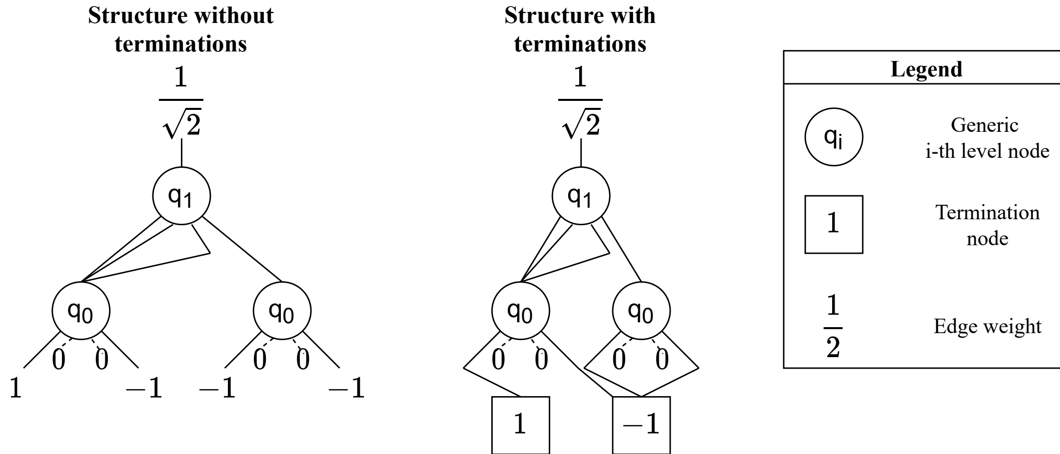


Figure 5.15: Comparison between the tree structure of the *DDSqMatrix* without and with the termination nodes.

The structure without termination nodes has to repeat the '−1' final edge weight three times and so three complex numbers must be used to store the same value. Instead, the structure with the termination nodes stores it only once in the related termination. The drawback of this second structure is that a complete node, and not only a complex value, must be used to store the final weight. However, considering that the memory occupation of a termination node is mainly due to its weight, the final result is a good memory saving.
Like in the *DDStateVec*, the class does not store all the tree but only its initial weight and the pointer to its head, however in this case also an std::vector containing pointers to all the termination nodes are saved. Moreover, various private and public methods are used for the implementation of all the needed functionalities, in particular:

- methods to create and modify the DD representation of the matrix;
- methods to create the optimized DD representation of the matrices representing the most important quantum gates;
- methods to combine different DD matrices and apply operations to them;
- methods to access the information stored in the DD square matrix;
- methods for the overloading of the operands.

In addition to them, multiple constructors are implemented to have bigger flexibility in the process of creating objects.
Only a few of the mentioned methods have not a trivial implementation and must be described in detail. This happens in particular for the methods used to generate the Decision-Diagram representation of a certain matrix and for the ones used to apply operations. Also in this case, the generation of a generic DD is based on the creation of the tree starting from the bottom level and raising up until the head. The general algorithm, reported in Figure 5.16, is very similar to the one used in the *DDStateVec* class (Figure 5.11): the tree is generated level by level considering the already existing children nodes and creating the related parents.

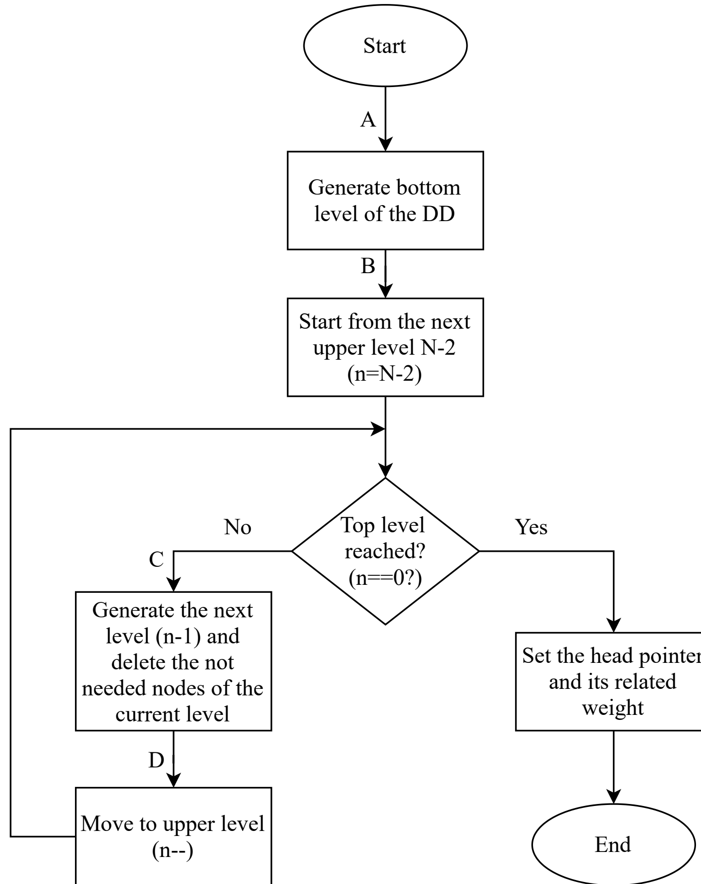


Figure 5.16: General algorithm for the creation of a DD representing a generic matrix.

The main difference with the algorithm used in the *DDStateVec* class is related

to the creation of the bottom level and the termination nodes because the complete matrix must be accessed considering 2x2 sub-matrices in the correct order. This is achieved using a row and a column offset and implementing the algorithm reported in the following picture:

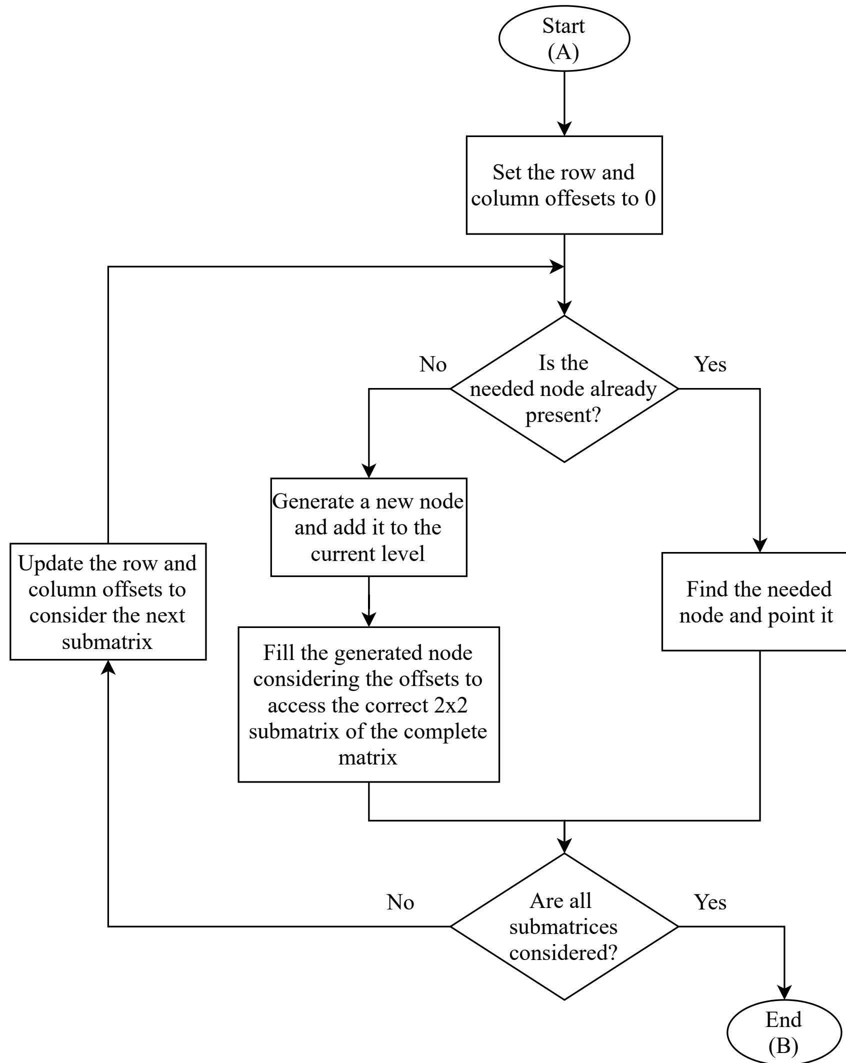


Figure 5.17: Algorithm for the creation of the bottom level of a matrix's DD.

The row and column offsets are managed by using two boolean arrays updated hierarchically. These arrays are used to access the correct 2x2 sub-matrix and fill the new node. The algorithm used to implement this is reported in the following picture:

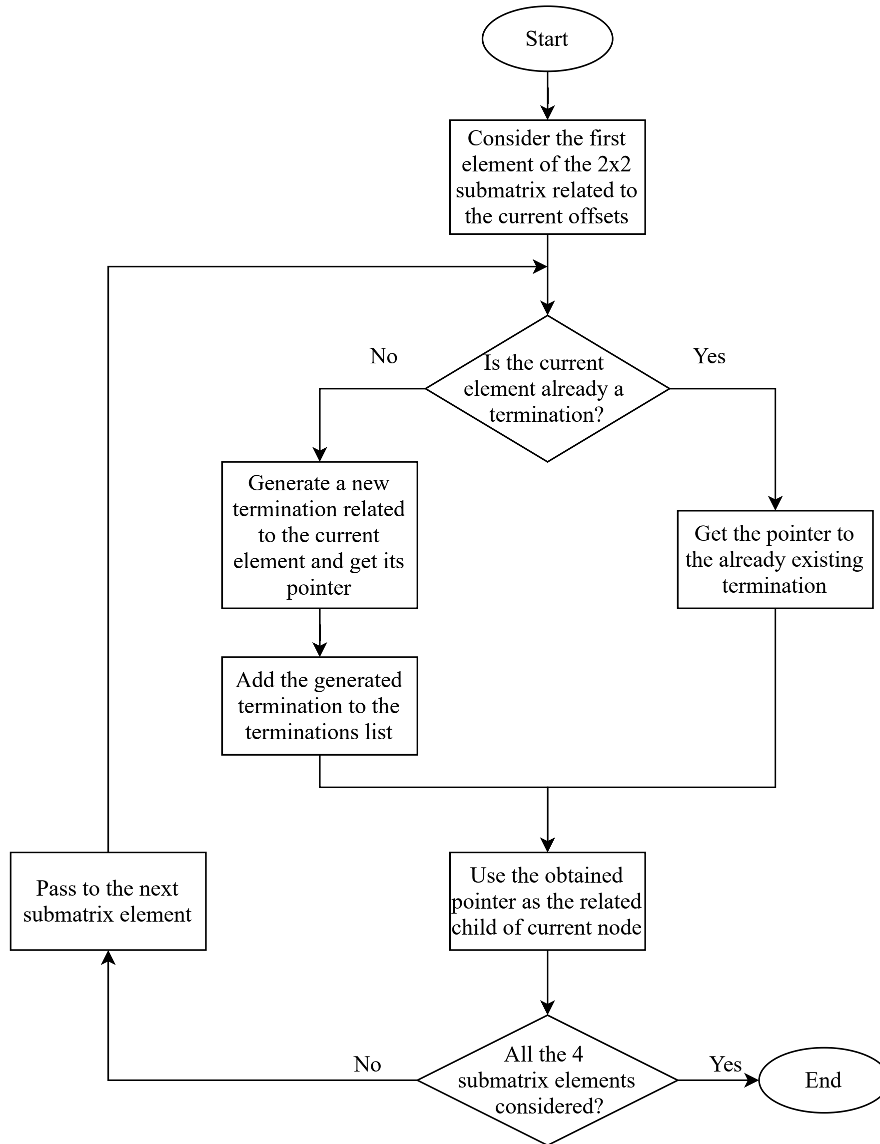


Figure 5.18: Algorithm to fill a bottom level node accessing the correct 2x2 submatrix.

After the creation of the bottom level, the general algorithm starts the generation of upper levels until the its end. This time the used algorithm, reported in Figure 5.19 is almost the same as the one seen before for the *DDStateVec* (Figure 5.13):

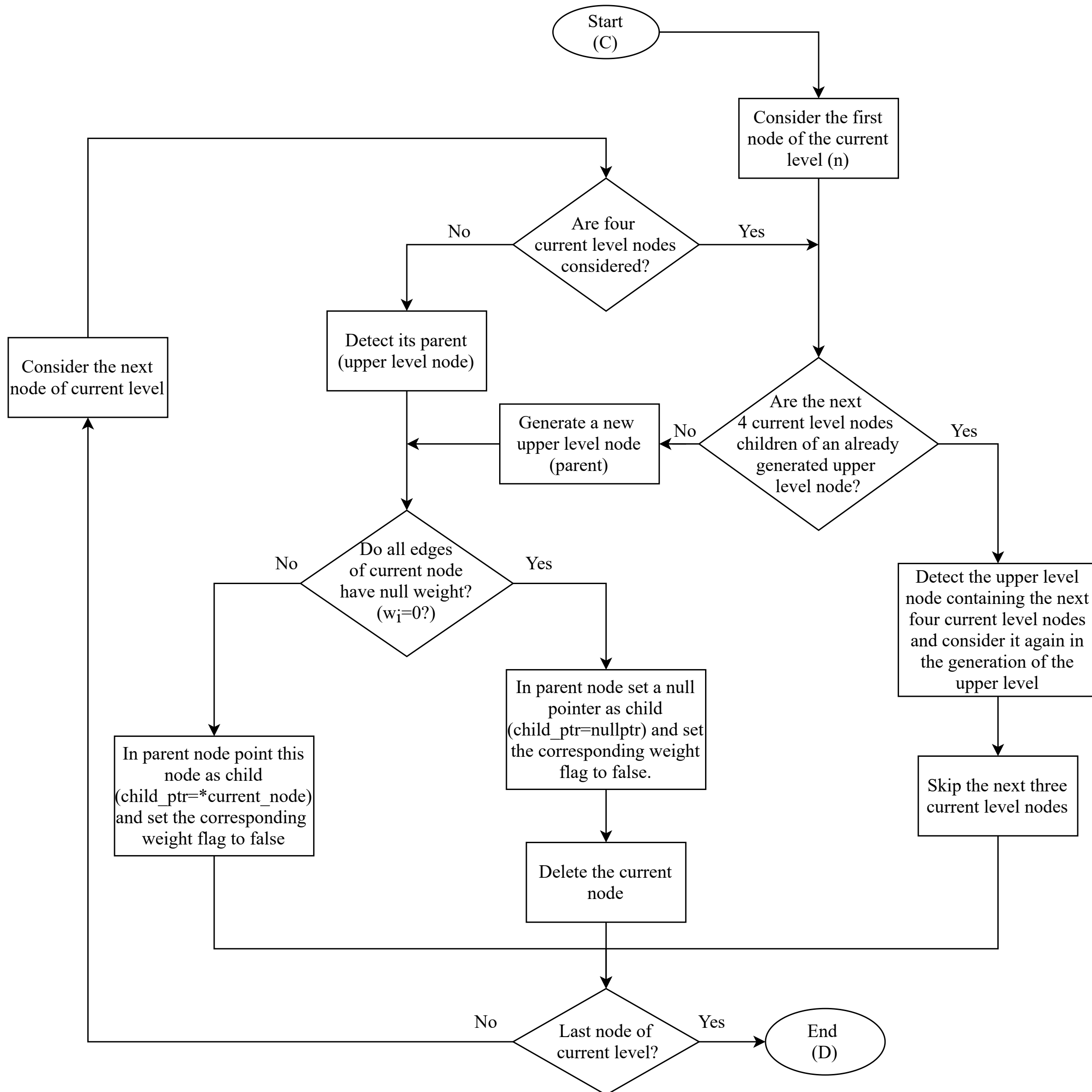


Figure 5.19: Algorithm for the creation of the upper levels of a matrix's DD.

Also in this case, during the generation of the tree, all the useless nodes are deleted to save memory and reduce the tree dimension.

5.2.3 *OperatorDD* Class

This final class is used to define, modify and manage a generic operator (or gate) of the circuit using the Decision-Diagram representation. As for the *OperatorArray* class (Section 4.1.2), each object of this class store the matrix of the operator (defined as a *DDSqMatrix* object) and all the related information. Also in this class, a string is used to store an identifier for the commonly used operators ("library operators") so that the related matrix is not stored in each object of the class and retrieved only when necessary.
In order to have the possibility to easily interchange the Array-based and the DD-based representation at a higher level, the public methods related to this class are the same that have been described for the *OperatorArray* class. In particular, we have again:

- methods to create and modify the operator using the DD representation for its matrix;
- methods to create operators related to important quantum gates ("library operators");
- methods to combine different operators and apply operations to them;
- methods to access the information stored in the operator;
- methods for the overloading of the operands.

As usual, multiple constructors are also implemented to have bigger flexibility.
The algorithms related to the different methods are the same as the *OperatorArray*, however, some of them have an internal implementation that is quite different. This is because the matrix representation and the functions to access and modify it are different: in the *OperatorArray* class the matrix is seen as an object of the *Eigen* library while in this case, it is a *DDSqMatrix*. These differences are more significant in the private methods that directly work on the operator's matrix. An example is the case of a library operator that has to retrieve its matrix from the internal library: in case of *OperatorArray*, the matrix is simply generated and assigned by the method while in the case of *OperatorDD* a new *DDSqMatrix* object must be

created and then assigned to the operator's matrix.
An important improvement is present in the calculation of the tensor product (Kronecker product) between the matrices of the two operators. This is because using the DD representation, this operation can be performed in a very simple way: the terminations of one DD are connected to the root of the other. The resulting tree is a new DD representing the matrix that contains the result of the tensor product. An example is reported in Figure 5.20:

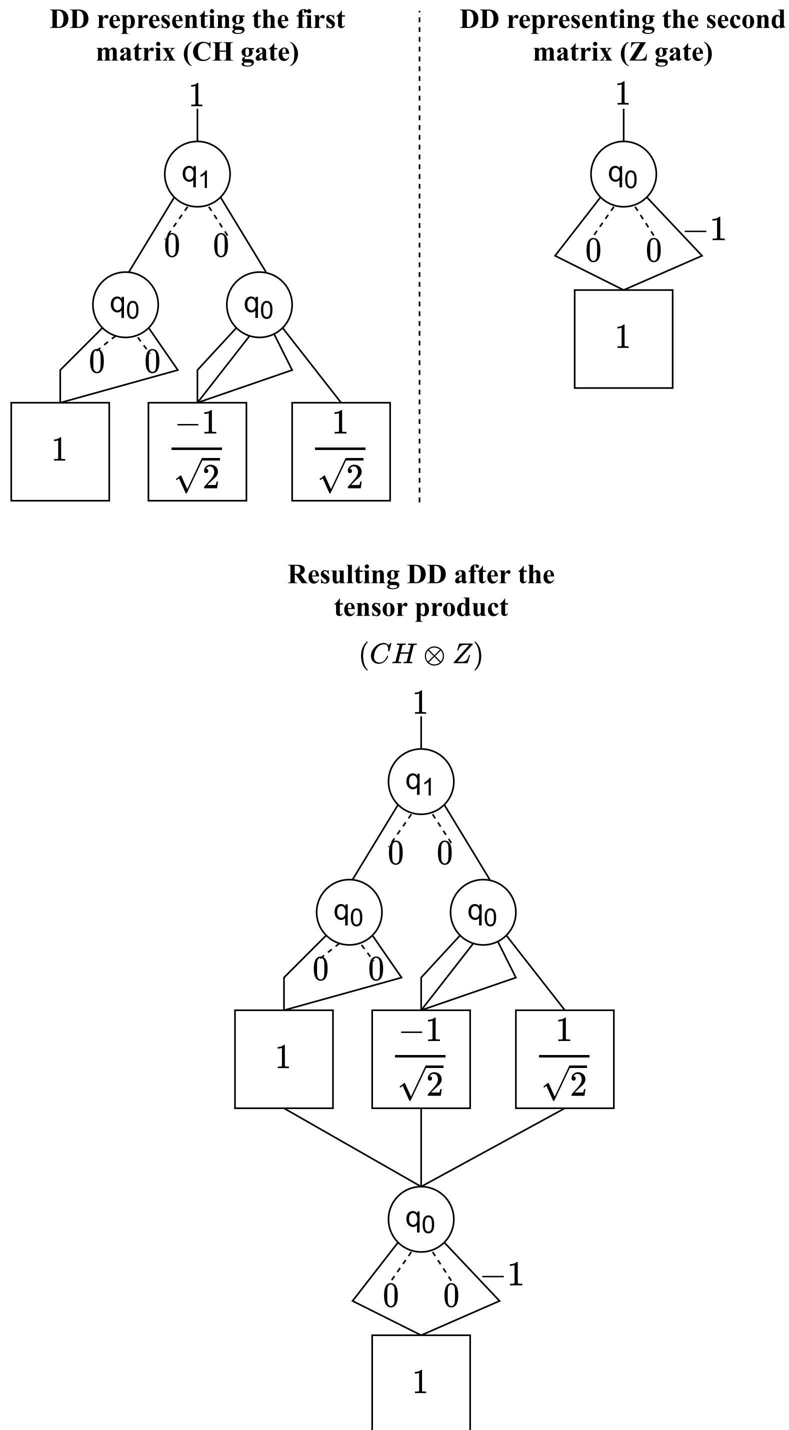


Figure 5.20: DD representation of two matrices and the related tensor product.

This operation is directly performed by the *DDSqMatrix* class with some dedicated methods. The general algorithm used for this purpose is trivial and reported in Figure 5.21:

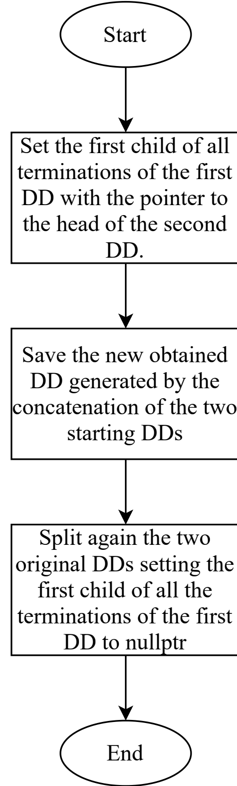


Figure 5.21: Algorithm for the calculation of a tensor product between two matrices represented using Decision Diagrams.

It can be noticed that this algorithm has a very low computational cost: it simply consists of modifying the pointers of the termination nodes. For this reason, the calculation of a tensor product using the DD-based representation is generally faster than the same operation performed using the Array-based representation. This helps to reduce the total simulation time when Decision Diagrams are used. Indeed, that operation is often performed during the simulation of a quantum circuit: to apply an operator to the quantum state multiple tensor products are used, as reported in Figure 4.1. It is important to underline that, using the algorithm reported in Figure 5.21, the newly generated tree representing the result of the tensor product

contains termination nodes also in the middle of it. This is more evident in Figure 5.20 where the terminations of the first matrix become intermediate nodes of the result tree. In that situation, the intermediate "termination nodes" are only transition nodes used to connect the two original DDs and consider the weights stored inside them. They do not correspond to a real level of the tree associated with a certain qubit. All the other algorithms are aware of this structure and are implemented accordingly.

5.3 Improvements

When vectors and matrices are represented using Decision Diagrams their memory usage is generally reduced. However, the tree structure increases the computational cost and the execution time of the standard algorithms used to implement many basic operations. This is because every time a single data has to be accessed it must be retrieved by crossing all the DD and not by simply pointing to a certain memory location as in the standard representation. Some of these algorithms can be modified by exploiting the tree structure to improve their efficiency. Sometimes, the obtained speed-up can also overcome the performance of the same operation performed by using the standard representation. One example is the scalar product for both vectors and matrices that can be implemented very easily using the DD structure with weighted edges: only the weight associated with the edge entering into the root node ("initial weight") has to be modified. A single scalar multiplication is needed independently on the vector or matrix dimension. This is because all the data stored inside the DD are always retrieved by starting from that weight. Instead, in the standard representation, one multiplication is needed for every element of the vector or matrix.

Another operation that can be improved for both vectors and matrices is the row-column product. From now on, the description is reported considering to work with matrices, however, the same property can be applied also to vectors. The key idea is to use the property reported in Equation (5.1) to calculate the product of two matrices M and N by considering the products of their sub-matrices M_{ij} and N_{ij}

[18]:

$$
\begin{aligned}
M \cdot N = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \cdot \begin{bmatrix} N_{00} & N_{01} \\ N_{10} & N_{11} \end{bmatrix} = \\
= \begin{bmatrix} M_{00}N_{00} + M_{01}N_{10} & M_{00}N_{01} + M_{01}N_{11} \\ M_{10}N_{00} + M_{11}N_{10} & M_{10}N_{01} + M_{11}N_{11} \end{bmatrix} = \\
= \begin{bmatrix} M_{00}N_{00} & M_{00}N_{01} \\ M_{10}N_{00} & M_{10}N_{01} \end{bmatrix} + \begin{bmatrix} M_{01}N_{10} & M_{01}N_{11} \\ M_{11}N_{10} & M_{11}N_{11} \end{bmatrix}
\end{aligned} \tag{5.1}
$$

Recursion can be used to reiterate the property until two 4x4 matrices are considered and the calculations can be done directly with their elements. At this point, recursion stops and the multiple obtained results are combined to determine the final overall product. In the DD structure, each node can be considered as the root node of a new matrix described by the tree starting from it. The children of that node represent its four sub-matrices. So, every new level of the overall tree is composed of the sub-matrices of the upper level nodes. For this reason, a new level of recursion corresponds to a new level of the tree. The Figure 5.22 reports the described algorithm used to calculate recursively the row-column product between two matrices (in the algorithm the matrices and sub-matrices are represented by the nodes of the tree):

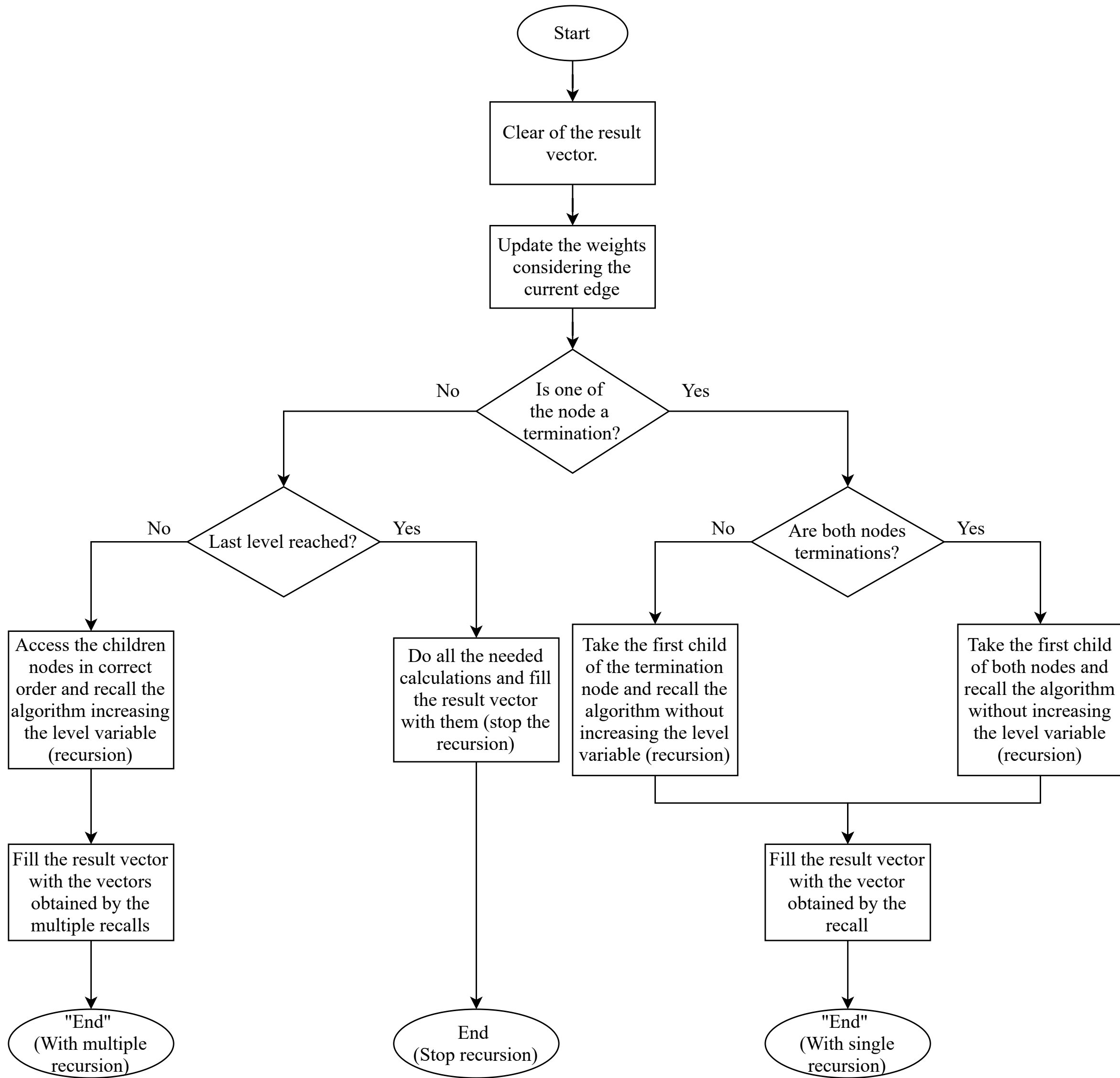


Figure 5.22: General algorithm for the calculation of the recursive row-column product between two matrices represented using Decision Diagrams.

It is necessary to say that the portion of this algorithm related to the identification and management of termination nodes (right part), is needed to correctly handle the used tree structure. In particular, considering how the Kronecker products are generated, it is possible to have “termination nodes” in the middle of the tree. They are only transition nodes that do not correspond to a real new level of the tree and so they have not to start a multiple recursion. They are considered

only to correctly update the related path weight.
Figure 5.23 reports the portion of the algorithm used to reiterate the recursion and calculate the partial row-column product in case of intermediate levels:

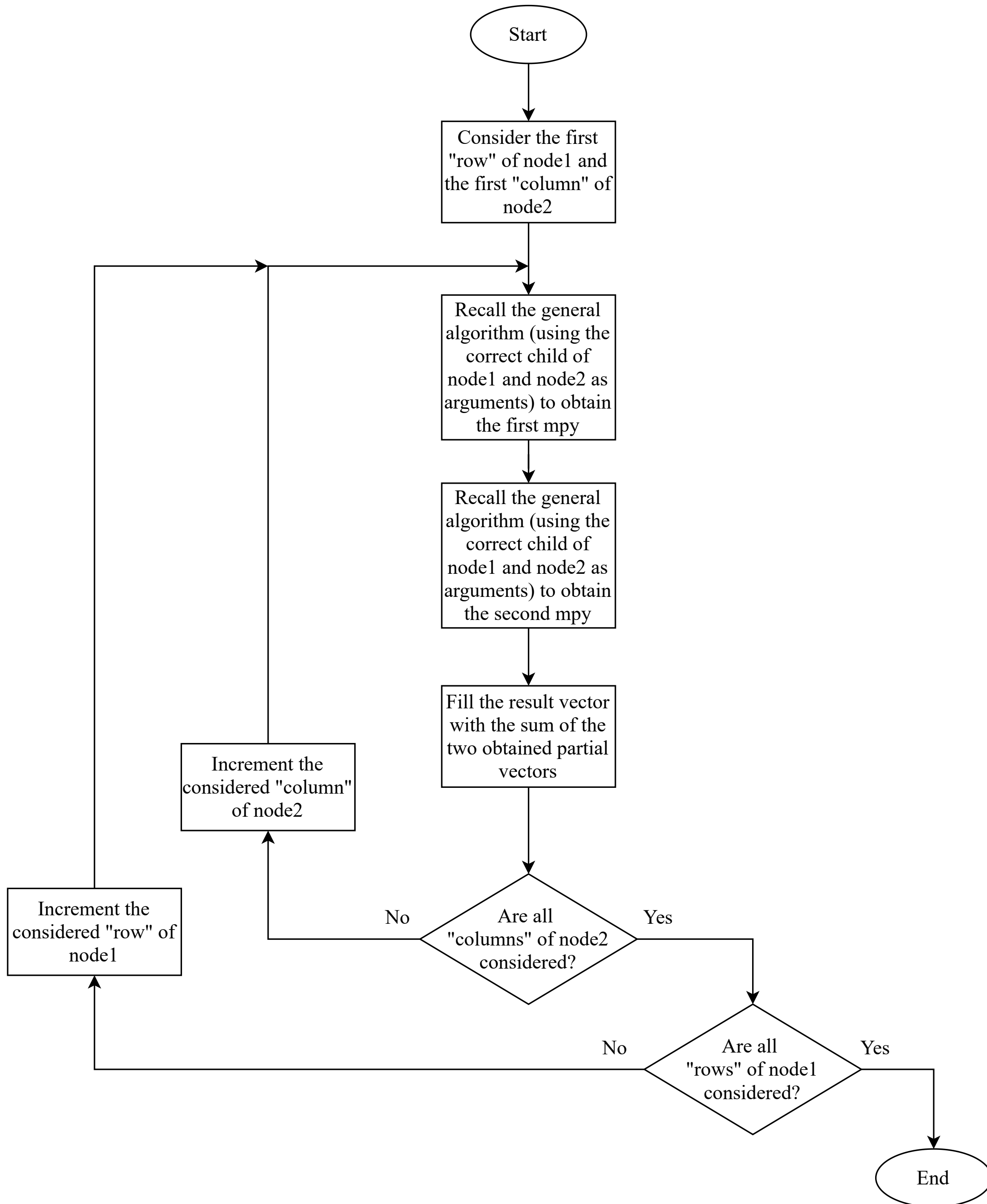


Figure 5.23: Algorithm for the calculation of the intermediate levels of the recursive row-column product between two matrices represented using Decision Diagrams.

The behavior of final levels, in which recursion stops and only the calculations

between the elements of the matrices are implemented, are reported in Figure 5.24:

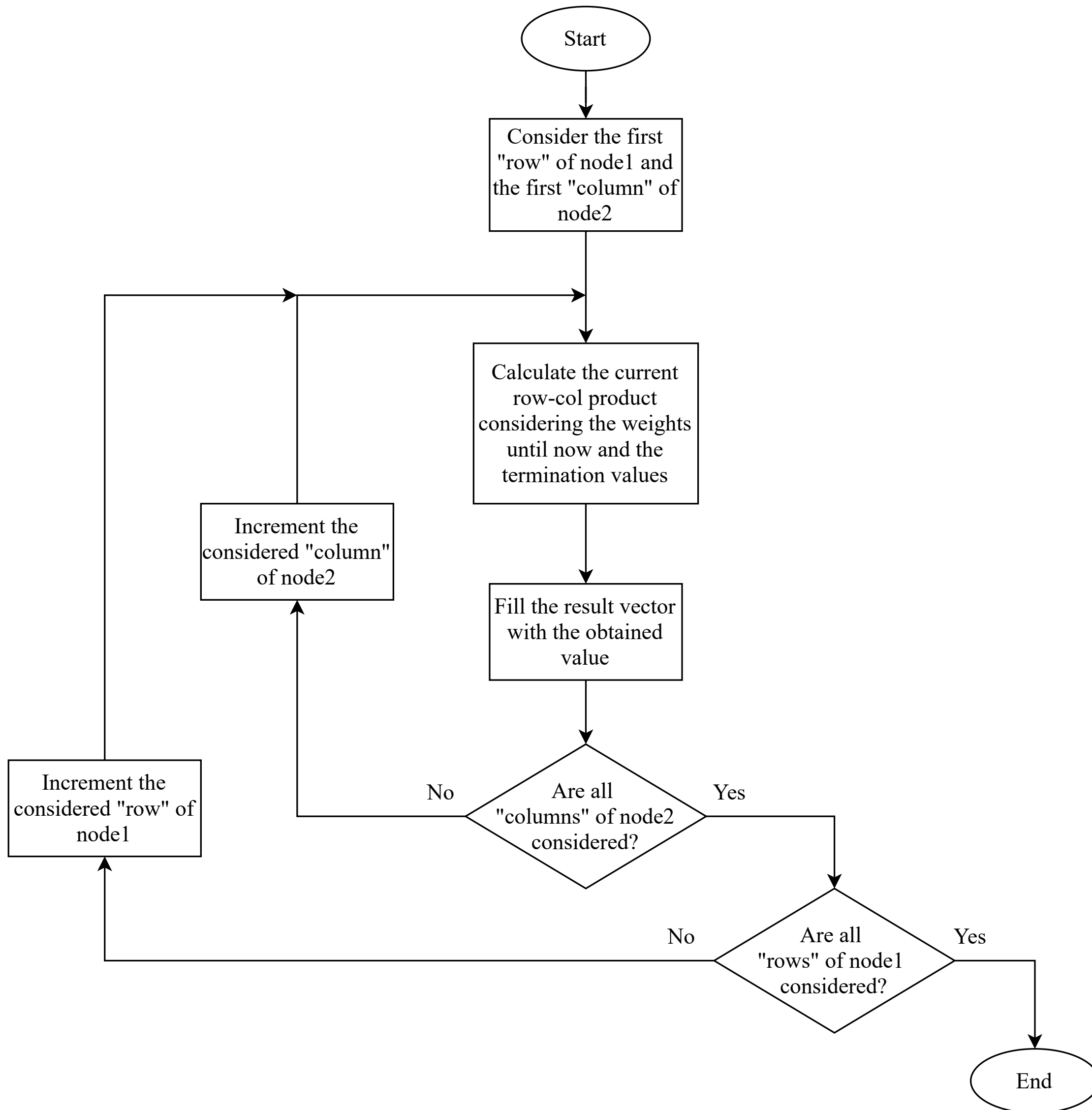


Figure 5.24: Algorithm for the calculation of the last level of the recursive row-column product between two matrices represented using Decision Diagrams.

In the described algorithms, a "result vector" is used to store the partial results obtained at every level of recursion. When recursion stops these vectors are combined to obtain a final one containing the elements of the product matrix. A proper method is then used to generate the DD representing the final matrix starting from this vector.

The recursive algorithm can be theoretically used also in the case of a product between a matrix and a vector. However, the structure of the implemented simulator does not allow it. Future improvements could be implemented to optimize this aspect.
A similar approach based on the Equation (5.2) can be used to improve also the sum between two DDs:

$$M + N = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} + \begin{bmatrix} N_{00} & N_{01} \\ N_{10} & N_{11} \end{bmatrix} = \begin{bmatrix} M_{00} + N_{00} & M_{01} + N_{01} \\ M_{10} + N_{10} & M_{11} + N_{11} \end{bmatrix} \tag{5.2}$$

This operation is not needed in case of noiseless simulation but becomes necessary when noise errors are considered (see the related Chapter 6 for more details). The algorithm used to implement the recursive sum is similar to the one reported in Figure 5.22, the only difference is how the sub-matrices are combined together (Equation (5.2) is used instead of Equation (5.1)).
Finally, two operations used only during the quantum circuit simulation can be also improved. They are the ones needed to set or reset a certain qubit to its $|1\rangle$ or $|0\rangle$ state. To do this in the standard representation (Array-based), a specific matrix ($|1\rangle\langle 1|$ or $|0\rangle\langle 0|$) must be applied to the considered qubit. In that situation, the computational cost is similar to the one needed when a generic gate is applied to the state. Instead, using the DD structure, these operations can be performed more efficiently by crossing the tree only once and modifying it properly. In particular, the tree is traversed until the level related to the considered qubit is reached and then the exiting edges of every node in that level are modified. When the qubit has to be set (reset) to $|1\rangle$ ($|0\rangle$) only the rightmost (leftmost) edge is left unchanged while all the others are set as null pointers. The described approach, summarized in Figure 5.25, can be used for both state vectors and density matrices.

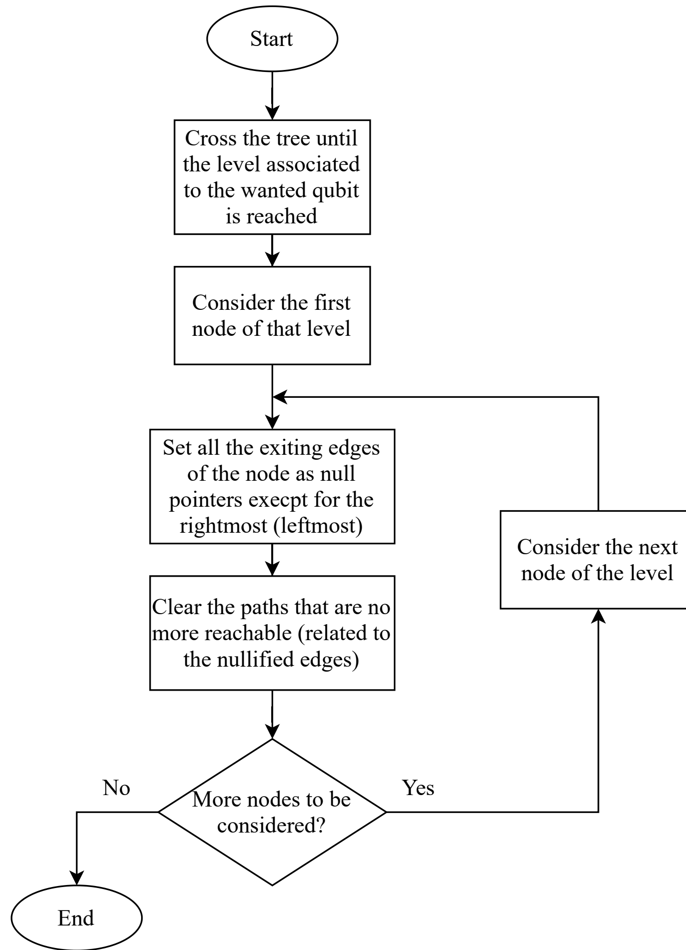


Figure 5.25: Algorithm used to set (reset) a certain qubit of the quantum state.

Chapter 6

Noisy Simulation

In this chapter, the effects related to the introduction of the noise during the simulation are introduced and analyzed. Moreover, the implemented classes used to consider the noisy simulations are described.

6.1 Noise In Simulation

The real quantum hardware is affected by non-ideality phenomena that randomly introduce errors in the state of the qubits. Quantum devices are often based on noisy registers without any quantum error correction technique. For this reason, their real behavior could be significantly different from the one described with an ideal simulation. Considering that the access to quantum hardware is still limited, carrying on an experimental analysis of these effects is problematic. So, it is important to characterize the noise in the classical simulation. Different researchers are involved in the creation of mathematical models that can be used to efficiently and reliably define the noise effects during a classical simulation [26, 27]. Today, the most used is the one based on the Kraus operators (Section 6.1.1) that, in practice, is considered a standard. However, other models are also investigated and proposed to optimize the noise management [28].
Introducing noise errors changes the simulation approach for two reasons:

- The deterministic formulation is no more valid because the noise relies on probabilistic effects;
- The system is in a mixed state (described by a density matrix) and no more in a pure state (described by a state vector).

The main consequence is that quantum states must be described using density matrices. A density matrix is a data structure used to describe a mixed state, i.e. a

collection of multiple possible states where each of them can appear with a certain semi-classical probability [2]. A particular state $|\varphi_i\rangle$ of the N possible states can appear with a certain probability p_i. The density matrix related to a certain mixed state is defined as:

$$\rho = \sum_{i=1}^{N} p_i |\varphi_i\rangle\langle\varphi_i|. \tag{6.1}$$

For a more detailed description of what a mixed state is and how it can be represented by a density matrix refer to chapter 2.4 of [2].
Until now, the quantum states were always considered as pure states and described using state vectors. A pure state can be seen as a special case of mixed state where $p_i = 1$ for a certain i and $p_j = 0$ for $i \neq j$. From now on, this description is no more sufficient and must be replaced by the use of density matrices that allow the management of mixed states. The density matrix ρ related to a generic pure state φ can be obtained as:

$$\rho = |\varphi\rangle\langle\varphi|. \tag{6.2}$$

The Equation (6.3) reports an example of the generation of a density matrix ρ starting from a pure state represented by the state vector $|\varphi\rangle$:

$$|\varphi\rangle = \left[\frac{1}{\sqrt{2}},0,0,\frac{1}{\sqrt{2}}\right]^T \rightarrow \rho = |\varphi\rangle\langle\varphi| = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix} \tag{6.3}$$

When the state is described using a state vector, the probabilities of measuring specific basic states can be obtained by squaring its elements. Instead, in the case of density matrix, they are reflected in the diagonal elements of the matrix.
A noisy circuit can be seen as a system in a mixed state affected by probabilistic effects. To consider this aspect, the simulation approach previously described in Section 3.1 must be slightly modified:

- Quantum states must be described by using density matrices;
- Quantum gates must be applied to the state considering also the noise errors related to them.

Different types of noise errors are possible and they can be modeled in different ways. The implemented simulator considers two noise models:

- the standard one based on the Kraus operators (Section 6.1.1);
- a more compact model developed by the VLSI research group of the Politecnico di Torino (Section 6.1.2) [28].

In both cases, during the simulation, the noise model is applied to the quantum state separately from the operators. In the first situation, the ideal operator is applied to the state and then the Kraus operators are considered. In this way, the quantum state has initially an ideal evolution and only after the noise errors are introduced. On the contrary, when the compact model is considered, the non-ideality phenomena are considered before the application of the gates, starting from the second one. In the noiseless simulation, the effect of a certain quantum operator on a quantum state is calculated by performing the row-column product between the vector describing the state ($|\varphi\rangle$) and the matrix U describing the operator:

$$|\varphi'\rangle = U|\varphi\rangle, \tag{6.4}$$

where $|\varphi'\rangle$ is the state after the application of the operator.
Instead, in the case of noisy simulation, the density matrix ρ' that describes the state after the application of an ideal operator U, is obtained by calculating two row-column products [2]:

$$\rho' = U\rho U^\dagger, \tag{6.5}$$

where ρ is the density matrix representing the initial state and $U^\dagger$ is the conjugate-transpose of the operator U.
It can be noticed how the usage of density matrices instead of state vectors worsens the simulation performance even without the introduction of noise errors. The simulation time is longer because the computational cost of applying an operator to a state vector is greatly increased. Moreover, more memory is necessary to store the states described as matrices instead of vectors. These drawbacks are present in both the Array-based and DD-based representations but are necessary to have the possibility of handling noise errors.

Before starting the description of the two considered noise models, it is important to underline that the worsening in both memory usage and simulation time is more evident when the noise errors are applied. This worsening partially depends on the used noise model and on how it is applied to the quantum state. However, it must be said that the DD-based representation is generally more affected. This is because noise errors reduce the redundancies inside the quantum state and so the improvements that the Decision Diagrams can offer.

6.1.1 Standard Noise Model

Physical noise errors in quantum systems are generally characterized by using a series of super-operators $\{K_1, K_2, K_3, ..., K_m\}$, called Kraus operators [29], that satisfy the Equation (6.6):

$$\sum_{i=1}^{m} K_i^\dagger K_i = I. \tag{6.6}$$

A matrix is associated with each operator and they are applied to the quantum state, described by the density matrix ρ, following the Equation (6.7):

$$\sum_{i=1}^{m} K_i \rho K_i^\dagger. \tag{6.7}$$

The most important single-qubit noises and the related operators are reported in Section 6.2.3. All of them are described by matrices associated with a certain probability p that defines the noise intensity.
In the case of two-qubits gate, the Kraus operators are obtained considering the Kronecker product between the single-qubit operators related to the two qubits of the gate. In particular, if one qubit is affected by the operators $\{K_1, K_2\}$ and the other by the operators $\{C_1, C_2\}$ the two-qubits gate is affected by the resulting operators $\{K_1 \otimes C_1, K_1 \otimes C_2, K_2 \otimes C_1, K_2 \otimes C_2\}$ [29].

6.1.2 Compact Noise Model

A compact formalism to describe relaxation and decoherence is proposed by the VLSI Lab research team of Politecnico di Torino [28]. These two dynamic non-ideality phenomena depend on the interactions between the not perfectly isolated qubits and an external environment. They are common to all technologies for quantum computing and are considered two of the main problems limiting the performance of the real hardware.
The effects of relaxation and decoherence can be described in a simplified way without the need to store all the related Kraus operators. For a single qubit the following matrix can be used for each qubit to describe both phenomena in time domain t:

$$\begin{bmatrix} (a-a_0)e^{-\frac{t}{T_1}}+a_0 & be^{-\frac{t}{T_2}} \\ b^*e^{-\frac{t}{T_2}} & (a_0-a)e^{-\frac{t}{T_1}}+1-a_0 \end{bmatrix}, \tag{6.8}$$

where a and a_0 are the probabilities of measuring the qubit in $|0\rangle\langle 0|$ for $t=0$ and $t \to \infty$, respectively, and T_1 and T_2 are the relaxation and effective decoherence time constants of the qubit.
In an n-qubit system, the model proposed in [28] can be exploited to simplify the description of decoherence and relaxation phenomena, which are considered separately. In particular, decoherence can be modeled with a matrix D defined as:

$$D = \bigotimes_{i=n-1}^{0} D_i = D_{n-1} \otimes \cdots \otimes D_0, \tag{6.9}$$

where $\bigotimes$ indicates the Kronecker product and D_i is the decoherence matrix of the ith qubit (associated with the decoherence time constant T_{2_i}):

$$D_i = \begin{bmatrix} 1 & e^{-\frac{t}{T_{2_i}}} \\ e^{-\frac{t}{T_{2_i}}} & 1 \end{bmatrix}. \tag{6.10}$$

Only the matrix D has to be applied to the state to determine the effects of decoherence on the quantum system. The element-by-element (Hadamard) product is used for this purpose.

The approach used to evaluate the total lost probability due to relaxation is a little more complex. However, a single vector $\vec{\mathbf{r}}$ with dimension 2^n, called "relaxation vector", is needed instead of a complete $2^n \times 2^n$ matrix:

$$\vec{\mathbf{r}} = \bigotimes_{i=n-1}^{0} \begin{bmatrix} 1 \\ e^{-\frac{t}{T_{1_i}}} \end{bmatrix} = \begin{bmatrix} 1 \ , \ e^{-\frac{t}{T_{1_0}}} \ , \ \cdots \ , \ e^{-\sum_{i=0}^{n-1} \frac{t}{T_{1_i}}} \end{bmatrix}^T = \\ = [r_{0,0} \ , \ r_{1,1} \ , \ \cdots \ , \ r_{2^n-1,2^n-1}]^T \tag{6.11}$$

In Equation (6.11) n is the number of qubits of the state and T_{1_i} is the relaxation time constant of the ith qubit.
The probability lost by each eigenstate $|k\rangle$ of the quantum state represented by the density matrix ρ can be calculated as $(1 - r_{k,k})\rho_{k,k}$. Moreover the Equation (6.12) can be used to calculate the total lost probability:

$$P_{\text{lost tot.}} = \sum_{k=0}^{2^n-1} (1 - r_{k,k})\rho_{k,k}. \tag{6.12}$$

After that, a certain probability amount is assigned at each eigenstate, as reported in Equation (6.13):

$$P_{\text{acquired by } |k\rangle} = \frac{w_{|k\rangle}}{\sum_k w_{|k\rangle}} P_{\text{lost tot.}}, \tag{6.13}$$

where $w_{|k\rangle}$ represents the weight associated with each eigenstate and can be calculated considering which qubits have been employed from the begin of the circuit, as reported in Equation (6.14):

$$w_{|k\rangle} = \sum_{l \in Q} \frac{1}{T_{1_l}} (1 - b_l). \tag{6.14}$$

In Equation (6.14), Q represents the set of the qubits employed at least once from the beginning of the circuit, while b_l is the binary value ({0,1}) assumed by the qubit l in the eigenstate $|k\rangle$. In this way, the states with an higher weight are the ones with a higher number of $b_l = 0$ and a lower relaxation time constants T_{1_l}.
Finally the density matrix ρ representing the quantum state is updated considering

the Equation (6.15) that affects only the main diagonal:

$$\rho_{k,k} = (1 - r_{k,k})\rho_{k,k} + P_{\text{acquired by }|k\rangle}. \tag{6.15}$$

The steps needed to evaluate relaxation can be summarized as:

1 compute the probability lost for every eigenstate;

2 compute $P_{\text{lost tot.}}$;

3 evaluate the eigenstates $|k\rangle$ affected by relaxation;

4 compute all the $w_{|k\rangle}$ and $P_{\text{acquired by }|k\rangle}$;

5 update the quantum state accordingly to Equation (6.15).

This approach can limit memory usage and, in some situations, improve the simulation time respectively to the traditional approach because only a single matrix describing the decoherence and a vector containing the effects of relaxation are needed instead of the multiple Kraus matrices (refer to Section 7.4.2 for more details).

6.2 C++ Implementation

The noisy simulation is based on the reuse of the classes implemented for the noiseless simulation with only two necessary modifications:

- The quantum states are described using two new classes (*ArrayDensityMatrix*, *DDDensityMatrix*) and no more using the *ArrayStateVec* and the *DDStateVec*;
- A new class called *NoiseModel* is implemented to define the noise model used during the simulation.

This is possible because the other classes of the simulator, such as the ones used to describe and manage the circuit (Section 3.5, Section 3.4) and its gates (Section 4.1.2, Section 5.2.3), can work with both noiseless and noisy systems.

6.2.1 *ArrayDensityMatrix* and *DDDensityMatrix* Classes

These two classes are used to represent and manage a quantum state described by its density matrix using the Array-based and the DD-based representations. They share the same external interface so that they can be used interchangeably by the simulator. However, their internal implementation is quite different and dependent on the used representation. The *ArrayDensityMatrix* class is based on the *Eigen* libraries [23]. It uses a *Dense matrix* to represent the density matrix and the different methods are mainly implemented using the functions of the library. Instead, in the *DDDensityMatrix* class, the density matrix is defined as a *DDSqMatrix* object (Section 5.2.2) and the related methods are used to manage it.
The available public methods are almost the same of the ones present in the *ArrayStateVec* (Section 4.1.1) and *DDStateVec* (Section 5.2.1) classes. This is because all of them describe a quantum state and must have a common interface to correctly interact with the other objects. The methods can be divided into four categories:

- methods to generate and modify the density matrix;
- methods to combine different density matrices together and apply operations to them;
- methods to access the information stored in the density matrix;
- methods for the overloading of the operands.

It is important to underline that, in both classes, the density matrix can be generated starting from a certain vector containing a pure state by exploiting the formula reported in Equation (6.2).
The methods of the *OperatorArray* (Section 4.1.2) and *OperatorDD* (Section 5.2.3) classes are used to apply the operators to the quantum states, as already described in the case of noiseless simulation. However, in this situation, these methods identify the presence of a mixed state and use Equation (6.5) to modify the quantum state, instead of Equation (6.4) that is correct only in the case of pure states.

6.2.2 *NoiseModel* Class

This class is used to define a noise model and apply it to a certain quantum state. It has a data structure capable of handling both the previously analyzed noise models (Section 6.1.1 and Section 6.1.2). The standard model is described by storing the related Kraus operators, while in the compact one the parameters t, T_1 and T_2 that are retrieved from a configuration file. The model can be generated in three different ways:

- By directly defining the operators that describe the effects of the noise in the standard model. This is done giving the Kraus matrices as input.
- By using one of the standard noise models stored inside the internal library (Section 6.2.3). In this case, a string identifier is stored and the Kraus operators are retrieved only when necessary (as for the library of the operators).
- By defining the configuration file from which the parameters of the compact model can be retrieved.

Considering that the standard and the compact models are based on different data structures and algorithms, the information about both of them can be present inside this class at the same time. In this way the two models can be used at the same time without the need to redefine them. Different methods are then used to apply the wanted model to a certain quantum state. In particular, a single method can be used to apply the compact model, while three different methods can be used to apply the standard model to all qubits, to multiple qubits, or to a single qubit. Obviously, before applying the noise to a certain quantum state with a specific method, the corresponding model must be previously defined. For example, it is not possible to apply the standard noise model until the related Kraus operators are properly defined.

Considering the standard noise model, different approaches can be used to apply it to a certain quantum state. In the case of multiple or all qubits, the noise can be applied to the wanted qubits altogether or one by one. When different non-ideality phenomena are considered together, the two approaches generate different results. In the first case, the noise is applied to the qubits at the same time, considering the

possible interaction between all the involved phenomena. To do this, the needed Kraus matrices are generated by calculating the Kronecker product between the single-qubit Kraus operators, as described in Section 6.1.1. Instead, in the second case, the noise is applied to the qubits separately, one after the other. The behavior is the same of applying consecutively the noise to a single different qubit. In this second situation, the final result depends also on the sequence of the qubits to which the noise is applied.
In the developed simulator it is not possible to have different noises associated with the qubits at the same time (see the Figure 6.2 for more details), so the two approaches lead to the same results. The first approach uses a more complex algorithm to consider the noise but it is applied only once after every operator. On the contrary, the second one has a more trivial implementation but multiple applications are needed, one for each qubit related to the noisy gate. For this reason, the second one is usually better when operators with a low number of qubits are considered.
An example of the application of a simple noisy CH gate (refer to Section 3.6 for more details) to the state $|10\rangle$ is analyzed to better clarify the differences between the two possible approaches. The considered noise model is described by the following Kraus operators:

$$K_1 = \sqrt{0.4}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, K_2 = \sqrt{0.6}\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix};,$$

describing a Bit-Phase flip error with probability $p = 0.4$ (refer to Section 6.2.3 for more details).
In the first described approach, four Kraus matrices are generated combining the original two, in particular the new operators are $\{K_1 \otimes K_1, K_1 \otimes K_2, K_2 \otimes K_2, K_2 \otimes K_1\}$. They are then applied to the quantum state accordingly to Equation (6.7) and the probability distribution of the final state is

$$\begin{aligned} |00\rangle &\rightarrow 0.24 \\ |01\rangle &\rightarrow 0.36 \\ |10\rangle &\rightarrow 0.16 \\ |11\rangle &\rightarrow 0.24 \end{aligned}$$

Instead, when the second approach is considered, the two original Kraus matrices are not combined together. They are applied to the state as they are but two times:

one considering the noise related to the target qubit and one considering the noise related to the control. In this way the two qubits are considered separately one after the other. As mentioned before, the final result is, in the implemented structure, the same of the one obtained with the first approach.
The choice to implement both approaches even if, in the developed simulator, they lead to the same results, was made also because in this way the structure is ready to be improved becoming capable of managing multiple noise models in two different ways.

6.2.3 Noise Model Library

To simplify and optimize their management, the Kraus operators related to the six most important single qubit noise models [29] are already stored inside the *NoiseModel* class. They are:

- **Bit Flip :** the state of a qubit randomly flips from $|0\rangle$ to $|1\rangle$ (or vice versa). It can be represented by the following matrices:

$$K_1 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ K_2 = \sqrt{1-p}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix};$$

- **Phase Flip :** the phase of the qubit changes as if a Z gate were applied. It can be represented by the following matrices:

$$K_1 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ K_2 = \sqrt{1-p}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix};$$

- **Bit-Phase Flip :** the combination of the two previous errors: the qubit flips its state and changes its phase. It can be represented by the following matrices:

$$K_1 = \sqrt{p}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \ K_2 = \sqrt{1-p}\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix};$$

- **Amplitude Damping :** model of the decay process of the qubits. It can be represented by the following matrices:

$$K_1 = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \sqrt{1-p} \begin{bmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{bmatrix};$$

- **Phase Damping :** loss of coherence between different basis states. It can be represented by the following matrices:

$$K_1 = \sqrt{p} \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{bmatrix}, K_2 = \sqrt{1-p} \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{p} \end{bmatrix};$$

- **Depolarizing :** reduction of the qubits entanglement and polarization. It can be represented by the following matrices:

$$K_1 = \sqrt{1 - \frac{3p}{4}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, K_2 = \frac{\sqrt{p}}{2} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

$$K_3 = \frac{\sqrt{p}}{2} \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, K_4 = \frac{\sqrt{p}}{2} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix};$$

The value p in the matrices represents the noise probability and can assume a value between 0 and 1, which is proportional to the noise intensity. In particular, for the first three cases, the noise is close to zero when p is close to 1. On the contrary, for the last three models, the noise is close to zero when also p is close to zero.
The *NoiseModel* class internally stores a copy of the Kraus operators related to these six noise models. When one of them is needed, the matrices are retrieved and applied to the quantum state without the need to define them by hand. In this way, only one identifier and not the complete set of Kraus operators can be used to refer to these six models.

6.3 Noisy Simulator Behavior

The general behavior of the simulator is similar in both the noiseless and noisy simulation. The main difference is that, in the second case, an appropriate noise model must be defined and used during the simulation. In the current implementation, the wanted noise model must be chosen before the simulation and it is applied to

the quantum state after that one new gate is considered. For now, it is not possible to run a complete simulation with different noise models. However, multiple partial simulations can be launched to consider different models for different sections of the circuit. This solution works but has a big loss in performance because the complete simulation must be divided into multiple less-optimized "sub-simulations" that increase the total simulation time and memory usage. This is because the configuration of the simulator and in particular the noise model has to be modified accordingly before every new "sub-simulation". For this reason, even if the simulation with multiple noise models can be theoretically performed by the simulator, no methods are implemented to automatically manage it.
Considering to work with a single standard noise model for the complete simulation, the noise is applied to the quantum state after that the effect of a gate is considered. The Figure 6.1 reports graphically the described behavior:

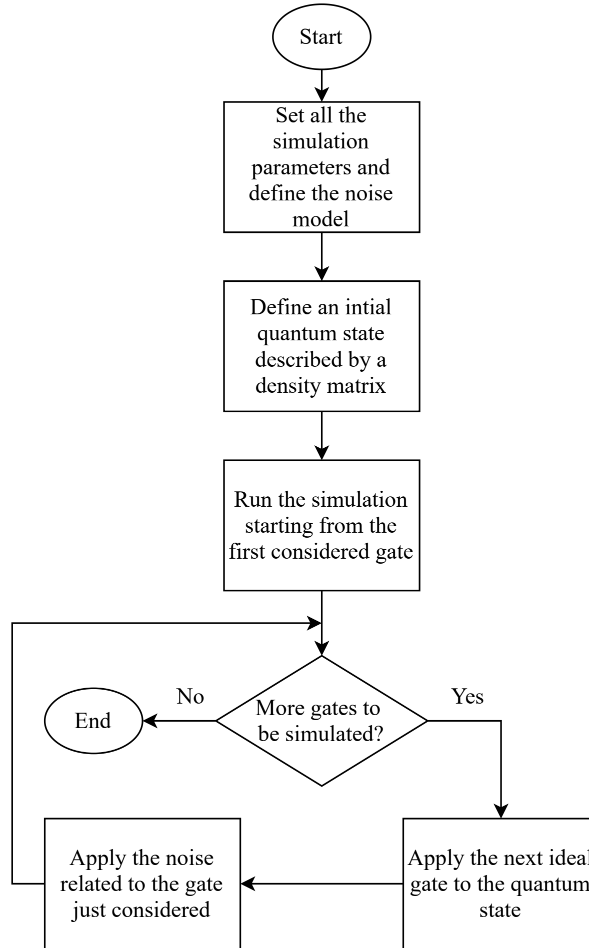


Figure 6.1: General algorithm used to simulate a noisy circuit.

When the compact noise model (Section 6.1.2) is considered the only difference is that the noise is applied before every gate except for the first one.
Considering that the noise must be applied at every gate, the optimized simulation approach described in Section 3.4.1 cannot be used. This is because that optimization is based on considering the effect of multiple gates and applying them together to the state and not one by one. Thus, the noise cannot be considered after every single gate to which it is related, but only at the end of a certain block.
Different approaches can be used to apply the noise to the quantum state. As mentioned in the description of the *NoiseModel* class (Section 6.2.2), the standard noise model can be applied to a single qubit, to multiple qubits together, or to multiple qubits one by one. Four different approaches can be chosen before starting the

simulation of a circuit:

- The noise is applied only to the qubits related to the previous gate (target(s) + control(s)) separately;
- The noise is applied only to the qubits related to the previous gate (target(s) + control(s)) altogether;
- The noise is applied to all the qubits of the state separately;
- The noise is applied to all the qubits of the state altogether;

Clearly, the last two options lead to a higher computational cost because the noise must be applied to all qubits, even the ones that are not affected by the previously considered gate. However, those approaches can be used to consider a worst-case scenario where all the qubits are always affected by the noise. This is because, in those situations, the total effect on the quantum state must be calculated by mixing together all the effects generated by the noise to all the considered qubits.
The more detailed behavior of the simulator in case of noisy simulation is reported in Figure 6.2.

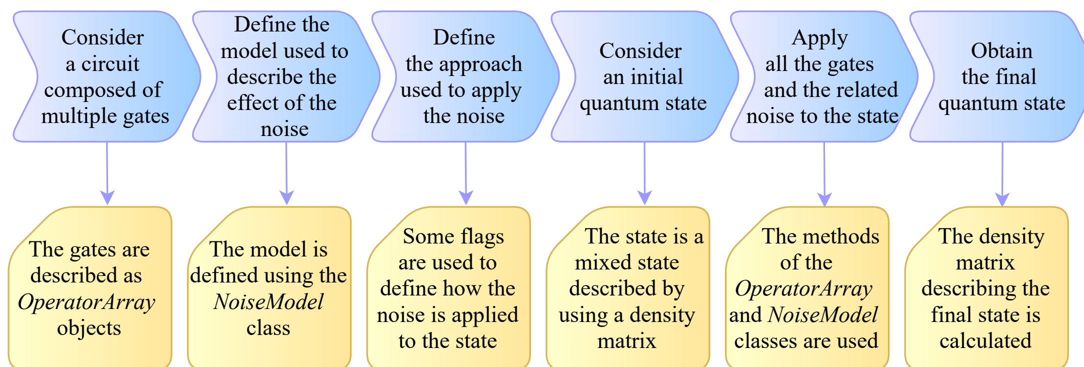


Figure 6.2: Simulator behavior during the noisy Array-based simulation.

Finally, it is important to summarize the interaction between the different implemented classes, as already done in case of noiseless simulation (Figure 4.2). Figure 6.3 reports the simulator structure in case of noisy Array-based simulation.

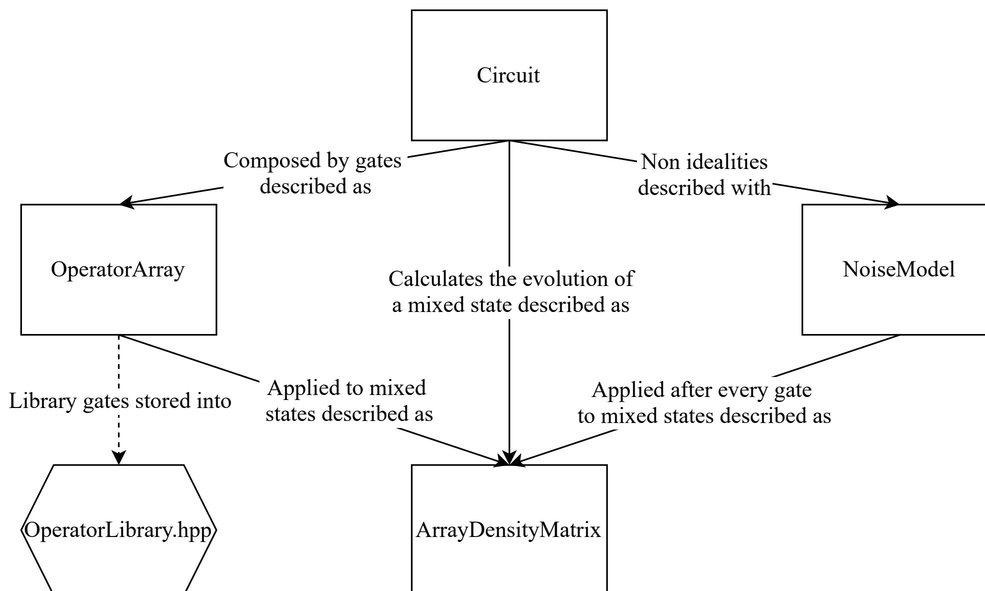


Figure 6.3: Detail of the noisy Array-based simulator structure.

As for the noiseless simulation, the *Circuit* class uses multiple *OperatorArray* objects to describe the gates of the quantum circuit. However, this time, also a *NoiseModel* object, describing the noise model to be used during the simulation, is associated with the circuit. The model must be defined before the start of the simulation and is then applied to the quantum state as reported in Figure 6.1. The quantum states are mixed states described using the *ArrayDensityMatrix* class. An *ArrayDensityMatrix* object is used to store the evolution of the state during the simulation. Every time a gate is encountered, it is applied to the state and the effect of the noise is considered using the appropriate methods of the *NoiseModel* class. As described before, the noise can be applied using different approaches depending on the simulator settings. After that all the gates and the related noises are applied, the simulation ends and the final quantum state is obtained as an *ArrayDensityMatrix* object.
In case of DD-based representation, the structure is the same but the considered classes to describe gates and states are respectively the *OperatorDD* (Section 5.2.3) and the *DDDensityMatrix*.

Chapter 7

Obtained Results

In this chapter, the results about the simulation time and the memory occupation obtained by the implemented infrastructure are reported and analyzed. Different working configurations of the simulator are considered and compared to identify their advantages and disadvantages.

7.1 Tools And Benchmarks

First of all, it is important to define the profiling tools and the benchmarks used to obtain the analyzed results. The set of considered quantum circuits is composed by:

- a circuit used to solve linear equations, working on 3 qubits and described in the `linearsolver.qasm` file;
- a simple adder working on 4 qubits and described in the `adder_small.qasm` file;
- a phase estimation circuit, working on 5 qubits described in the `phaseest.qasm` file;
- a more complex adder working on 10 qubits and described in the `adder_medium.qasm` file;
- multiple circuits implementing the Quantum Fourier Transform (QFT) with different parallelism (from 2 to 10 qubits), described in the `qft_`n`.qasm` files (n is the considered number of qubits);
- multiple circuits implementing the Inverse Quantum Fourier Transform (IQFT) with different parallelism (from 2 to 10 qubits), described in the `iqft_`n`.qasm` files (n is the considered number of qubits);

The QFT and IQFT circuits are taken from the master thesis of Luca Nurisso [30] while the others are retrieved from a repository of an existing QASM Benchmark Suite, QASMBench [31]. From now on, these circuits will be referenced in the tables and graphs by using the name of the Open QASM file describing them (without the `.qasm` extension). For example, the first-mentioned circuit on the previous list will be identified as *linearsolver*. Multiple parallelisms for the QFT and IQFT circuits are considered to analyze the performance of the same circuit working on a different number of qubits.
Various simulations with different settings were run on these circuits to profile their behavior and performance. The two considered metrics are the simulation time and the memory occupation. The first one is calculated using directly the functions of the C++ *time.h* library. It is important to underline that, in the reported results, the generation of the circuit starting from the `.qasm` file is considered part of the simulation time (for more details refer to the related Section 3.5). In particular, a timer is started at the beginning of the file reading and it is stopped at the end of the simulation, after having printed the final results. The elapsed time is considered the total simulation time.
The information about the memory occupation are obtained by using the *massif* tool of the *Valgrind* profiler [32, 33]. This tool is useful to obtain information about the trend of the allocated memory during the execution of the code (see the related Section 7.1.1 for more details). The data retrieved from its log files are then elaborated to find the average and the peak values of the memory usage that are reported in this chapter. It is important to underline that the employed tool gives information only about the dynamic memory. However, the portion of static memory used by the simulator is very limited and its contribution to the total allocation is negligible compared to the dynamic one.
All the simulations used to obtain the described results have been performed on the private server of the VLSI Lab research group at Politecnico di Torino, having the following characteristics:

- **CPU**: Intel® Xeon® Gold 6134;
- **Clock**: 3.20 GHz;
- **Cores**: 8 (however the simulations are run on a single core and thread);

- **Cache**: 24.75 MB L3 cache;

7.1.1 Massif Log File

The log file generated by the *massif* tool can be employed to obtain information about the memory allocation during the program execution. A graphical representation can be obtained by using the *ms_print* command, an example is reported in Figure 7.1.

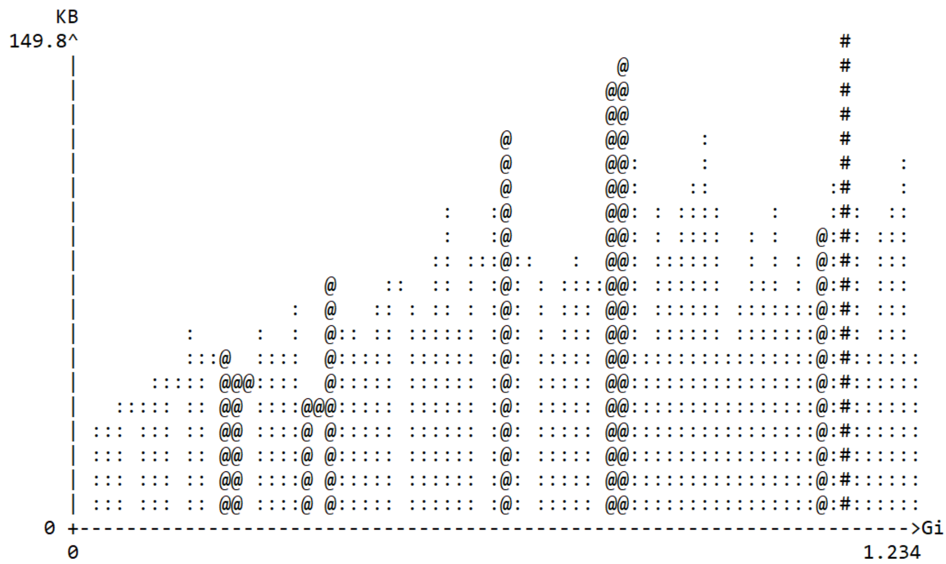


Figure 7.1: Example of the graphical representation of the memory usage trend during the program execution. On the horizontal and vertical axes are reported respectively the progression of the program and the allocated memory.

The progression of the program, in terms executed instructions, and the memory occupation are respectively reported on the horizontal and vertical axes. Vertical bars represent the measurement of memory usage at a certain time period and are called snapshots. Three different types of snapshots can be present:

- **Normal :** only basic information is reported in the log file. They are represented in the graph by using ':' characters;
- **Detailed :** more details about the allocation are reported in the log file. They are represented in the graph by using '@' characters;

- **Peak :** a detailed snapshot recording the point where the greatest memory occupation is present. It is represented in the graph by using '#' characters.

The information about the average and peak memory usage reported from now on can be obtained from these graphs and log files. The complete information about the memory occupation trend during the execution is reported only in some significant situations.

7.2 Simulator Validation

Before starting the description of the obtained results, it is important to clarify the methodology employed to check the correct behavior of the simulator. First of all, the different modules composing the simulator have been tested individually. For this purpose, specific test-benches have been adopted to validate all the implemented functionalities. Then, after having verified the correct behavior of all the developed modules, also the complete simulator infrastructure has been tested. Initially, some simple custom circuits have been employed to validate the different simulation steps. In this phase the online simulator *Quirk* [34] was used to compare the obtained results. This approach is helpful to take always under control the evolution of the state through the circuit. However, with the increase of the system complexity, it becomes time-consuming because the circuit generation cannot be automatized. For this reason, when bigger circuits are considered, the validation of the simulation outcomes has been performed by comparing them with the ones obtained with commercial simulators. In particular, the distribution of probabilities associated with the final quantum state after the simulation is compared to the one achieved with *Qiskit* framework [8].
Some differences were present in the outcomes of the tested circuits. The different simulation steps were analyzed to find the problem and it was noticed that they mainly derive from numerical errors occurred during the simulation. Indeed, while the implemented simulator employs complex amplitudes represented by using a couple of 32-bit floating point values, *Qiskit* is based on the Pyhton language that works with a different numerical precision. The errors on the final states are generally negligible in case of noiseless simulation but become more evident when

noise is introduced. This is mainly due to two reasons:

- more calculations are needed to apply noisy quantum gates to the system and so more numerical errors arise;
- the mixed quantum states affected by non-ideality phenomena have a more distributed probability and so the numerical errors combine and grow faster.

Thus, even if the outcomes of noisy simulations have usually the same trend of the the ones obtained with *Qiskit*, the final individual amplitudes can have also a quite big error. To clarify the problem an example is analyzed. The circuit reported in Figure 7.2 is simulated considering a Bit-Phase flip noise (refer to Section 6.2.3 for more details) with probability $p = 0.4$ and all the intermediate distributions assumed by the quantum state are compared with the ones obtained with *Qiskit* (Table 7.1).

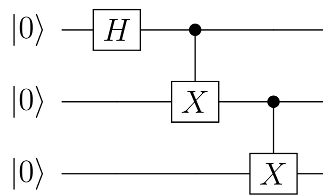


Figure 7.2: Simple quantum circuit used to analyze the effects of numerical errors inside the simulator.

State	Simulator	After the first gate	After the second gate	After the third gate
$\|000\rangle$	Implemented	0.500	0.260	0.128
	Qiskit	0.480	0.268	0.132
$\|001\rangle$	Implemented	0.500	0.240	0.132
	Qiskit	0.520	0.237	0.124
$\|010\rangle$	Implemented	0	0.240	0.120
	Qiskit	0	0.277	0.130
$\|011\rangle$	Implemented	0	0.260	0.120
	Qiskit	0	0.218	0.119
$\|100\rangle$	Implemented	0	0	0.120
	Qiskit	0	0	0.113
$\|101\rangle$	Implemented	0	0	0.120
	Qiskit	0	0	0.114
$\|110\rangle$	Implemented	0	0	0.132
	Qiskit	0	0	0.155
$\|111\rangle$	Implemented	0	0	0.128
	Qiskit	0	0	0.112

Table 7.1: Comparison between the quantum state probability distribution affected by numerical errors obtained with the implemented simulator and the correct one deriving from *Qiskit*.

There are also some rare situations where the different errors compensate themselves and so the final result is the correct one.
The described methodology has been employed to validate both the noiseless and the noisy simulation procedures, except for the situation where the compact noise model is considered (Section 6.1.2). Indeed, in that case, the amplitudes of the final quantum state were compared to the ones obtained by the MATLAB model described in [28], detecting again some differences related to the numerical errors.
All the problems related to the discrepancies between the simulation outcomes and the correct ones could be solved by increasing the simulator precision or by applying some error correcting algorithms that are not present in this implementation.

7.3 Ideal Simulation Results

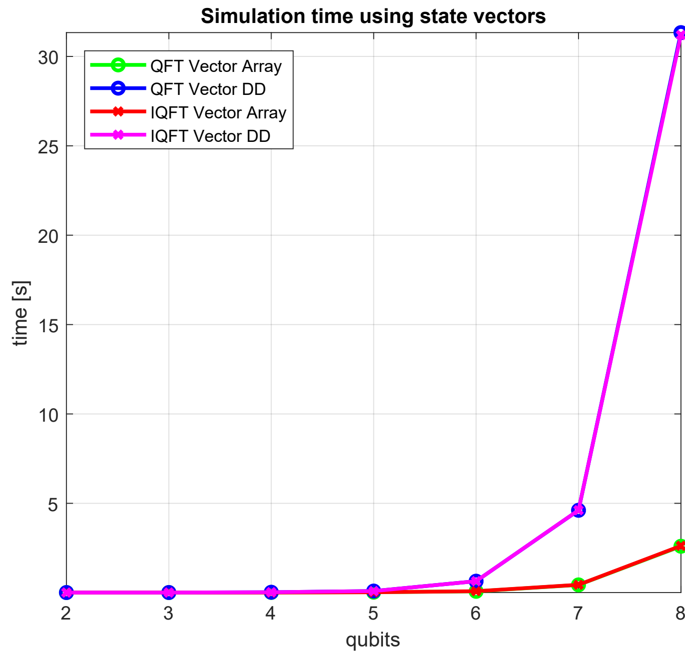
First of all, the noiseless simulation based on state vectors is considered for both the Array-based and the DD-based representations. Table 7.2 reports the obtained

results concerning the simulation time and the memory occupation for some of the analyzed circuits. They are ordered placing at the top of the table the circuit with the smaller parallelism and at the bottom the one with the larger one.

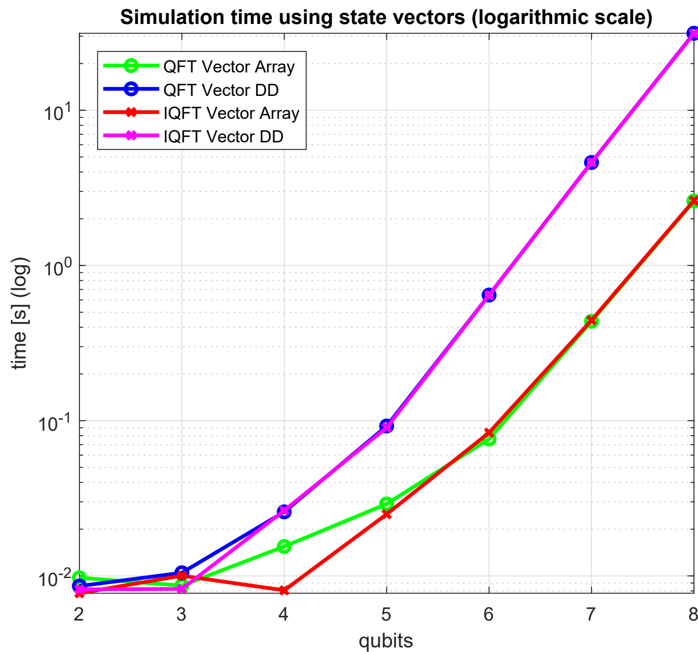
Circuit	**Simulation Time [s]**		**Average Memory Occupation [KiB]**	
	Array	**DD**	**Array**	**DD**
linearsolver	7.330×10^{-3} s	8.861×10^{-3} s	21.240 KiB	24.264 KiB
qft_4	1.547×10^{-2} s	2.586×10^{-2} s	21.240 KiB	22.622 KiB
iqft_4	8.071×10^{-3} s	2.638×10^{-2} s	21.184 KiB	22.621 KiB
adder_small	1.188×10^{-2} s	4.450×10^{-2} s	23.807 KiB	25.752 KiB
phaseest	1.007×10^{-1} s	7.680×10^{-1} s	45.363 KiB	45.470 KiB
qft_7	4.367×10^{-1} s	4.610 s	127.62 KiB	32.065 KiB
iqft_7	4.438×10^{-1} s	4.617 s	136.66 KiB	32.399 KiB
adder_medium	1.648×10^{2} s	2.187×10^{3} s	7061.1 KiB	71.341 KiB

Table 7.2: Obtained results from the simulation of generic quantum circuits considering noiseless simulation using state vectors.

When small parallelism is considered, the results obtained with the two representations are very similar. With the increase of the number of considered qubits, the Decision Diagrams become much slower than the standard Array representation but their memory usage is greatly reduced. This trend is more evident in the QFT and IQFT circuits, where more redundancies are present and so a more compact representation can be generated. Figure 7.3 and Figure 7.4 report respectively the trend of simulation time and memory usage with the increase of the number of considered qubits, in linear and logarithmic scale.

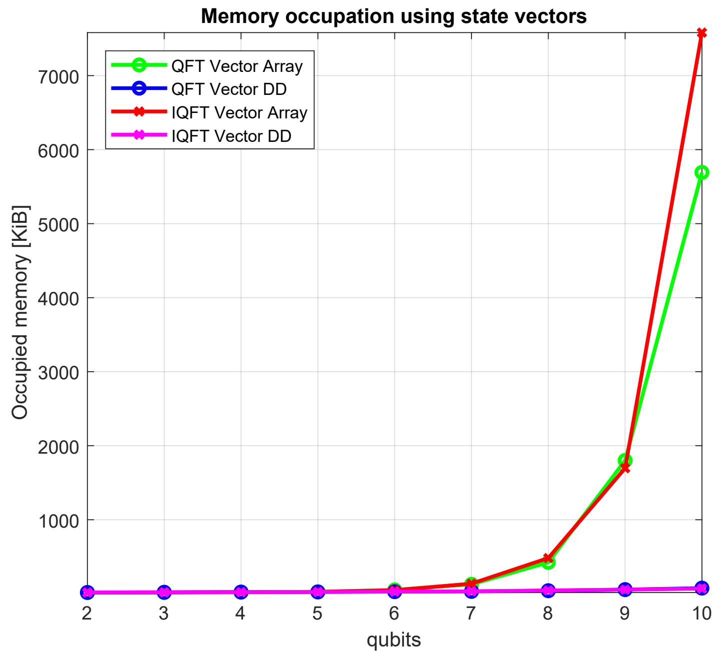


(a) Simulation time (linear scale).

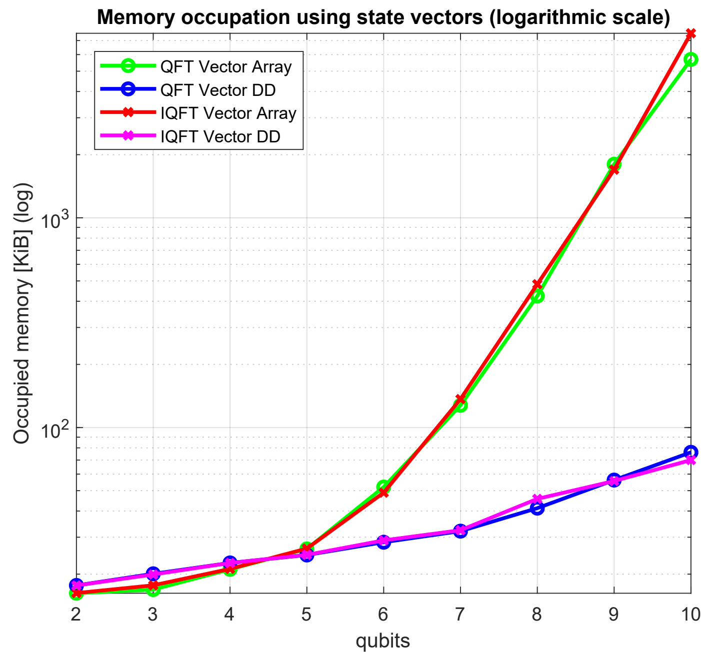


(b) Simulation time (logarithmic scale).

Figure 7.3: Simulation time of the QFT and IQFT circuits depending on the considered number of qubits in the case of noiseless simulation with state vectors.



(a) Memory occupation (linear scale).



(b) Memory occupation (logarithmic scale).

Figure 7.4: Memory occupation of the QFT and IQFT circuits depending on the considered number of qubits in the case of noiseless simulation with state vectors.

The increase of simulation time is more gentle when the Array-based representation is considered. On the contrary, the memory occupation grows slower when DDs are employed. Thus, each of the two approaches can be used to optimize a specific aspect of the simulation, depending on the requirements. This trend is mainly due to the structure employed in the *DDStateVec* class (Section 5.2.1) to represent state vectors using Decision Diagrams. Indeed, the main focus was the reduction of the memory occupation taking into account an acceptable increase of the computational cost for the stored data access.
The results show also that the differences between the QFT and IQFT circuits are very limited, in particular the ones related to the simulation time (the two lines related to the same representation are almost always overlapped). This is normal in the Array-based simulation where the data structure related to a certain parallelism is similar: the dimension of the quantum states and matrices depends only on the number of considered qubits and not on the considered benchmark. Instead, the redundancies inside the quantum states exploited by the Decision Diagrams are usually more affected by the characteristics of the simulated circuit. In this situation, the differences are minimized because the QFT and the IQFT are both quite redundant algorithms. However, in general, the DDs are a more dynamic structure: the tree describing the quantum state changes its shape and dimension every time a gate is applied to it and the amplitudes are modified.
Figure 7.5 and Figure 7.6 report the trend of the memory usage obtained with the *massif* tool (Section 7.1.1) for the *iqft_8* circuit considering respectively the noiseless Array-based and the DD-based simulation.

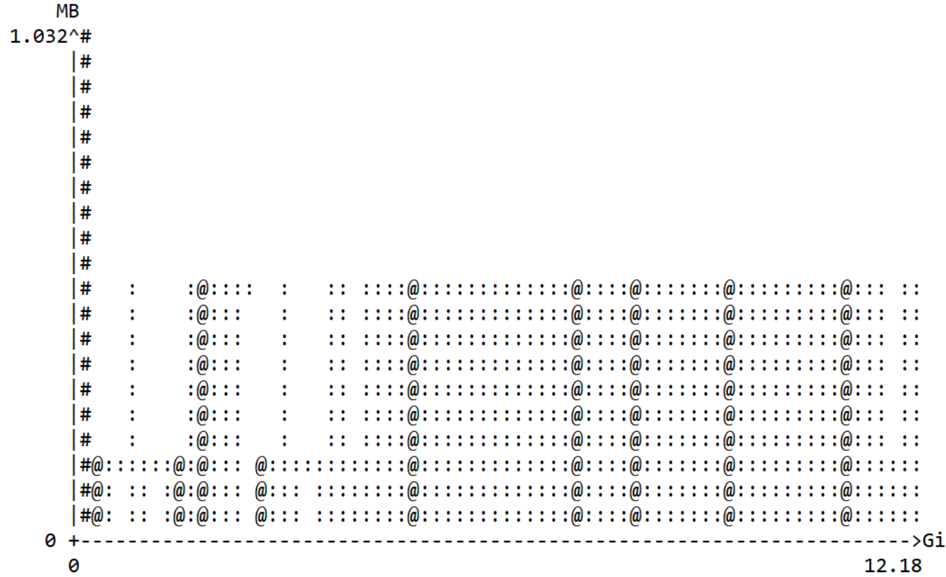


Figure 7.5: Trend of memory usage in the noiseless simulation of the *iqft_8* circuit using the Array-based state vectors.

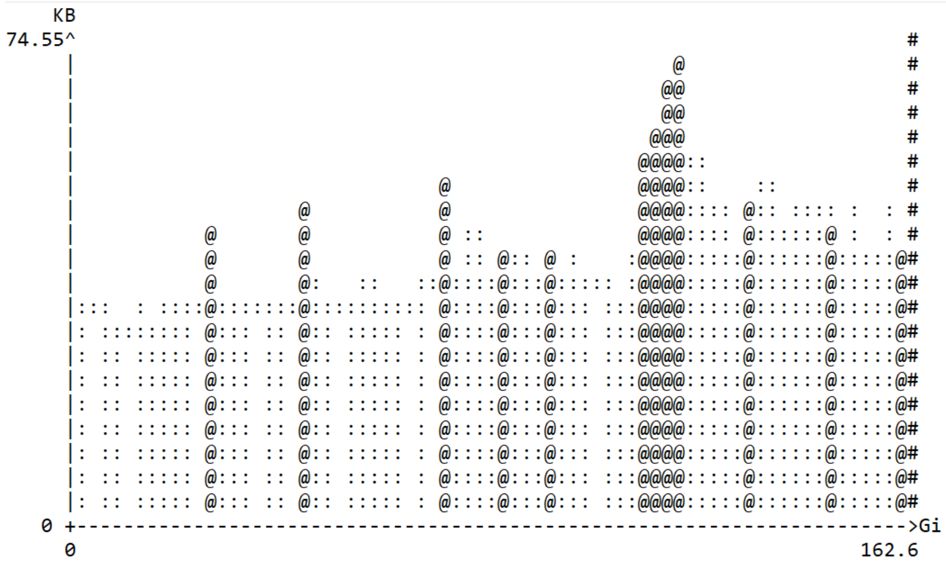


Figure 7.6: Trend of memory usage in the noiseless simulation of the *iqft_8* circuit using the DD-based state vectors.

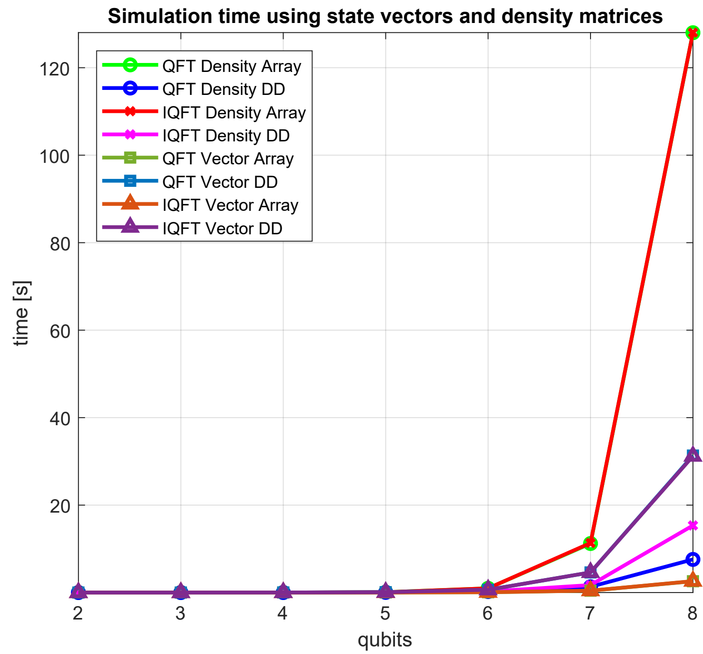
The memory usage in the simulation performed considering the Array-based representation is quite constant and only a few “holes” are present. Indeed, the

allocated memory to store the quantum states and gates is always the same and does not depend on the values assumed by their elements. Instead, in the DD-based simulation, the behavior is more irregular because the data structure is more dynamic, and so the allocated memory varies a lot.
Some simulations have been also performed considering working with density matrices but still referring to noiseless circuits. That situation is very uncommon because, normally, using matrices instead of vectors worsens the performance of the simulator. However, the obtained results, reported in Table 7.3, can be used to highlight the possible advantages of employing matrices represented by Decision Diagrams.

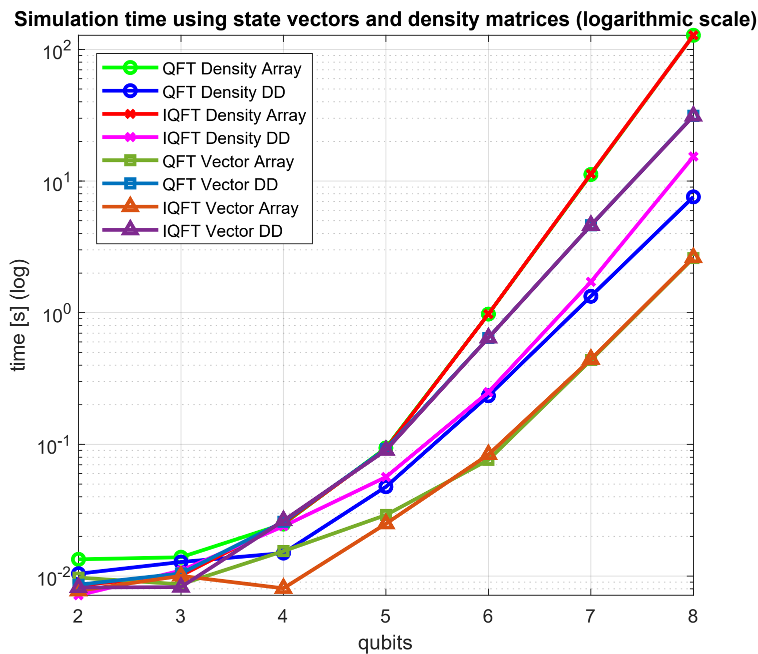
Circuit	**Simulation Time [s]**		**Average Memory Occupation [KiB]**	
	Array	**DD**	**Array**	**DD**
linearsolver	2.018×10^{-2} s	1.246×10^{-2} s	24.211 KiB	23.581 KiB
qft_4	2.484×10^{-2} s	1.495×10^{-2} s	31.872 KiB	24.593 KiB
iqft_4	2.520×10^{-2} s	2.390×10^{-2} s	29.756 KiB	24.599 KiB
adder_small	2.547×10^{-2} s	2.199×10^{-2} s	35.547 KiB	30.330 KiB
phaseest	6.912×10^{-1} s	1.394 s	95.696 KiB	155.24 KiB
qft_7	1.147×10^{1} s	1.317 s	936.21 KiB	134.66 KiB
iqft_7	1.149×10^{1} s	1.733 s	836.41 KiB	156.43 KiB
adder_medium	2.743×10^{3} s	2.170×10^{2} s	74 981.3 KiB	5365.5 KiB

Table 7.3: Obtained results from the simulation of generic quantum circuits considering the noiseless simulation using density matrices.

Differently from the previous analysis (Table 7.2), the obtained results show that the DD-based representation is the optimal choice in almost all the situations, regarding both the simulation time and the memory occupation. The only exception is the *phaseest* circuit. This is mainly because matrices are usually more redundant than vectors, and so the resulting DDs are more compact. However, it is important to compare these outcomes with the ones obtained using state vectors and identify the best configuration for noiseless simulations. In Figure 7.7 and Figure 7.8, the results obtained with the QFT and IQFT circuits are employed for this purpose.

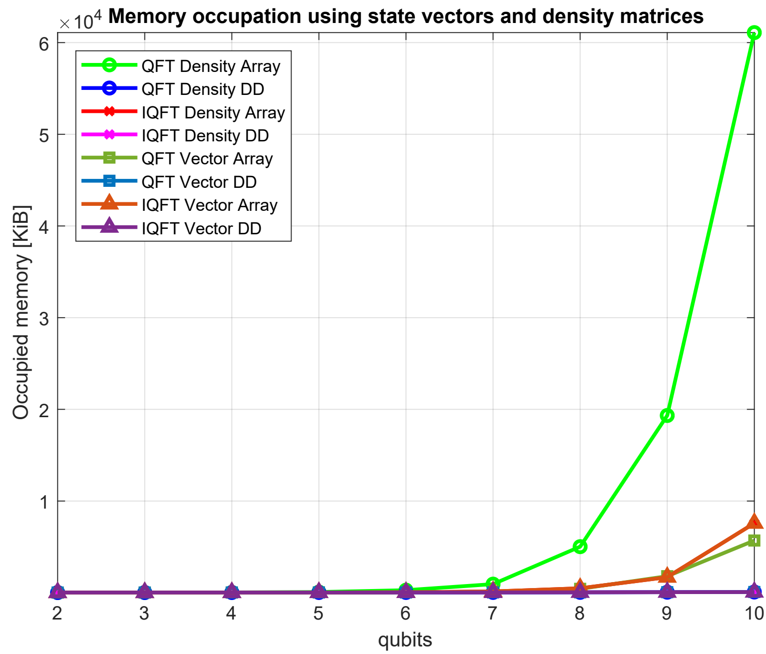


(a) Simulation time (linear scale).

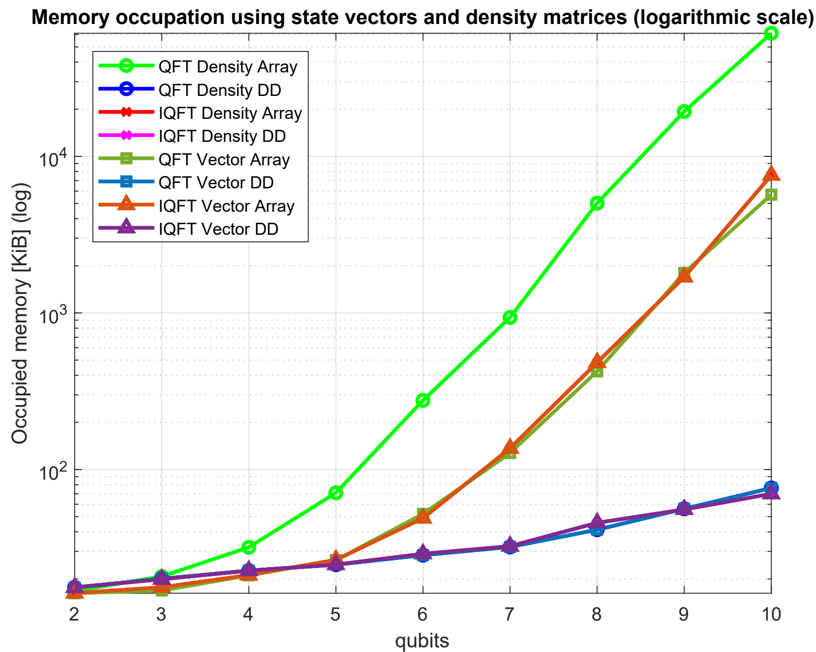


(b) Simulation time (logarithmic scale).

Figure 7.7: Comparison between the simulation time of the QFT and IQFT noiseless circuits using state vectors and density matrices.



(a) Memory occupation (linear scale).



(b) Memory occupation (logarithmic scale).

Figure 7.8: Comparison between the memory occupation of the QFT and IQFT noiseless circuits using state vectors and density matrices.

From the plots, it can be noticed that the overall fastest simulation is again obtained with the Array-based representation using state vectors. However, good results are also achieved considering to work with density matrices represented using Decision Diagrams. In that situation, the simulation time is also lower than the one obtained by using DD state vectors. This is because the structure implemented to represent matrices (Section 5.2.2) is generally more optimized than the one used for the vectors (Section 5.2.1). Concerning the memory occupation, the DDs are again the best choice.

Also in this case, the trend of memory allocation can be analyzed for both the Array-based (Figure 7.9) and the DD-based (Figure 7.10) representations.

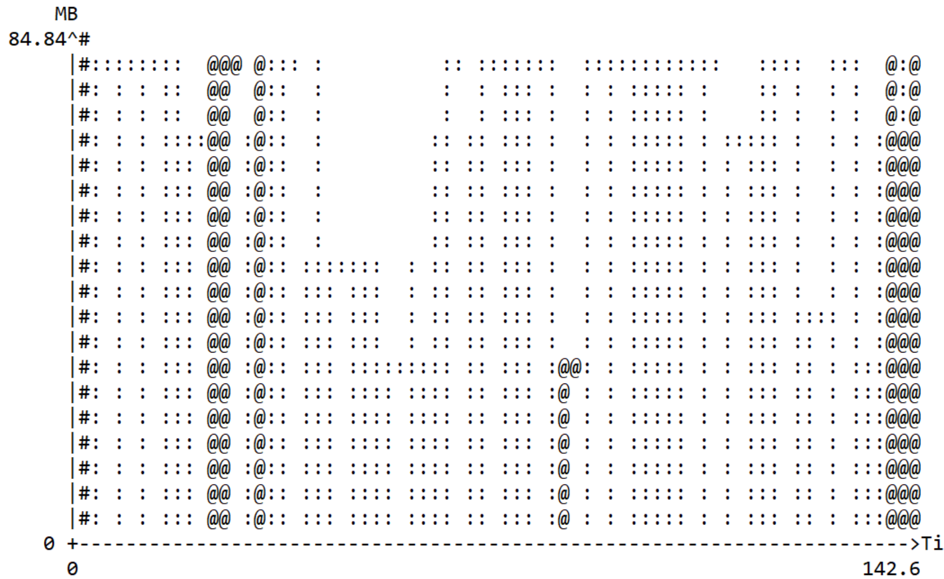


Figure 7.9: Trend of the memory usage in the noiseless simulation of the *adder_medium* circuit using the Array-based density matrices.

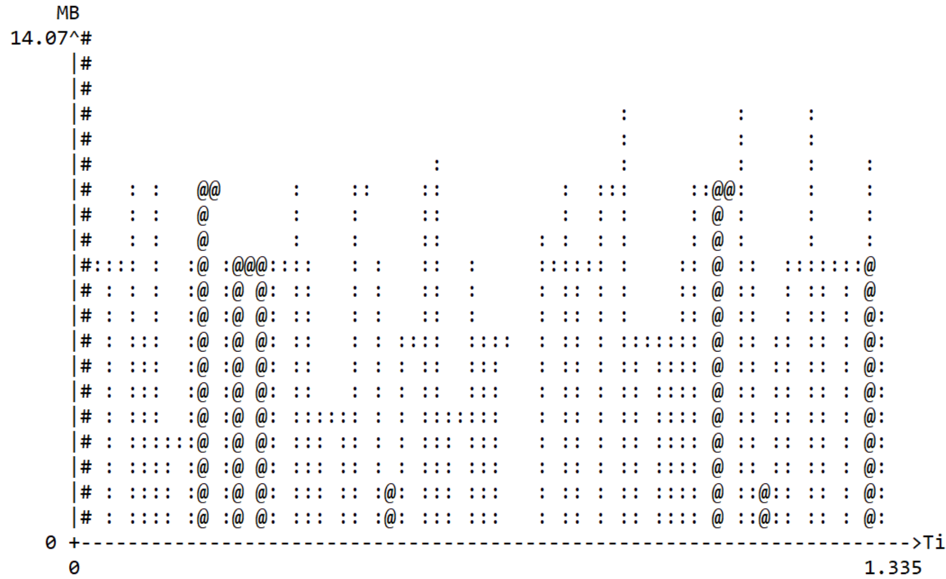


Figure 7.10: Trend of the memory usage in the noiseless simulation of the *adder_medium* circuit using the DD-based density matrices.

Decision Diagrams show again a more irregular trend compared to the one obtained with the Array-based simulation. This is again because their structure is dynamic and varies a lot during the simulation.
In conclusion, the analysis performed considering to work with density matrices, even if they are not necessary for noiseless simulations, shows that the Decision Diagram structure is more optimized in this situation. For this reason, they seem the optimal choice in the case of noisy simulation. However, it must be taken into account that noise reduces the redundancies inside the quantum states and so the benefits of the DD-based representation. A more detailed analysis is performed in the related Section 7.4.

7.3.1 “Condensed Gate” Simulation Results

Before starting the analysis of the results obtained in case of noisy simulation, it is necessary to briefly describe the benefits obtained using the optimized simulation procedure described in Section 3.4.1. The key idea of this optimization is to merge some non-overlapping gates and simulate them together in the same simulation step. This approach can reduce the computational cost and so the time needed for the

simulation. However, the improvements are strongly dependent on the circuit characteristics: many consecutive non-overlapping gates must be present to have a good optimization. Moreover, it must be considered that a small increase of the computational cost is present during the creation of the circuit when this optimization is considered. This overhead is usually negligible compared to the total simulation time.
Table 7.4 reports the comparison between the simulation time of some generic circuits with and without the described optimization. The information about memory usage are not reported because this approach does not affect it in a significant way.

Circuit	**Simulation Time With Optimization**		**Simulation Time Without Optimization**	
	Array	**DD**	**Array**	**DD**
linearsolver	1.68×10^{-2} s	1.20×10^{-2} s	1.15×10^{-2} s	1.80×10^{-2} s
qft_4	6.41×10^{-3} s	1.68×10^{-2} s	8.02×10^{-3} s	2.32×10^{-2} s
iqft_4	1.00×10^{-2} s	2.02×10^{-2} s	1.53×10^{-2} s	1.93×10^{-2} s
adder_small	1.45×10^{-2} s	3.69×10^{-2} s	1.26×10^{-2} s	3.86×10^{-2} s
phaseest	1.01×10^{-1} s	7.55×10^{-1} s	1.02×10^{-1} s	8.01×10^{-1} s
qft_7	4.15×10^{-1} s	4.52 s	4.44×10^{-1} s	5.41 s
iqft_7	4.36×10^{-1} s	4.61 s	4.40×10^{-1} s	5.36 s
adder_medium	1.36×10^{2} s	1.99×10^{3} s	1.65×10^{2} s	2.19×10^{3} s

Table 7.4: Obtained results from the noiseless simulation of generic quantum circuits considering the “condensed gate” optimization.

The results show that the optimized approach is better only when more than 4 qubits are considered, while it is generally worse in the benchmarks with smaller parallelism. This is mainly because, in the circuits with a reduced number of qubits, it is difficult to have a situation where multiple non-overlapping gates are present. Moreover, in those situations, the simulation time is short and the overhead due to the optimization is more evident.

7.4 Noisy Simulation Results

In the case of noisy simulation, multiple results have been obtained for the different possible noise models. However, in this section, only the most meaningful results

are reported. In particular, they are the ones related to:

- Bit-Phase Flip error with probability $p = 0.3$, chosen because it can be modeled easily;
- Depolarizing error with probability $p = 0.3$, chosen because it is based on a computationally expensive model;
- Decoherence and Relaxation errors, chosen because they are very common in quantum hardware and can be described by using the compact model.

The effect of the noise is always considered together with the application of every gate. In the first and second situation, the Kraus operators are employed to describe the noise (Section 6.1.1). At each simulation step, they act only on the qubits related to the previous applied gate, one by one. In the third case, the compact noise model (Section 6.1.2) is considered.

7.4.1 Standard Noise Model Results

Table 7.5 reports the obtained results for the simulation time and the memory occupation considering the Phase-Bit Flip error.

Circuit	Simulation Time [s]		Memory Occupation [KiB]	
	Array	**DD**	**Array**	**DD**
linearsolver	1.113×10^{-3} s	3.121×10^{-2} s	26.687 KiB	48.923 KiB
qft_4	3.365×10^{-2} s	4.308×10^{-2} s	35.871 KiB	30.439 KiB
iqft_4	2.736×10^{-2} s	4.069×10^{-2} s	36.366 KiB	33.446 KiB
adder_small	4.001×10^{-2} s	8.002×10^{-2} s	40.140 KiB	56.557 KiB
phaseest	1.230 s	1.669×10^{1} s	107.95 KiB	506.93 KiB
qft_7	2.166×10^{1} s	5.757 s	1688.1 KiB	577.47 KiB
iqft_7	2.172×10^{1} s	9.929 s	1688.1 KiB	648.01 KiB
adder_medium	4.749×10^{4} s	3.811×10^{5} s	78 492.85 KiB	49 543.01 KiB

Table 7.5: Obtained results from the simulation of generic quantum circuits affected by Bit-Phase Flip error.

As expected, there is a worsening of the simulator performance when the noise is considered. This is more evident in the case of DD representation because the noise

errors reduce the redundancies inside the quantum states and so their compactness. Indeed, in this situation, Decision Diagrams are the optimal choice only to simulate some quite big and redundant circuits like the *qft_7* and the *iqft_7*. In all the other circumstances they are overcome by the Array-based simulation.
Figure 7.11 and Figure 7.12 report the trend of memory occupation when the noisy simulation is considered for both the two representations.

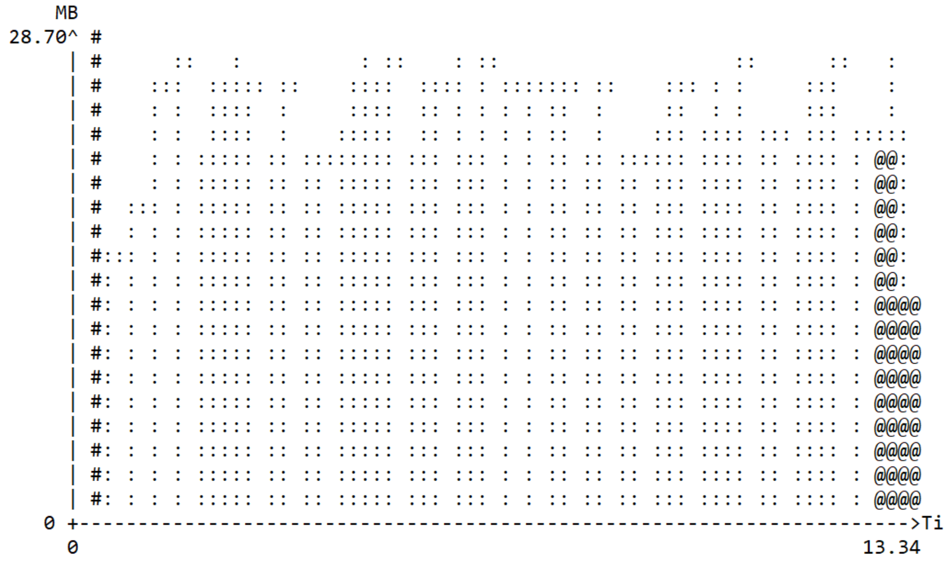


Figure 7.11: Trend of the memory usage in the noisy simulation of the *qft_9* circuit using the Array-based representation.

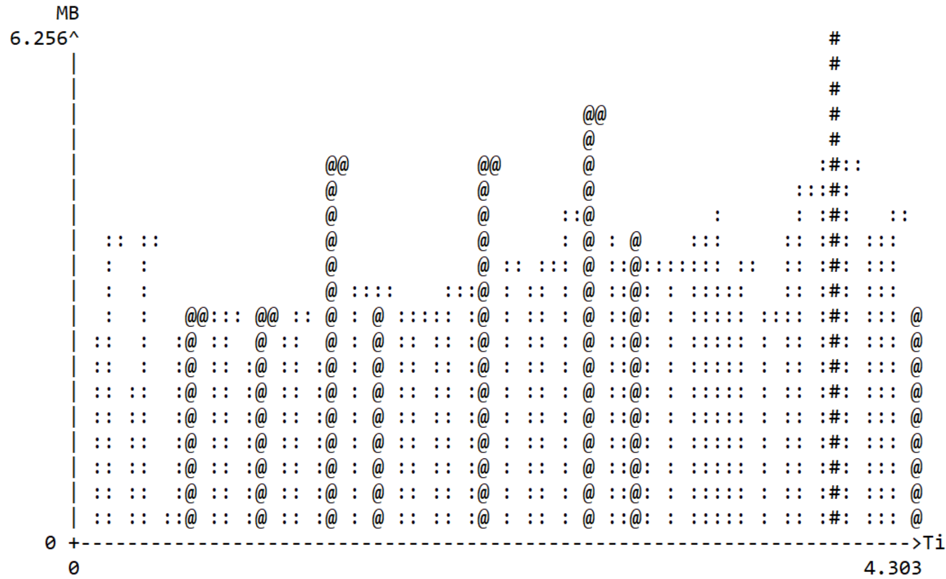


Figure 7.12: Trend of the memory usage in the noisy simulation of the *qft_9* circuit using the DD-based representation.

The allocation in the Array-based representation is, as usual, quite constant in time because the data structure needed to store quantum states and gates is fixed during the execution. On the contrary, the memory usage of the Decision Diagrams grows with the progress of the simulation. Indeed, every time that a gate and the related noise are applied to the quantum state, the redundancies inside it are reduced and the size of the DD increases.
When the Depolarizing noise is considered, there is a further performance worsening. This is because its effect must be described by using a larger number of Kraus operators and so an higher computational cost is required (for more details see the related Section 6.1.1 and Section 6.2.3). The obtained results are reported in Table 7.6.

Circuit	Simulation Time [s]		Memory Occupation [KiB]	
	Array	**DD**	**Array**	**DD**
linearsolver	2.596×10^{-2} s	8.039×10^{-2} s	30.641 KiB	97.477 KiB
qft_4	3.718×10^{-2} s	6.240×10^{-2} s	47.430 KiB	215.77 KiB
iqft_4	3.686×10^{-2} s	6.346×10^{-2} s	47.398 KiB	173.80 KiB
adder_small	8.083×10^{-2} s	2.205×10^{-1} s	51.797 KiB	149.84 KiB
phaseest	1.827 s	3.166×10^{1} s	150.14 KiB	1152.7 KiB
qft_7	3.216×10^{1} s	1.889×10^{1} s	1688.4 KiB	12 105.3 KiB
iqft_7	3.210×10^{1} s	3.385×10^{1} s	1688.4 KiB	9506.4 KiB
adder_medium	5.879×10^{4} s	4.013×10^{5} s	119 659 KiB	171 895 KiB

Table 7.6: Obtained results from the simulation of generic quantum circuits affected by Depolarizing errors.

Generally, the DD representation is worse considering both the memory usage and the simulation time. This is due to the reduction of the redundancies inside the quantum states: Decision Diagrams become bigger and the advantages deriving from their structure are limited. Also the IQFT circuits, which are quite redundant, perform better with the Array-based representation when this noise model is considered.
From Table 7.5 and Table 7.6, it can be noticed that the differences about the memory occupation of the Array-based simulation using the two noise models are almost null. Indeed, as mentioned before, the data structure required by the Array-based representation depends principally on the number of considered qubits and it is slightly affected by the circuit characteristics. For this reason, using that representation, the memory usage is almost the same for all the possible non-ideality phenomena described with the "standard model" (Section 6.1.1). Instead, the DD-base simulation is greatly affected by the circuit topology and the considered noise errors. Both the simulation time and the memory occupation considerably depend on the number of redundancies inside the quantum states. For this reason, the obtained results for this representation are very different in the two considered noisy simulations.

7.4.2 Compact Noise Model Results

The last considered noise model is the one describing the effects of decoherence and relaxation. Table 7.7 reports the results obtained when the compact noise model (Section 6.1.2) is considered during simulation to describe the effects of these non-ideality phenomena. In this situation only the small benchmarks with less than 5 qubits are considered because the parameters of the model are available only for these circuits.

Circuit	Simulation Time [s]		Memory Occupation [KiB]	
	Array	DD	Array	DD
linearsolver	1.441×10^{-2} s	1.491×10^{-2} s	28.84 KiB	40.71 KiB
qft_4	3.120×10^{-2} s	3.193×10^{-2} s	45.52 KiB	30.31 KiB
iqft_4	1.548×10^{-2} s	1.294×10^{-1} s	45.49 KiB	25.58 KiB
adder_small	3.365×10^{-2} s	1.351×10^{-1} s	46.88 KiB	102.5 KiB

Table 7.7: Obtained results from the simulation of generic quantum circuits affected by decoherence and relaxation errors described by the compact model analyzed in Section 6.1.2.

The simulation times are generally lower than the ones obtained considering the depolarizing error while are higher than the ones concerning the Bit-Phase flip noise. Moreover, also the memory occupation is lower in all the considered benchmarks. However, his comparison is not so meaningful because the considered non-ideality phenomena are different. These results can be used only to have an idea of the performance of the compact model.

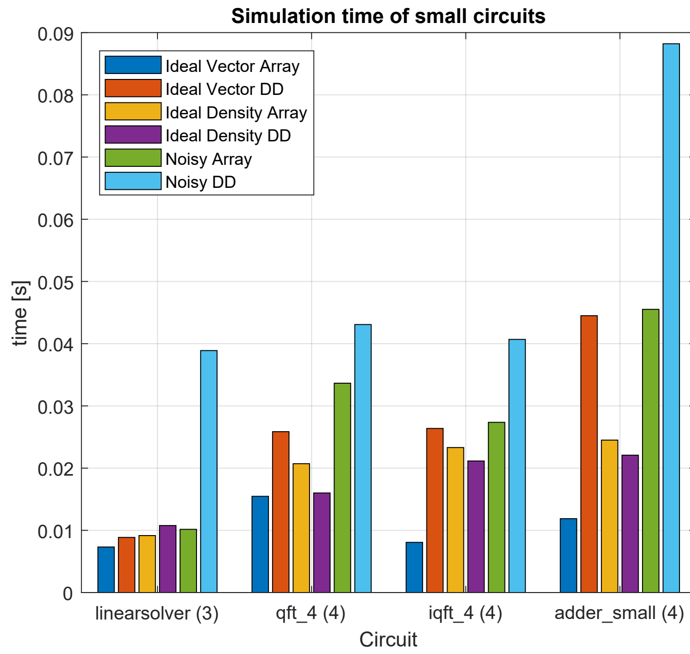
7.5 Overall Comparison

To conclude this analysis, it is necessary to consider an overall comparison between the most important results obtained until now. Six different configurations are compared, in particular:

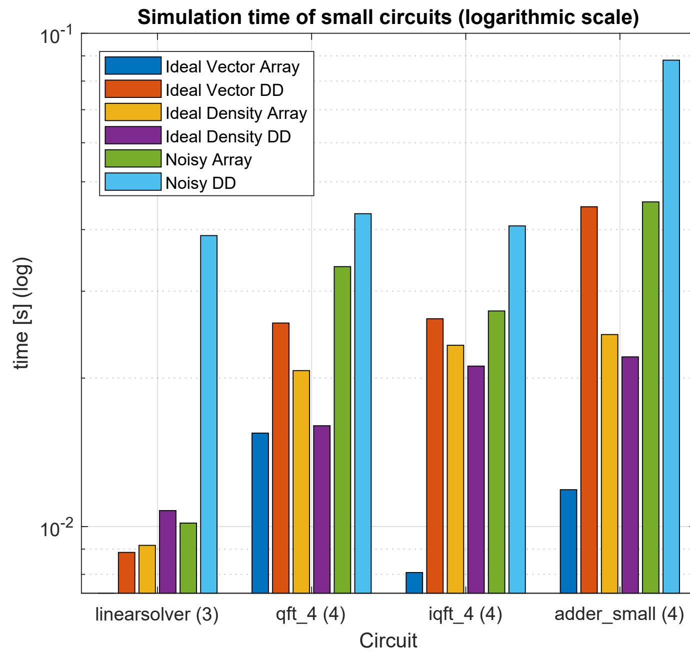
- Ideal simulation using state vectors, with both the Array-based and the DD-based representation;

- Ideal simulation using density matrices, with both the Array-based and the DD-based representation;
- Noisy simulation considering a Bit-Phase Flip noise with $p = 0.3$, with both the Array-based and the DD-based representation;

In all situations, the same set of circuits are used as benchmarks, divided into two categories depending on their parallelism. The "small circuits" are the ones composed of less than 5 qubits, while the "intermediate circuits" are the ones with 5 to 10 qubits. Figure 7.13 and Figure 7.14 report the comparison between the six different configurations for the small and intermediate set of circuits.

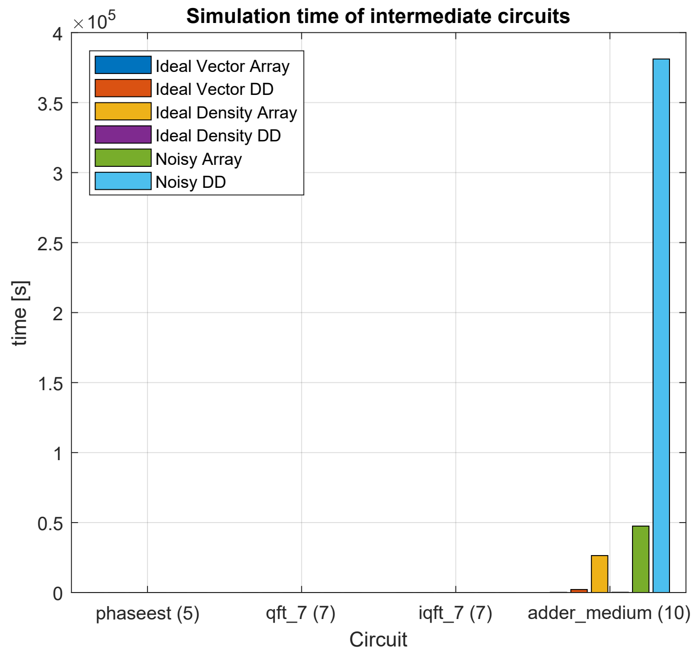


(a) Linear scale.

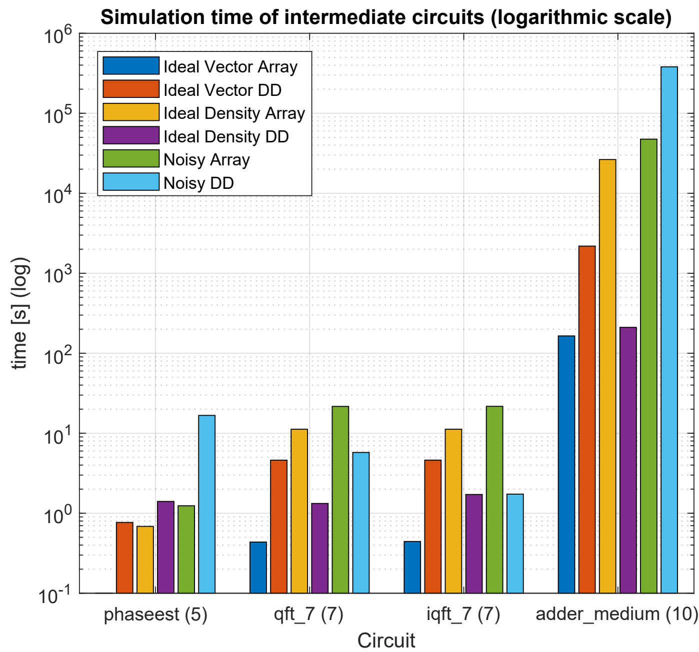


(b) Logarithmic scale.

Figure 7.13: Comparison between the simulation time of small circuits considering different possible configurations.



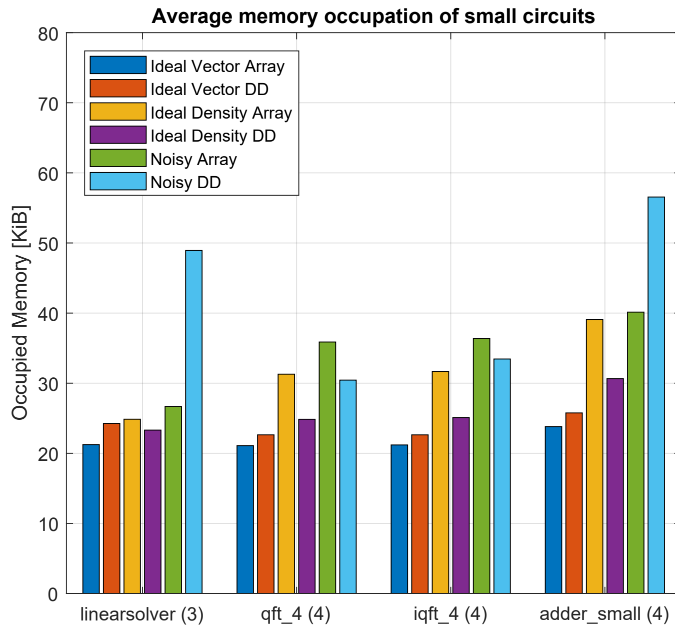
(a) Linear scale.



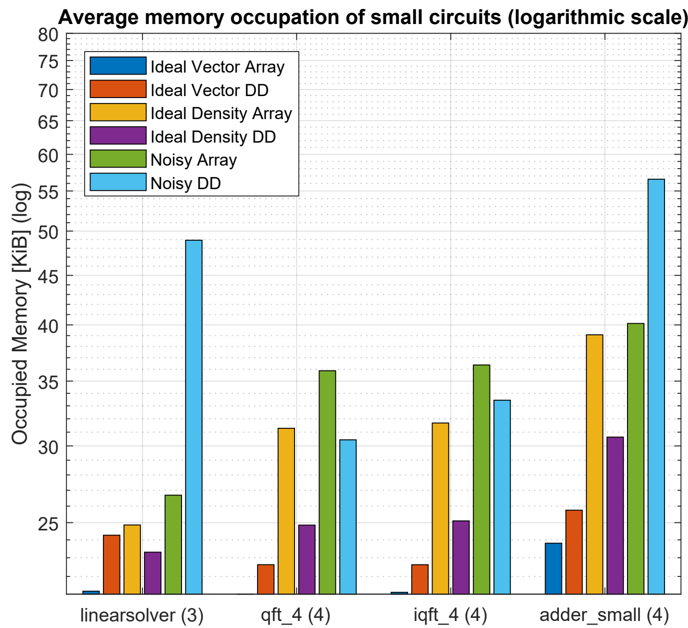
(b) Logarithmic scale.

Figure 7.14: Comparison between the simulation time of intermediate circuits considering different possible configurations.

Concerning the ideal simulation, the DDs obtain better results when density matrices are employed to describe noiseless quantum states but they cannot compete with the Array-based state vectors. Indeed, these last minimize the execution time, independently from the simulated circuit. However, when noise is introduced, the situation changes a little. If small circuits are considered, the Array-based representation is still optimal but its advantages are strongly reduced. Moreover, the Decision Diagrams become the most advantageous choice for the simulation of intermediate circuits.
The same analysis can be performed also considering the memory usage. The comparison between the results obtained for the small and intermediate circuits are reported respectively in Figure 7.15 and Figure 7.16.

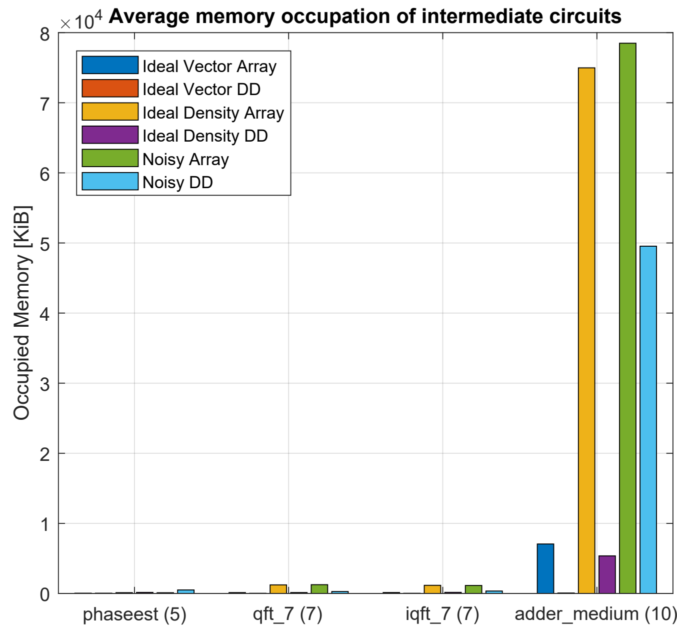


(a) Linear scale.

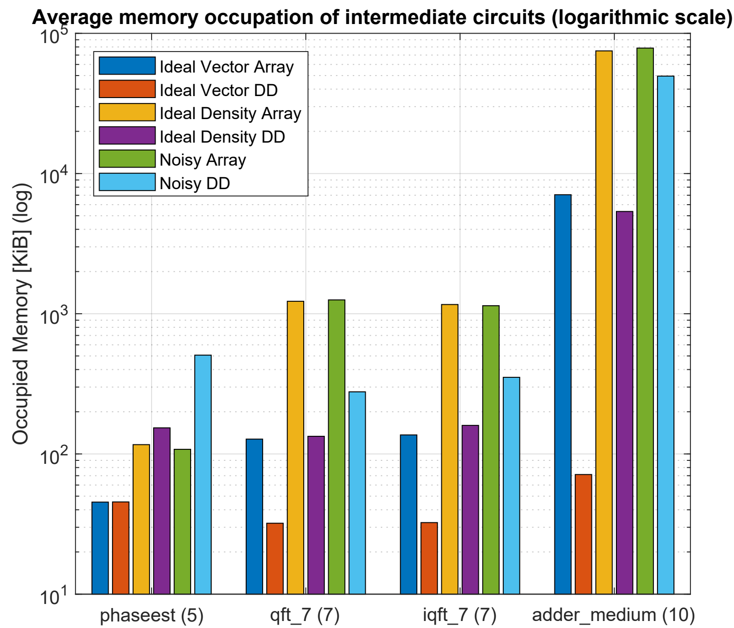


(b) Logarithmic scale.

Figure 7.15: Comparison between the average memory occupation of small circuits considering different possible configurations.



(a) Linear scale.



(b) Logarithmic scale.

Figure 7.16: Comparison between the average memory occupation of intermediate circuits considering different possible configurations.

In the noiseless simulation the use of state vectors reduces the needed resources. In particular, the Array-based representation minimizes the required memory when small circuits are considered, while the DDs achieve better results with the increase of the number of considered qubits. This is because when higher parallelism are considered the Decision Diagrams can generally exploit more redundancies and obtain a more compact structure.
From the plots it can be noticed also the difference between the memory usage when density matrices are used to simulate noiseless or noisy circuits. The introduction of the errors related to the non-ideality phenomena always increases the needed resources. However, the worsening related to the Decision Diagrams is generally more evident. Indeed, the advantages of that representations in case of noiseless simulation are maintained also when noise is introduced only in some redundant circuits, such as the ones related to the QFT and IQFT.

Chapter 8

Conclusion

In this final chapter, a summary of the achieved results is reported together with a discussion about the possible improvements and the future prospects.

8.1 Summary

The goal of this thesis was the development of a classical infrastructure capable of simulating noisy quantum circuits. This infrastructure can be employed after the logical and layout synthesis to verify the correctness of the designed circuit, taking into account also the errors and the non-ideality phenomena that affect the real quantum hardware. The simulator employs two different representations of quantum states and gates: the Array-based (Chapter 4) and the Decision Diagram-based (Chapter 5). Moreover, two different models can be used to consider the effects of the noise during the simulation (Section 6.1.1 and Section 6.1.2). The implemented structure can work with multiple configurations: different settings and parameters are used to customize the simulation procedure. First of all, the type of simulation (noiseless or noisy), the employed representation, and the noise model that has to be applied in the case of noisy circuits are chosen. Then, more specific actions can be taken to configure secondary traits of the simulator. In particular, various approaches can be considered for the circuit generation, the state measurements, the noise application, and the internal optimizations. All these features and their combinations broaden considerably the simulator workspace.
The results obtained by the simulation of simple quantum circuits show that the Array representation is generally the best one to have a faster noiseless simulation. However, when non-ideality phenomena are introduced, the optimal solution concerning the simulation time depends mostly on the considered circuit and noise

model. Regarding the memory occupation, the DD-based representation requires fewer resources in almost all noiseless simulations. This is due to the compact structure employed to represent vectors and matrices. The benefits are less evident in the case of noisy simulation because the introduction of noise errors reduces the redundancies inside the quantum states.

8.2 Possible Improvements And Future Prospects

The developed simulator is based on a quite complex modular structure with multiple integrated features. Even if it can already perform different tasks, further improvements can be implemented. Many of them can be focused on the optimization of some algorithms or simulation steps, while others can rely on the addition of new features or modules. Considering the first category, some possibilities are:

- The improvement of the Decision Diagram structure used to store state vectors, indeed, this is the less-optimized part of the simulator. In particular more redundancies can be exploited by implementing the tree reported in Figure 5.5.
- The modification of the DD structure used in the case of noisy simulation. The tree reported in Figure 5.4 can be used to minimize the computational cost when two DD matrices are combined;
- The utilization of sparse vectors and matrices in the Array-based representation to minimize the occupied memory.
- The parallelization of the code execution. Some parts of the simulation, especially in the case of Array-based representation, can be easily parallelized. In this way, the total execution time can be greatly reduced, as already demonstrated by some researches [14, 17].

Regarding the addition of new features the most important are:

- The addition of new representations for vectors and matrices. Some possibilities are based on the DDs, like the LIMDD [19] or the BDD [35], while others are more generic and optimize the operations between vectors and matrices [27].

- The introduction of new approaches to the simulation, considering also the Feynman [13], Heisenberg [15] or hybrid formalism [36, 14].
- The addition of a module capable of handling also other versions of the Open QASM language [9], such as the 3.0.
- The development of a higher-level module to manage the simulation. The python language can be used for this purpose.
- The possibility to interface with the Qiskit [8] environment.
- The development of a more interactive user interface. The implementation of a Graphical User Interface (GUI) can be also considered.

These are only a small part of the possible improvements that the implemented simulator can support. Indeed, the employed modular structure can be easily integrated with new and specialized modules, opening the doors to future optimization. This can be helpful to further broaden the simulator workspace.
Another possibility is to realize specific improvements and integrate special-purpose modules to optimize the simulation of a precise set of circuits. Until now, the implemented infrastructure is quite generic and not so specialized. This is because it was developed to be placed at the end of a toolchain dedicated to the creation of generic quantum circuits. However, if the target circuits and their technology are known in advance, there is the possibility to optimize their simulation, acting on some specific steps. On the contrary, the structure can be made even more generic using a complementary approach and integrating general-purpose modules.
In conclusion, the developed simulator is a quite generic working structure that can be used as it is or become the starting point for different types of applications and improvements.

Bibliography

[1] Noson S. Yanofsky and Mirco A. Mannucci. *QUANTUM COMPUTING FOR COMPUTER SCIENTISTS*. Cambridge University Press, 2008.

[2] Mikio Nakahara and Tetsuo Ohmi. *QUANTUM COMPUTING From Linear Algebra to Physical Realizations*. Taylor and Francis Group, LLC, 2008.

[3] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and AlÃ¡n Aspuru-Guzik. Noisy intermediate-scale quantum (nisq) algorithms, 2021. arXiv:2101.08448.

[4] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2(1), Jan 2016. URL: http://dx.doi.org/10.1038/npjqi.2015.23, doi:10.1038/npjqi.2015.23.

[5] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, Aug 2018. URL: http://dx.doi.org/10.22331/q-2018-08-06-79, doi:10.22331/q-2018-08-06-79.

[6] Richard Jozsa. Entanglement and quantum computation. *arXiv preprint quant-ph/9707034*, 1997.

[7] David P DiVincenzo. Quantum gates and circuits. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):261–276, 1998.

[8] Qiskit: An open-source framework for quantum computing, 2021. doi:10.5281/zenodo.2573505.

[9] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. arXiv:1707.03429.

[10] A. M. Krol, A. Sarkar, I. Ashraf, Z. Al-Ars, and K. Bertels. Efficient decomposition of unitary matrices in quantum circuit compilers, 2021. arXiv:2101.02993.

[11] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits.

Physical Review A, 100(3), Sep 2019. URL: http://dx.doi.org/10.1103/PhysRevA.100.032328, doi:10.1103/physreva.100.032328.

[12] Alwin Zulehner and Robert Wille. *Introducing Design Automation for Quantum Computing*. Springer Nature Switzerland AG, 2020. URL: https://doi.org/10.1007/978-3-030-41753-6.

[13] Andrew Shi. Recursive path-summing simulation of quantum computation, 2017. arXiv:1710.09364.

[14] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M. Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, and et al. Qulacs: a fast and versatile quantum circuit simulator for research purpose. *Quantum*, 5:559, Oct 2021. URL: http://dx.doi.org/10.22331/q-2021-10-06-559, doi:10.22331/q-2021-10-06-559.

[15] Simon Anders and Hans J. Briegel. Fast simulation of stabilizer circuits using a graph-state representation. *Physical Review A*, 73(2), Feb 2006. URL: http://dx.doi.org/10.1103/PhysRevA.73.022334, doi:10.1103/physreva.73.022334.

[16] Aneeqa Fatima and Igor L. Markov. Faster schrödinger-style simulation of quantum circuits, 2020. arXiv:2008.00216.

[17] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, Adrián Pérez-Salinas, Diego García-Martín, Artur Garcia-Saez, José Ignacio Latorre, and Stefano Carrazza. Qibo: a framework for quantum simulation with hardware acceleration, 2020. arXiv:2009.01845.

[18] Alwin Zulehner and Robert Wille. Matrix-vector vs. matrix-matrix multiplication: Potential in dd-based simulation of quantum computations. pages 90–95, 03 2019. doi:10.23919/DATE.2019.8714836.

[19] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. Limdd a decision diagram for simulation of quantum computing including stabilizer states, 2021. arXiv:2108.00931.

[20] Damian S. Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, Jan 2018. URL: http://dx.doi.org/10.22331/q-2018-01-31-49, doi:10.22331/q-2018-01-31-49.

[21] Microsoft. Azure qdk. Accessed on 15/10/2021. URL: https://azure.

microsoft.com/en-us/resources/development-kit/quantum-computing/.

[22] Simone Pont. Simulator documentation, Dec 2021. Accessed on 05/12/2021. URL: https://drive.google.com/drive/folders/1X7PcJsN-0fD6W3MZFi1jHZrGIF6zcKJ3?usp=sharing.

[23] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010. Accessed on 20/11/2021. URL: http://eigen.tuxfamily.org.

[24] Boost C++ libraries. Accessed on 05/11/2021. URL: https://www.boost.org/.

[25] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[26] Thomas Grurl, Jürgen Fuß, and Robert Wille. Considering decoherence errors in the simulation of quantum circuits using decision diagrams, 2020. arXiv:2012.05629.

[27] Song Cheng, Chenfeng Cao, Chao Zhang, Yongxiang Liu, Shi-Yao Hou, Pengxiang Xu, and Bei Zeng. Simulating noisy quantum circuits with matrix product density operators. *Physical Review Research*, 3(2), Apr 2021. URL: http://dx.doi.org/10.1103/PhysRevResearch.3.023005, doi:10.1103/physrevresearch.3.023005.

[28] Mario Simoni, Giovanni Amedeo Cirillo, Giovanna Turvani, Mariagrazia Graziano, and Maurizio Zamboni. Towards compact modeling of noisy quantum computers: A molecular-spin-qubit case of study. *J. Emerg. Technol. Comput. Syst.*, 18(1), 2021. doi:10.1145/3474223.

[29] Zhimin Wang, Zhaoyun Chen, Shengbin Wang, Wendong Li, Yongjian gu, Guoping Guo, and Zhiqiang Wei. A quantum circuit simulator and its applications on sunway taihulight supercomputer. *Scientific Reports*, 11, 01 2021. doi:10.1038/s41598-020-79777-y.

[30] Luca Nurisso. Scomposizione di matrici unitarie per la realizzazione di circuiti quantistici in openqasm 2.0. Dec 2021. Politecnico Di Torino.

[31] Qasmbench circuits repository. Accessed on 05/12/2021. URL: https://github.com/uuudown/QASMBench.

[32] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 89–100. ACM, 2007. URL: http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07.

[33] Valgrind website. Accessed on 20/11/2021. URL: https://valgrind.org.

[34] Gidney C and contributors. Quirk. quantum circuit simulator. Accessed on 15/10/2021. URL: https://github.com/Strilanc/Quirk.

[35] Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. Bit-slicing the hilbert space: Scaling up accurate quantum circuit simulation to a new level, 2020. arXiv:2007.09304.

[36] Lukas Burgholzer, Hartwig Bauer, and Robert Wille. Hybrid schrödinger-feynman simulation of quantum circuits with decision diagrams, 2021. arXiv:2105.07045.