Data structures for quantum circuit verification and how to compare them

Vinkhuijzen, L.T.

Chapter 2

Preliminaries

This chapter covers the background necessary for the remainder of the thesis: we introduce the basic notion of quantum computing in Section 2.2; we introduce decision diagrams, which are the primary data structures considered in this thesis, in Section 2.3; we introduce some methods for comparing data structures (in Sec. 2.3.2); lastly, we provide a brief overview of quantum model checking in Section 2.4. Each chapter contains in addition a preliminary section highlighting material for that chapter. The reader may opt to skip this chapter and instead read the specific preliminaries found in the relevant chapters.

Further reading. The interested reader is encouraged to consult the books of Nielsen and Chuang, [240] or of Kitaev, Shen and Vyalyi [182] on quantum information and computing. Wille and Zulehner's book [376] lays out the problem of design automation for quantum computing, and explains decision diagrams as a method for addressing this problem. Wegener's book gives a more broad view on decision diagrams [344]. A basic understanding of complexity theory, such as can be found in the first few chapters of the books of Papadimitriou [252] and Arora and Barak [20], will be useful to understand the analysis of our algorithms and several hardness proofs in this thesis. By far the most amusing text which covers much of the above material is Aaronson's *Quantum Computing Since Democritus* [2].

The following resources are complementary but somewhat tangential to the topics in this thesis. A good introduction to model checking can be found in Baier and Katoen [28], or in the more recent book of Clarke et al. [88]. For quantum model

checking, one may consult the book by Ying and Feng [361], or their short survey on the topic [360]. Turrini provides an accessible introduction to the topic [317]. Garwahl et al. [126] and Ferreina [118] provide excellent surveys of quantum programming languages.

2.1 Mathematical preliminaries

A *Boolean function* is a map $f\colon \{0,1\}^n \to \{0,1\}$ which associates each bit-string $x \in \{0,1\}^n$ with a bit $f(x) \in \{0,1\}$. The bits that make up this bit-string are the *input variables* and will be denoted by lower case letters, for example as $x_1, \ldots, x_n$. If $a \in \{0,1\}^\ell$ is a bit-string of length ℓ, and if $\ell \leq n$, then a *induces the subfunction* $f_a\colon \{0,1\}^{n-\ell} \to \{0,1\}$. This induced subfunction is defined by $f_a(y) = f(a,y)$ for $y \in \{0,1\}^{n-\ell}$ (sometimes we say that f_a *restricts* f to a). The string a is called a *partial assignment* to the variables of x.

Example 2.1. Consider the formula $x_1 \vee x_2$. This formula has an associated function $f\colon \{0,1\}^2 \to \{0,1\}$, which takes values $f(x_1,x_2) = x_1 \vee x_2$. This function outputs the value $f(x_1,x_2) = 0$ at the point $(x_1,x_2) = (0,0)$ and outputs $f(x_1,x_2) = 1$ everywhere else. Consider now the length-1 bit-string $a = 0$. This partial assignments induces the subfunction $f_a\colon \{0,1\}^1 \to \{0,1\}$. This function takes the values $f_a(x_2) = x_2$ for $x_2 \in \{0,1\}$.

This example also teaches us that, although a formula is associated with a function, the two are not the same thing; notably, two different formulas may effect the same function, e.g., the formula $x_1 \vee x_2$ has the same truth values as the formula $x_1 \vee (x_1 \wedge x_2)$.

We will often make use of the *Shannon decomposition* of a function [288]. For any function f, and a single Boolean variable x_1, its Shannon decomposition with respect to x_1 is the following expression,

$$f(x_1, x_2, \ldots, x_n) = \neg x_1 \wedge f_0(x_2, \ldots, x_n) \;\vee\; x_1 \wedge f_1(x_2, \ldots, x_n) \tag{2.1}$$

This construction can be applied recursively to the $n-1$-variable functions f_0 and f_1. The usefulness of the Shannon decomposition lies in that it allows us both to break down a function into its constituent parts (as is done in Equation 2.1); and to build a function from smaller functions on fewer variables (namely, in a scenario in which we have two functions f_0 and f_1, we can use Equation 2.1 to build the larger function f).

A function of the form $f\colon \{0,1\}^n \to \mathbb{C}$ is called a *pseudo-Boolean* function. The concepts of induced subfunctions and Shannon decompositions are applicable here just as to Boolean functions.

For a set A of group elements, we write $\langle A \rangle$ to denote the group generated under multiplication. For example, consider a set of two matrices, $A = \{\left[\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right], \left[\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right]\}$. The group $\langle A \rangle$ generated by this set is

$$\langle A \rangle = \left\{\left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right], \left[\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right]\right\} \tag{2.2}$$

Given an $n \times m$ matrix A, and a $k \times \ell$ matrix B, the *tensor product* of matrices A and B, denoted $A \otimes B$, is the following $nk \times m\ell$ matrix,

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1m}B \\ a_{21}B & \ddots & \cdots & a_{2m}B \\ \vdots & \vdots & \ddots & \cdots \\ a_{n1}B & a_{n2}B & \cdots & a_{nm}B \end{bmatrix} \tag{2.3}$$

Note that each entry $a_{ij}B$ above denotes a $k \times \ell$ matrix block. The tensor product is associative but not commutative. Notably, it distributes over the usual matrix product: $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$. We will often consider the special case of the tensor product when A and B are both vectors. We will write $A^{\otimes k} = A \otimes \cdots \otimes A$ to denote the k-fold tensor product of a matrix A.

2.2 Introduction to quantum computing

The basic unit of information in quantum computers is the quantum bit, called a *qubit*.* The state of a qubit (at a specific moment in time) is described by a complex vector $\left[\begin{smallmatrix} \alpha_0, & \alpha_1 \end{smallmatrix}\right]^T \in \mathbb{C}^2$ with norm 1, usually written in Dirac notation† as $\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$.

*A qubit can be physically realized in many different ways, e.g., as the polarization of a photon, or as the spin of an electron. By analogy, a classical bit can be described by the abstract entities 0 or 1, independent of how the bit is physically realized, e.g., stored on a magnetic tape, or on a cd, or as an electric charge in a DRAM, or even as several copies of one of the above in an error-corrected context. Likewise, in this thesis, we will deal with qubits only abstractly as vectors in a complex Hilbert space, which is independent of their physical realization in a quantum computer.

†Aaronson notes, "This notation usually drives computer scientists up a wall when they first see it – especially because of the asymmetric brackets! But if you stick with it, you see that it's really not so bad." [2]

Here $|0\rangle$ is shorthand for the vector $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $|1\rangle$ is shorthand for $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, so the expression above is shorthand for

$$\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle = \alpha_0 \cdot \begin{bmatrix}1\\0\end{bmatrix} + \alpha_1 \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\alpha_0\\0\end{bmatrix} + \begin{bmatrix}0\\\alpha_1\end{bmatrix} = \begin{bmatrix}\alpha_0\\\alpha_1\end{bmatrix} \tag{2.4}$$

and satisfies

$$|\alpha_0|^2 + |\alpha_1|^2 = 1 \qquad \textbf{i.e., quantum state vectors have norm } 1 \tag{2.5}$$

The complex numbers α_0, α_1 are called the *amplitudes* of the state. Such an amplitude vector describes all information about the quantum state. If both amplitudes α_0 and α_1 are non-zero, the state is in *superposition*. It follows from Equation 2.5 that the zero vector is not a quantum state.

It is sometimes convenient, when doing calculations, to temporarily neglect the constraint that quantum state vectors have norm 1 (e.g., when doing calculations by hand, or when using methods such as QMDDs). A quantum state vector is called *(non-)normalized* when it (does not) have norm 1, to emphasize which convention is used.

Example 2.2. The states $|0\rangle = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \in \mathbb{C}^2$ and $|1\rangle = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ are (normalized) quantum states on one qubit. The state $|+\rangle = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$ is in superposition of $|0\rangle$ and $|1\rangle$.

Two amplitude vectors differing only by a complex factor are considered (physically) equivalent. More precisely, if $|\varphi\rangle, |\psi\rangle$ are quantum state vectors, and $|\varphi\rangle = \lambda\,|\psi\rangle$ for some complex value $\lambda \in \mathbb{C}$, then $|\varphi\rangle$ and $|\psi\rangle$ are said to describe the same quantum state. Notably, no physical measurement can distinguish between two such equivalent amplitude vectors.

A quantum register may consist of multiple qubits. A quantum register consisting of n qubits has 2^n basis states $|x\rangle$ with $x \in \{0,1\}^n$, each with a corresponding amplitude $\alpha_x \in \mathbb{C}$. We denote $|x\rangle = |x_n\rangle \otimes \cdots \otimes |x_1\rangle = |x_n \ldots x_1\rangle$, for example, $|00\rangle = |0\rangle \otimes |0\rangle$. The state $|\varphi\rangle$ can therefore be written,

$$|\varphi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x\,|x\rangle \tag{2.6}$$

The normalization constraint, requiring $|\varphi\rangle$ to be a unit vector, is generalized to:

$$\sum_{0 \leq x < 2^n} |\alpha_x|^2 = 1 \tag{2.7}$$

A quantum state $|\varphi\rangle$ is said to be *entangled* if it cannot be written as a tensor product of single qubit states, i.e, as $|\varphi\rangle = |\varphi_1\rangle \otimes \cdots \otimes |\varphi_n\rangle$.

Example 2.3. Consider the quantum state $1/\sqrt{2} \cdot (|00\rangle + |11\rangle)$, known as the *Bell state* [240]. As a vector, it would be written as $1/\sqrt{2} \cdot [\,1\ 0\ 0\ 1\,]^{\mathrm{T}}$. In addition to superposition, this quantum state shows *entanglement*.

Alternatively, a state $|\varphi\rangle$ can be understood as the pseudo-Boolean function $f\colon \{0,1\}^n \to \mathbb{C}$, where $f(x) = \alpha_x$. It follows that an amplitude vector can also be expressed using a Shannon decomposition, as follows,

$$|\varphi\rangle = \alpha_0 |0\rangle \otimes |\varphi_0\rangle + \alpha_1 |1\rangle \otimes |\varphi_1\rangle \tag{2.8}$$

where $|\varphi_0\rangle, |\varphi_1\rangle$ are $n-1$-qubit states and $\alpha_0, \alpha_1 \in \mathbb{C}$. Put simply, this expression defines the vector $|\varphi\rangle$ as the vector whose top half is $\alpha_0 \cdot |\varphi_0\rangle$ and whose bottom half is $\alpha_1 \cdot |\varphi_1\rangle$.

We denote the complex conjugate transpose of a given vector $|\varphi\rangle$ as $\langle\varphi| = (|\varphi\rangle^T)^\dagger$. Thus, Dirac notation makes it easy to see that $|\varphi\rangle$ is a column vector and $\langle\varphi|$ is a row vector. It follows, for example, that $\langle\varphi| \cdot |\psi\rangle$ (or $\langle\varphi|\psi\rangle$ for short) is a scalar. A column vector $|\varphi\rangle$ is called a *ket* (or *ket vector*); a row vector $\langle\varphi|$ is called a *bra*; for this reason Dirac notation is also called *bra-ket notation*. For example, if $|\varphi\rangle = \frac{3}{5}|0\rangle + i\frac{4}{5}|1\rangle$, then its complex conjugate transpose is,

$$\langle\varphi| = \frac{3}{5}\langle 0| - i\frac{4}{5}\langle 1| = \begin{bmatrix} \frac{3}{5} & -i\frac{4}{5} \end{bmatrix}^T \tag{2.9}$$

The value $|\langle\varphi|\psi\rangle|^2$ is the *fidelity* of $|\varphi\rangle$ and $|\psi\rangle$, and tells us how close two states are.

Measurement of quantum states. A quantum state can be *measured*. For the purposes of this thesis, it will be sufficient to consider only single-qubit measurements in the computational basis.[‡] Suppose that a quantum state $|\varphi\rangle$ can be written as in Equation 2.8 and that $|\varphi_0\rangle, |\varphi_1\rangle$ are normalized vectors, i.e., with norm 1. When the

[‡]For a broader treatment of measurement in quantum computing, see Nielsen and Chuang [240].

first qubit of this state is measured, this results in either outcome 0, or outcome 1. The probability of obtaining outcome m is equal to

$$\mathbb{P}[\text{measurement outcome is } m] = \frac{|\alpha_m|^2}{|\alpha_0|^2 + |\alpha_1|^2} \tag{2.10}$$

After measurement, the state "collapses". Specifically, if the outcome 0 was measured, then the state after measurement (the *post-measurement state*) is $|0\rangle \otimes |\varphi_0\rangle$; similarly, if 1 was measured then the state becomes $|1\rangle \otimes |\varphi_1\rangle$. In the new state, the first qubit is no longer entangled with the other qubits. This is reflected in the amplitude vector: if, for example, the measurement outcome was $m = 1$, then the first 2^{n-1} entries of the state vector are set to 0. For this reason, measurement is sometimes called *destructive*, since information about the quantum state is lost.

Example 2.4. Consider the Bell state from Example 2.3. If this state is measured, then the post-measurement state has equal probabilities of being one of the basis states $|00\rangle$ and $|11\rangle$, and zero probability of seeing the states $|01\rangle$ and $|10\rangle$. Measuring a value for one qubit of the Bell state immediately fixes the value of the other qubit corresponding to the measurement outcome, e.g., after measuring $q_1 = |0\rangle$ (or $q_1 = |1\rangle$) we immediately know that $q_0 = |0\rangle$ (or $q_0 = |1\rangle$).

2.2.1 Quantum states and operations

Quantum states are manipulated by applying quantum gates. A quantum gate is any unitary linear operator, i.e., a linear operator mapping quantum states to quantum states. A gate on n qubits, therefore, corresponds to a matrix $U \in \mathbb{C}^{2^n \times 2^n}$. If a quantum state $|\varphi\rangle$ serves as input to a gate U, then the output is the quantum state $U \cdot |\varphi\rangle$. A *quantum circuit* consists of a series of gates applied sequentially. Therefore, if an circuit U consists of sequentially applying first the gate U_1, then U_2, etc., until U_m, then the action of the circuit U is described by the $2^n \times 2^n$ unitary matrix $U_m \cdots U_1$, i.e., the matrix of U is simply the product of the (matrices corresponding to the) gates. We say that a quantum circuit $U_1, \ldots, U_m$ is equivalent to another quantum circuit $V_1, \ldots, V_\ell$ iff they effect the same unitary matrix modulo a complex factor, i.e., if $U_m \cdots U_1 = \lambda \cdot V_\ell \cdots V_1$ for some $\lambda \in \mathbb{C}$. The reason for considering operators up to a complex factor is because two unitary operators U, V map equivalent quantum states to equivalent quantum states if and only if these unitaries are equal modulo a complex factor, i.e., if $U = \lambda V$.

If U is a k-qubit gate, and it is applied to the first k qubits of a quantum register containing n qubits, then its action on the n-qubit system is described by the unitary matrix $U \otimes \mathbb{I}^{\otimes n-k}$. Here $\mathbb{I} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ is the identity gate and $\mathbb{I}^{\otimes n-k} = \mathbb{I} \otimes \cdots \otimes \mathbb{I}$ is its $(n-k)$-fold tensor product; this is a matrix of size $2^{n-k} \times 2^{n-k}$. We say that U *acts as the identity* on the remaining $n-k$ qubits. More generally, a gate U may of course be applied to any subset of qubits, not only to the first qubits in some variable order, as above. A matrix which acts as the identity on all but k qubits, such as the matrix $U \otimes \mathbb{I}^{\otimes n-k}$ above, is called k-*local*; in particular the gate U is also called k-local (we use the term *local* independent of whether the affected qubits are located next to each other).

If U and V are two quantum gates, then the matrix $U \otimes V$ applies U and V simultaneously to separate quantum registers. This is called the *parallel composition* of two quantum gates. The earlier expression $U \otimes \mathbb{I}^{\otimes n-k}$ can be thought of as a special case of parallel composition: we apply U to a k-qubit register, and apply the identity gate $\mathbb{I}$ to the remaining $n-k$ qubits.

Example 2.5. Three examples of common quantum gates are the single-qubit phase-shift operation S, the single-qubit Hadamard operation H, and the two-qubit controlled-NOT operation $CNOT$ (here shown with control on the first qubit, target on the second qubit).

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.11}$$

Example 2.6. The Bell state from Example 2.3 is created by the circuit in Figure 2.1. In this circuit, the quantum state starts in the initial state $|00\rangle$. First, a Hadamard gate is applied to the first qubit. The corresponding unitary is $H \otimes \mathbb{I}$, so the state after applying this gate is

$$H \otimes \mathbb{I} \cdot |00\rangle = \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \tag{2.12}$$

Next, a controlled-NOT gate is applied. Here the the first qubit acts as the *control*, denoted in the circuit by a •; the second qubit is the *target*. After applying the

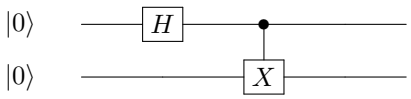


Figure 2.1: A 2-qubit quantum circuit preparing the Bell state from Example 2.3. It contains two gates: first a Hadamard is applied to the first qubit, then a controlled X gate (a $CNOT$ gate) is applied. The top qubit is the *control* and the bottom qubit is the *target* of this controlled X gate.

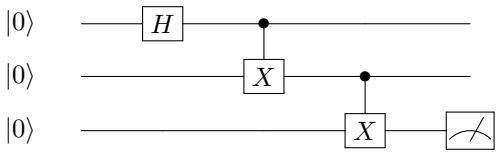


Figure 2.2: 3-qubit quantum circuit preparing the GHZ state. The rightmost box denotes a measurement to be performed on the last qubit.

controlled-NOT gate, the state is

$$\text{CNOT} \cdot \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.13}$$

The circuit is described by the 4×4 matrix $U = CNOT \cdot (H \otimes \mathbb{I})$.

Example 2.7. Figure 2.2 gives an example of a circuit preparing a quantum state on 3 qubits, the so-called GHZ state: $|\text{GHZ}\rangle = 1/\sqrt{2}(|000\rangle + |111\rangle)$. At the end of the circuit, after the three gates are applied, the last qubit is measured. This measurement has equal probability of measuring a 0, collapsing the state to $|000\rangle$, or measuring a 1, collapsing the state to $|111\rangle$.

As the examples above illustrate, a quantum circuit is typically given as a sequence of local gates (as opposed to as a sequence of exponentially large matrices). Consequently, a circuit can be succinctly described, even if the number of qubits is large.

Table 2.1 lists the most important gates encountered in this thesis. It groups gates into *Pauli gates* and *Clifford gates*, which we introduce shortly, in Sec. 2.2.3.

2.2.2 How to simulate a quantum circuit

Given a quantum circuit $U_1, \ldots, U_m$ and an initial state $|\varphi_0\rangle$, this circuit can be simulated on a classical computer straightforwardly by repeated matrix-vector multiplication. The simulation starts with the initial state and applies the quantum gates one after the other, obtaining the intermediate states $|\varphi_1\rangle, \ldots, |\varphi_m\rangle$. Each quantum operation is represented by a unitary matrix U_t of dimension $2^n \times 2^n$ and each quantum state by a unit vector $|\varphi_t\rangle$ of dimension 2^n (with $|\varphi_0\rangle$ commonly chosen to be $|\varphi_0\rangle = |0 \ldots 0\rangle$, called the all-zero state). The evolution of a state at time step t is then given by $|\varphi_{t+1}\rangle = U_{t+1} |\varphi_t\rangle$. In principle, these matrices and vectors can be stored in memory in a straightforward way as matrices of complex numbers. However, the memory requirements of this straightforward approach grow exponentially with the number of qubits (namely as 2^n for the vectors and 4^n for the matrices). For many quantum circuits, the simulation can be conducted much more efficiently by employing data structures to compactly store the intermediate states, such as the ones we introduce in Section 2.3.

2.2.3 Clifford circuits, stabilizer states and Pauli operators

An important subset of quantum circuits are Clifford circuits [137], which consist only of the three Clifford gates S, H and $CNOT$, defined in Equation 2.11. Clifford circuits play an essential role in many quantum subroutines and protocols, such as superdense coding [42], cryptography [38, 131, 263], error-correcting codes [41, 135, 137, 310] and measurement-based quantum computing [272] Clifford circuits can be efficiently simulated on a classical computer [3]. We remark that this does not immediately give an efficient way to simulate all quantum circuits, because Clifford circuits do not capture all of quantum computing: some quantum circuits cannot be expressed using only Clifford gates (see, e.g., [61, 63]). Table 2.1 lists some common Clifford gates.

We briefly sketch the idea of fast simulation of Clifford circuits before we give a detailed exposition. The fast simulation algorithm for Clifford circuits is based on Gottesman and Knill's [136] description of quantum states based on symmetries, rather than (exponential) amplitude vectors. In this formalism, we consider a set of operators which *stabilize* a state (specifically, we consider the Pauli operators that stabilize a state; see below for a definition). This set is called the *stabilizer group* of that state. The idea is to uniquely identify a state with its stabilizer group. The set of states that

can be described this way is called the *stabilizer states*. A state's stabilizer group can be efficiently represented by its generator set, which is what yields a representation of a set of states that is more compact than an amplitude vector. Lastly, consider a state $|\varphi\rangle$ with stabilizer group G, and a Clifford gate U, so that the state after applying the gate is $U|\varphi\rangle$. Then the stabilizer group of $U|\varphi\rangle$ can be efficiently computed given G and U. The same is true if we wish to simulate a measurement: the probability of obtaining a given measurement outcome can be efficiently computed, and stabilizer group of the post-measurement state can be efficiently found. This yields the fast simulation algorithm. A compact representation is feasible because there are only few n-qubit stabilizers for any $n \geq 1$, namely, $2^{\mathcal{O}(n^2)}$. The relation between Clifford circuits and stabilizer states is that the set of stabilizer states is precisely the set of intermediate and final states of Clifford circuits, if the initial state was the all-zero state $|0\rangle$.

We now describe the stabilizer formalism in detail. A unitary operator U *stabilizes* a state $|\varphi\rangle$ if $|\varphi\rangle$ is a $+1$ eigenvector of U, i.e., if it satisfies $U|\varphi\rangle = |\varphi\rangle$. The set of stabilizers of any state $|\varphi\rangle$ forms a group, since if U and V stabilize $|\varphi\rangle$, then so do UV, VU and $U^\dagger$. In the stabilizer formalism, we consider only those stabilizers that are Pauli matrices:

$$\textsc{Pauli} \triangleq \{\mathbb{I}_2, X, Y, Z\} \tag{2.14}$$

$$\text{where } \mathbb{I}_2 \triangleq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, X \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Y \triangleq \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z \triangleq \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.15}$$

The n-qubit Pauli group is defined as the group generated by all *Pauli strings* $P_1 \otimes ... \otimes P_n$ with $P_i \in \textsc{Pauli}$, i.e., $\textsc{Pauli}_n \triangleq \langle \textsc{Pauli}^{\otimes n} \rangle$. One can check that $\textsc{Pauli}_n = \{i^c P_1 \otimes ... \otimes P_n \mid P_1, ..., P_n \in \textsc{Pauli}, c \in \{0, 1, 2, 3\}\}$. In particular, we have $\textsc{Pauli}_1 = \{\pm P, \pm iP \mid P \in \textsc{Pauli}\}$ (the Pauli set with a factor ± 1 or $\pm i$). Pauli operators A, B either commute ($A \cdot B = B \cdot A$) or anticommute ($A \cdot B = -B \cdot A$).

Definition 2.1 (Stabilizer state, stabilizer group)**.** For a given n-qubit state $|\varphi\rangle$, its *stabilizer group* is the group of Pauli operators $P \in \textsc{Pauli}_n$ that stabilize $|\varphi\rangle$. This group is denoted $\text{Stab}(|\varphi\rangle)$. An element $P \in \text{Stab}(|\varphi\rangle)$ is called a *stabilizer* of $|\varphi\rangle$. A state $|\varphi\rangle$ is called a *stabilizer state* if it has 2^n stabilizers, i.e., if $|\text{Stab}(|\varphi\rangle)| = 2^n$. $\diamond$

The stabilizer group of an n-qubit state always contains exactly 2^k elements for some $0 \leq k \leq n$. For any stabilizer group $G = \text{Stab}(|\varphi\rangle)$ with 2^k elements, there is a generating set S with $|S| = k$ elements. Moreover, no two stabilizer states have the

same stabilizer group. Since (i) a stabilizer state's stabilizer group has n generators; (ii) each of which is a Pauli string of n Pauli matrices, plus a complex factor $c \in \{\pm 1, \pm i\}$ called the *phase*; and (iii) there are only 4 Pauli matrices, a stabilizer state can be uniquely described using only $\mathcal{O}(n^2)$ bits. It immediately follows that there are only $2^{\mathcal{O}(n^2)}$ n-qubit stabilizer states. More precisely, Aaronson and Gottesman (see [3], Proposition 2) show that the number of stabilizer states on n qubits grows asymptotically as $2^{(1/2+o(1))n^2}$.

Example 2.8. Examples of (generators of) stabilizer groups are $\mathrm{Stab}(|0\rangle) = \langle Z \rangle$ and, letting $|\Phi_0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ be the Bell state, $\mathrm{Stab}(|\Phi_0\rangle) = \langle X \otimes X, Z \otimes Z \rangle$.

Updating a stabilizer state's generators after application of a Clifford gate or a single-qubit computational-basis measurement can be done in polynomial time in the number of qubits [3, 136]. Various efficient algorithms exist for manipulating stabilizer (sub)groups G, including testing membership (is $A \in \textsc{Pauli}_n$ a member of G?) and finding a generating set of the intersection of two stabilizer (sub)groups. These algorithms predominantly use standard linear algebra, e.g., Gauss-Jordan elimination, as described in Sec. 3.5.2 in detail.

Example 2.9. We show how to use the stabilizer formalism to simulate the circuit in Figure 2.2. First, we use the explicit state vector representation; then we show the same simulation but using the stabilizer formalism. Here q_j denotes the j-th qubit register.

$$
\begin{aligned}
|\varphi_0\rangle &= |000\rangle && \text{Initial state} \\
|\varphi_1\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|100\rangle && \text{applied } H \text{ to } q_1 \\
|\varphi_2\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|110\rangle && \text{applied CNOT to control } q_1 \text{ and target } q_2 \\
|\varphi_3\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|111\rangle && \text{applied CNOT to control } q_1 \text{ and target } q_3
\end{aligned}
$$

To perform the simulation of this circuit using the stabilizer formalism, we identify

each state $|\varphi_t\rangle$ with its stabilizer group $G(\varphi_t) = \{G_1, G_2, G_3\}$, as follows.

$$
\begin{aligned}
G(\varphi_0) &= \{Z\mathbb{I}\mathbb{I}, \mathbb{I}Z\mathbb{I}, \mathbb{I}\mathbb{I}Z\} && \text{Initial state} \\
G(\varphi_1) &= \{\mathbb{I}\mathbb{I}Z, \mathbb{I}Z\mathbb{I}, X\mathbb{I}\mathbb{I}\} && \text{applied } H \text{ to } q_1 \\
G(\varphi_2) &= \{\mathbb{I}\mathbb{I}Z, ZZ\mathbb{I}, XX\mathbb{I}\} && \text{applied CNOT to control } q_1 \text{ and target } q_2 \\
G(\varphi_3) &= \{ZZ\mathbb{I}, Z\mathbb{I}Z, XXX\} && \text{applied CNOT to control } q_1 \text{ and target } q_3
\end{aligned}
$$

For the purposes of this Introduction, we omit the detailed algorithms for obtaining the new stabilizer group after applying a Clifford gate; the interested reader may consult [3].

Graph states on n qubits are the output states of circuits with input state $\frac{1}{2^{n/2}}(|0\rangle + |1\rangle)^{\otimes n}$ followed by only controlled Z gates. Graph states form a strict subset of all stabilizer states that is also important in error correction and measurement-based quantum computing [153].

We remark that, in contrast to other work, as noted in Definition 2.1, we also consider the stabilizer groups of states that are not stabilizer states. In general, we will refer to any abelian subgroup of $\textsc{Pauli}_n$, not containing $-\mathbb{I}_2^{\otimes n}$, as an n-qubit *stabilizer subgroup*; such a group has $\leq n$ generators. Such objects are also studied in the context of simulating mixed states [24] and quantum error correction [135]. A state is uniquely defined by its stabilizer group G if and only if $|G| = 2^n$, i.e., the elements of the group G have a unique joint $+1$ eigenvector if and only if $|G| = 2^n$.

Example 2.10. Examples of stabilizer subgroups are $\text{Stab}(\frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)) = \{\mathbb{I}_2\}$, $\text{Stab}(|1\rangle) = \langle -Z \rangle$ and $\text{Stab}(\frac{1}{\sqrt{3}}(|00\rangle + |11\rangle) + \frac{2}{\sqrt{3}}(|01\rangle + |10\rangle)) = \langle X \otimes X \rangle$.

2.3 Decision Diagrams

Decision Diagrams are the protagonists of the story told in this dissertation.

We start with the definition, below, of a Binary Decision Diagram (BDD). It will serve as a template for all decision diagrams that follow.

Definition 2.2 (Binary Decision Diagram (BDD))**.** A BDD is a rooted, directed acyclic graph. It has two leaves, labeled True (or 1) and False (or 0). A non-leaf node is called a *Shannon node*; it is labeled with (the index of) a variable and has two

Table 2.1: The quantum gates used in this thesis. We remark that all Pauli gates are, in particular, Clifford gates.

	Gate name	**Symbol**	**Matrix**	**num. of qubits**
Pauli gates	Identity	$\mathbb{I}$	$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	1
	Pauli X	X	$= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	1
	Pauli Y	Y	$= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	1
	Pauli Z	Z	$= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	1
Clifford gates	Hadamard	H	$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	1
	Phase shift	S	$= \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	1
	Controlled NOT	CX	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & X \end{bmatrix}$	2
	Controlled Y	CY	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & Y \end{bmatrix}$	2
	Controlled Z	CZ	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & Z \end{bmatrix}$	2
	Swap	Swap	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	2
Other	T-gate	T	$= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$	1
	Toffoli	CCX	$= \begin{bmatrix} \mathbb{I} & 0 \\ 0 & CX \end{bmatrix}$	3

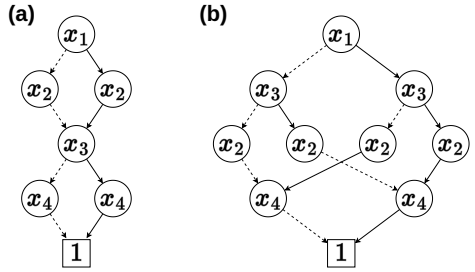


Figure 2.3: Two reduced, ordered BDDs representing the function $f(x_1, x_2, x_3, x_4) = (x_1 = x_2) \wedge (x_3 = x_4)$. BDD (a) uses the variable order $x_1 < x_2 < x_3 < x_4$; BDD (b) uses the variable order $x_1 < x_3 < x_2 < x_4$.

outgoing edges, called the *low edge* and the *high edge*. We will use the notation $\text{—}(v)$ to denote an edge to node v, and $(v_0)\cdots(v)\text{———}(v_1)$ to denote a node with a low edge to v_0 and a high edge to v_1. Each node v represents a Boolean function $[\![v]\!]$, defined inductively as follows. In the base case, the TRUE and FALSE leaves represent the constant functions $[\![\text{TRUE}]\!] = 1$ and $[\![\text{FALSE}]\!] = 0$, respectively. If a Shannon node v is labeled with variable x and has low edge to v_0 and high edge to v_1, then it represents the function $[\![v]\!] \triangleq \neg x \wedge [\![v_0]\!] \vee x \wedge [\![v_1]\!]$. A BDD is *ordered* if, on each path from the root to a leaf, each variable appears at most once and always in the same order. Two nodes u, v are called *equivalent* if they represent the same function, $[\![u]\!] = [\![v]\!]$. A BDD is *reduced* if there are no equivalent nodes. *Layer* i in an ordered BDD, is the set of nodes with variable label x_i. ◇

Example 2.11. Figures 2.3(a) and (b) show examples of BDDs. The two BDDs represent the same function, $f \triangleq (x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$. They have different shapes because they employ different variables orders, $x_1 < x_2 < x_3 < x_4$ and $x_1 < x_3 < x_2 < x_4$, respectively.

In the figure, the value $f(x)$ of an assignment x can be found by traversing the diagram from root to leaf as follows. One starts at the root node. A node is labeled with a variable x_i; if $x_i = 0$, we traverse the low (dotted) edge; otherwise, if $x_i = 1$, we traverse the high (solid) edge, until we arrive at a leaf, at which point we have found $f(x)$. More generally, a path from the root to a node v on layer k corresponds to a *partial assignment* $x_n = a_n, \ldots, x_{k-1} = a_{k-1}$ for some string $a \in \{0,1\}^{n-k}$. This node represents the induced subfunction f_a. To avoid cluttering the diagram, edges to the FALSE Leaf are not drawn in the figure.

To emphasize that the size of the BDD is influenced by the variable order, let us consider what happens if we generalize the function f and the variable orders from $n = 4$ to $n > 4$ variables. The function then becomes $f = (x_1 \Leftrightarrow x_2) \wedge \cdots \wedge (x_{n-1} \Leftrightarrow x_n)$ and the variable orders become $\sigma = x_1 < \cdots < x_n$ and $\pi = x_1 < x_3 < x_5 < \cdots < x_{n-1} < x_2 < x_4 \cdots < x_n$. Then the BDD with variable order σ has linear size, whereas the BDD with variable order π has exponential size (in the number of variables). In general, finding the optimal variable order for a given function is an NP-hard problem. [56].

Bryant [67] observed that BDDs can be queried and manipulated in polynomial time in the size of the diagrams (size is defined as the number of nodes). For example, (i) given a BDD of size m representing the function f, the number of models of f can be counted in time $\mathcal{O}(m)$; (ii) given BDDs f and g, with k and m nodes, respectively, a BDD representing the function $f \wedge g$ can be constructed in $\mathcal{O}(km)$ time.

A reduced and ordered BDD (ROBDD) is a canonical representation of its corresponding Boolean function [67]: a given function has a unique ROBDD. Canonicity ensures that the BDD is always as small as possible as equivalent nodes are merged. But more importantly, canonicity allows for quick equality checks: two reduced diagrams represent the same state *if and only if* the diagrams are exactly identical. In a practical decision diagram software package, this equality check can be performed in constant time: it suffices to check whether the two diagrams have the same root node, since the diagrams will share equivalent nodes in such a software package. More generally, canonicity allows for efficient manipulation algorithms through dynamic programming, which avoids traversing all paths (exponentially many in the size of the diagram in the worst case).

2.3.1 Decision diagrams for quantum states

The BDD, defined in Definition 2.2, can represent only Boolean functions, which are of the form $f\colon \{0,1\}^n \to \{0,1\}$. To represent quantum states, we use decision diagrams which are capable of representing *pseudo-Boolean* functions, of the form $f\colon \{0,1\}^n \to \mathbb{C}$. We review two prominent examples of decision diagrams that have this capability: the Quantum Information Decision Diagram (QuIDD, Definition 2.3) and the QMDD (Definition 2.4). The QuIDD extends the BDD by allowing more than two leaves: it contains one leaf for every element in the image of f. The interpretation

of the QuIDD is similar to the BDD: to obtain the value $f(x)$ given a string $x \in \{0,1\}^n$, one simply traverses the diagram from root to leaf, traversing the low edge of a node when $x_j = 0$ and otherwise the high edge, when $x_j = 1$. The QuIDD is a specialization of an ADD; specifically, a QuIDD is an ADD in which the range of the function is restricted to the complex numbers. The QMDD (Definition 2.4) improves on the QuIDD (Definition 2.3) by merging nodes which are equivalent up to a complex factor. This complex factor is then stored as a label on the edges incident to the node that was merged. Because the QMDD merges nodes, it sometimes contains fewer nodes than the QuIDD. In the best case, the QMDD contains exponentially fewer nodes than the QuIDD. Given a QMDD representing a quantum state, measurement probabilities can be computed in polynomial time in the size of the QMDD. For all quantum gates there are algorithms to update the QMDD accordingly, i.e., given a gate U and a QMDD representing $|\varphi\rangle$, to construct a QMDD representing the state $U|\varphi\rangle$.

Definition 2.3 (Quantum Information Decision Diagram (QuIDD) [333])**.** A QuIDD is a rooted directed acyclic graph (DAG), representing a quantum state $|\varphi\rangle$. It has a leaf node for each unique amplitude of $|\varphi\rangle$, i.e., for each element in the set $\{\langle x|\varphi\rangle \mid x \in \{0,1\}^n\}$. A leaf labeled $z \in \mathbb{C}$ represents the length-1 vector $[z]$. Just as in a BDD, a non-leaf node is called a Shannon node. A Shannon node v with a low edge to node v_0, and a high edge to v_1, represents the quantum state $|v\rangle = |0\rangle \otimes |v_0\rangle + |1\rangle \otimes |v_1\rangle$. ⋄

Definition 2.4 (Quantum Multi-valued Decision Diagram (QMDD) [227, 374])**.** A QMDD (QMDD) is a QuIDD in which (i) each edge is labeled with a complex value; and (ii) there is a unique leaf node, labeled with 1. An edge to node v with label z is denoted $\xrightarrow{z}(v)$, and denotes the state $z|v\rangle$. A Shannon node with outgoing low edge $\xrightarrow{a}(v_0)$ and high edge $\xrightarrow{b}(v_1)$ is denoted $(v_0) \overset{a}{\cdots} (v) \overset{b}{—} (v_1)$. Such a node represents the quantum state $|v\rangle = a|0\rangle \otimes |v_0\rangle + b|1\rangle|v_1\rangle$ (this is the Shannon decomposition from Equation 2.8). ⋄

The DDs above all represent vectors. It is also possible to represent matrices using DDs [227, 333], but we defer this topic to Chapter 3.

Example 2.12. Figure 2.4 shows an example of a QMDD (d) and its construction from a binary tree (a).

It is instructive to consider the similarities and differences between the diagrams. In each type of decision diagram, a node v has two outgoing edges to nodes v_0, v_1, which represent functions on one fewer variable; the decision diagrams are all *ordered*, and

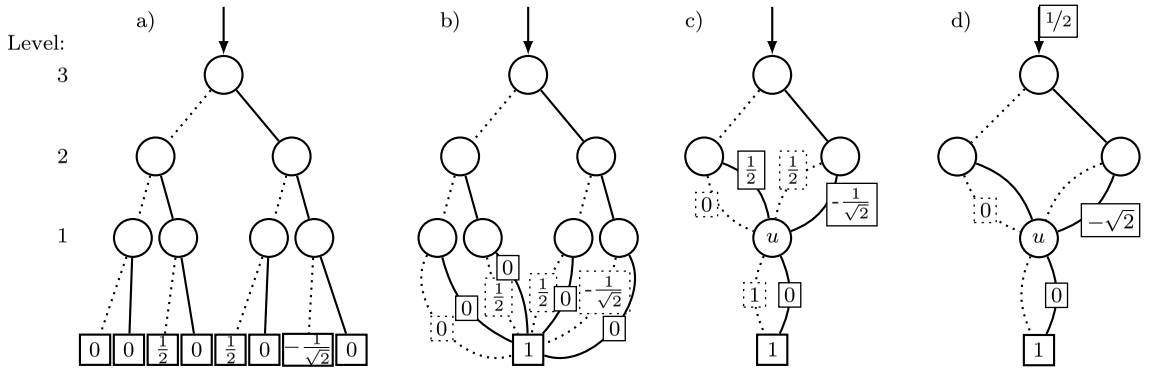
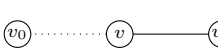


Figure 2.4: Different decision diagrams representing the 3-qubit state $[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0]^\top$, evolving into a QMDD (right). Left, (a) shows the exponential binary tree, where a node on level i represents x_i (see Equation 2.8) and its outgoing arrows $x_i = 0$ (dashed) and $x_i = 1$ (solid). The leaf contains the complex amplitude for the assignment corresponding to the path from the root. Next (b), the leafs are merged by dividing out common factors, putting these as weights (shown in boxes) on the edges of level 1 nodes (we can suppress a separate 0 leaf, as $0 = 0 \cdot 1$). Then the same trick is applied to level 1 nodes in (c). Here all level 1 nodes become *isomorphic* and can be merged into a new node u (note again that $0 \cdot |u\rangle = [0, 0]^\top$, where $|u\rangle = [1, 0]$ is the vector that node u represents). Finally, (d) shows the resulting QMDD, applying the same tactic to nodes on levels 2 and 3. Note that a QMDD requires a root edge. Merging (*isomorphic*) nodes makes QMDDs succinct. Adapted from Fig 2 in [374].

they can all be *reduced* by merging equivalent nodes. On the other hand, the various DDs differ in that they may label the edges with additional information. An edge e, labeled with a label ℓ, pointing to a node v, represents the state $|e\rangle = \ell \circ |v\rangle$, where $\circ$ is some appropriate notion of multiplication. Moreover, although all decision diagrams merge equivalent nodes, they understand equivalence differently. For example, the QuIDD considers two nodes u, v equivalent when they represent the same function, i.e., when $|u\rangle = |v\rangle$, whereas the QMDD considers two nodes equivalent when there exists a complex constant λ such that $\lambda\,|u\rangle = |v\rangle$. Lastly, different decision diagrams may impose different *reduction rules* in order to ensure canonicity.

The field of decision diagrams is a fascinating and diverse field of research. Many decision diagrams have been proposed and implemented, of which Table 2.2 lists a small selection. The interested reader is directed to [344] for a more comprehensive treatment.

Lastly, although the definitions above are useful and unambiguous, it is often helpful to view decision diagrams from several complementary perspectives. We list some below.[§] We have found these different perspectives especially useful while designing new DDs and when trying to understand differences between DDs.

- A DD is a method of lossless data compression. A striking feature about decision diagrams is that they allow us to operate on the data *without decompressing it first*.
- A DD is a rooted DAG in which the set of paths from the root to the leaves correspond precisely to the assignments of f.
- A DD is a graphical depiction of the subfunctions of a function f. More precisely, the nodes of the diagram are in one-to-one correspondence with the induced subfunctions of f; and two subfunctions g, h of f are connected by an edge $g \to h$ iff h is a subfunction of g.
- A DD is a way to inductively define quantum states by using the Shannon decomposition. Namely, to define an amplitude vector $|\varphi\rangle$, whose top half is $|\varphi_0\rangle$, and whose bottom half is $|\varphi_1\rangle$, we first construct the DDs v_0, v_1 representing these two smaller vectors; then the vector $|\varphi\rangle$ can be represented by a node 

, whose two edges go to v_0 and v_1.

[§]We do not claim that all these perspectives are novel, only that, to the best of our knowledge, no source collects them into a list.

- A DD is obtained from a binary decision tree by merging equivalent nodes (under some suitable notion of equivalence).
- A DD is a finite state machine which computes a function $f(x)$ by reading an input string x and terminating after exactly $\text{len}(x)$ steps. Possibly the machine is equipped with a small amount of internal memory to process the DD's edge labels.
- A DD is a Boolean (or algebraic, depending on the range) circuit composed of AND, OR and NOT gates (or multiplication and addition gates), with a single output gate. The output gate computes the function we are interested in. Not all circuits are decision diagrams, but every decision diagram is a circuit.

2.3.2 Comparing decision diagrams

The decision diagrams described above have different behaviour on the same problem, owing to their different reduction rules and merging strategies. Consequently, some decision diagrams may consume less memory, or less computation time, than another, for some given task. To quantify these differences, decision diagrams, and data structures more broadly, can be analytically compared on three criteria: succinctness, tractability, and rapidity. Succinctness is a partial order which tells us whether one data structure (DS) consumes less computer memory than another. Tractability tells us whether or not a given DS performs a given operation (such as applying a certain gate or measurement) in polynomial time in the size of the input diagram(s). Lastly, rapidity is a partial order which tells us whether one DS performs a given operation faster than another DS. Rapidity was introduced by Lai et al. [194]. Formal definitions follow.

Definition 2.5 (Succinctness, adapated from Darwiche and Marquis [97])**.** Let D_1, D_2 be two data structures. Then D_1 is *at least as succinct as* D_2, denoted $D_1 \preceq_s D_2$, iff there exists a polynomial p such that for every instance $\alpha \in D_2$ there exists an equivalent instance $\beta \in D_1$ such that $|\beta| \leq p(|\alpha|)$. Here $|\alpha|$ and $|\beta|$ are the sizes of α and β, respectively.

In this definition, the size $|\alpha|$ of a decision diagram is commonly taken to be the number of nodes in the diagram, or on the widest layer.

Table 2.2: Various decision diagrams (DDs) treated by the literature. A DD represents a function $f\colon \{0,1\}^n \to R$ where the column Range specifies the set R. Here S is an arbitrary algebraic structure. The column *Merging strategy* lists the conditions under which two nodes v, w, representing subfunctions $f, g\colon \{0,1\}^k \to \mathbb{R}$ are merged. Here $z, a \in \mathbb{C}$ are complex constants, P_i are Pauli gates and $f + a$ denotes the function defined by $f(x) + a$ for all x. The DDs in **bold underlined text** are treated in depth in this thesis. See Chapter 7 for a definition of *variable tree*; see Gergov and Meinel [129] for a definition of FBDD type.

Architecture	**(Quantum) decision diagrams (and variants)**	Range	**Node merging strategy**
variable order	Decision Tree	(any)	(no merging)
	BDD, [67] ZDD, [229] TBDD, [328] CBDD, [68] KFBDD, [103] DSDBDD, [46, 264] CCDD, [195] Partitioned ROBDD, [237] Mod-2-OBDD, [130] ROBDD$[\wedge_i]_{\mathcal{C}}$, [194] CFLOBDD [293]	$\{0,1\}$	$f = g$
	MTBDD [87], **QuIDD** [331]	$\mathbb{C}$	$f = g$
	ADD [27]	S	$f = g$
	SLDD$_\times$ [114, 352], **QMDD** [227, 374], TDD, [163] WCFLOBDD [296]	$\mathbb{C}$	$f = z \cdot g$
	SLDD$_+$ [114], EVBDD [193]	$\mathbb{C}$	$f = g + a$
	AADD [281], FEVBDD [308]	$\mathbb{C}$	$f = z \cdot g + a$
	LIMDD [337]	$\mathbb{C}$	$f = zP_1 \otimes ... \otimes P_n \cdot g$
	DDMF	$U(2)$	$M = \mathbb{I} \otimes \cdots \otimes id \otimes U \cdot R$
variable tree	**SDD**, [96] ZSDD, [245] TSDD, [111] VS-SDD [235]	$\{0,1\}$	$f = g$
	PSDD [180]	$\mathbb{R}$	$f = g$
FBDD type	FBDD [129], Partitioned ROBDD [237]	$\{0,1\}$	$f = g$

Definition 2.6 (Tractability, inferred from Darwiche and Marquis [97])**.** A data structure D *supports* a query or transformation OP iff there is a polynomial-time algorithm performing OP, taking as input (an) instance(s) of D. That is, the algorithm is polynomial in the size $|\alpha|$ of the instance α. In that case, the operation is said to be *tractable* for D.

Here, a *query* is a function mapping (tuples of) quantum states to some fixed range, whereas a *transformation* maps (tuples of) quantum states to a quantum state represented by the same data structure D. For example, computing the probability of a measurement outcome is a *query*; therefore, algorithms on both QuIDD and QMDD have the same range, namely, they output a real number in $[0, 1]$. By contrast, the addition of two quantum states is a *transformation*. Here, on input $(|\varphi\rangle, |\psi\rangle)$, the QuIDD (QMDD) algorithm for addition is expected to output a QuIDD (resp. QMDD) representing the vector $|\varphi\rangle + |\psi\rangle$. Therefore, the codomain of the addition algorithms of QuIDD and QMDD are different.

Definition 2.7 (Rapidity, adapted from Lai et al. [194])**.** Let D_1, D_2 be two canonical data structures and OP an operation. Then D_1 performs OP *at least as rapidly* as D_2 iff, for each algorithm ALG_1 performing $OP(D_2)$, there exists some polynomial p and an algorithm ALG_1 performing $OP(D_1)$ such that, for every input x to $OP(D_2)$, and its equivalent input y to $OP(D_1)$, the runtime of ALG_1 is $p(t_2 + |x|)$, where t_2 is the runtime of ALG_2 on y.

We emphasize that this definition requires both data structures to be canonical. For non-canonical data structures, "*the equivalent input*" is not defined, as there may be multiple instances of D_1 representing the same quantum state, possibly having different sizes. In Chapter 5, we give a generalized definition which drops the requirement that the data structures are canonical.

2.4 Approaches to quantum software verification

In this section, we list several approaches to quantum software verification. This establishes the context of the work of this thesis, but is not intended as an exhaustive survey of the topic. For a more in-depth treatment, we refer the reader to Ying and Feng's book on quantum model checking [361], their survey [360] and to an accessible introduction to the topic by Turrini [317].

We divide these contributions into *specification languages*, and *software tools*. A specification language for quantum programs is a logic in which a user can specify the permitted behaviour or output of a quantum program, including how the (quantum) state of the program may evolve over time (very broadly speaking). A software tool puts such a specification language into practice by providing a practical method to check whether this quantum protocol satisfies its specification. Several existing tools provide support for quantum programming languages: they allow programmers to write their quantum programs in a high-level programming language, as opposed to defining a quantum circuit gate by gate. Many of these languages support abstract data types, such as integers, Boolean values and qubits. Assertions can then be formulated inside the code, e.g., as preconditions and postconditions, which the model checking tool can then verify. Several of the tools mentioned below build on two established quantum programming languages: QPL [285] and Q# [306]. We refer the interested reader to the surveys on quantum programming languages of Garwahl et al. [126] and Ferreina [118].

Relation to the present work. Relative to the work below, in this work we adopt the simplest "specification language," namely, we aim to simply check whether two circuits are equivalent. We present a software tool which lays the groundwork for this purpose in Chapter 3 and Chapter 4. We discuss possibilities for extending the present work to more expressive specification languages and logics, such as those described below, in Section 8.3.

Specification languages. A specification language is a logic which specifies how a state may evolve over time. Most such logics can be thought of, informally, as being constructed in two steps: first, we define a set of atomic propositions, which allows us to express assertions about quantum states; in the second step, we add temporal operators, which allows us to express assertions about how quantum states evolve over the course of the program, e.g., which quantum states are supposed to be reachable in the program. The second step makes the logic a quantum *temporal* logic. By analogy, in the classical domain, CTL can be obtained using this recipe by starting with propositional logic, which we can use to formulate assertions about states of a system, and then adding temporal operators. Similarly, quantum CTL [30] (see below) can be obtained by starting with so-called quantum propositional logic (EQPL [217]) and adding temporal operators. The approaches listed below use one of three methods for expressing assertions about quantum states, thus accomplishing the first step in our recipe: they use either (i) exogenous quantum propositional logic (EQPL), defined

by Mateus and Sernadas [217]; or (ii) quantum preconditions and postconditions, formulated by D'Hondt and Panangaden [100]; or (iii) closed subspaces of the Hilbert space, used by Ying in [363]. Below, we start by treating work based on the EQPL formalism of Mateus and Sernadas, followed by work which builds on D'Hondt and Panangaden's approach, and lastly we mention Ying's quantum LTL based on closed subspaces.

Mateus and Sernadas introduce *exogenous quantum propositional logic* (EQPL), which allows one to express assertions about pure quantum states, e.g., that a given subset of the qubits is not entangled with another subset [217]. Chadha, Mateus and Sernadas extend this logic to mixed states (i.e., probability distributions over quantum states), obtaining *Ensemble EQPL* (EEQPL) [79].

Baltazar, Chadha and Mateus introduce quantum computation tree logic (QCTL) [30]. This logic is obtained from the EQPL of Mateus and Sernadas [217] by adding the usual temporal operators of CTL. For example, one can specify that *eventually*, a given subset of qubits will be entangled with another subset.

Chadha, Mateus and Sernadas formulate a Hoare-style logic for quantum programs, in which preconditions and postconditions are assertions in EEQPL [79]. They formulate axioms and inference rules for this logic and show that the resulting calculus is sound. Their Hoare logic can reason only about so-called quantum while-programs with bounded iteration.

The literature contains two approaches to formulating a quantum analogue of linear time logic, by Mateus et al. [216] and by Ying, Li and Feng [363]. First, Mateus et al. [216] introduce a quantum linear time logic (QLTL) based on the EQPL of Mateus and Sernadas [217]. In QLTL, one can assert, for example, that in the next computation step, some subset of qubits is unentangled with another set of qubits. Building on the QCTL of Baltazar et al. [30], Mateus et al. study the satisfiability and model checking problems of QCTL and QLTL.

Second, Ying, Li and Feng [363] introduce another quantum analogue of linear-time logic. In this logic, an atomic proposition asserts that a state $|\psi\rangle$ is an element of a set X, where X is a closed subspace of the Hilbert space. A linear-time property P is then defined as a (finite or infinite) collection of (finite or infinite) sequences of conjunctions of atomic propositions. They use the quantum automaton [186] as their model of computation. A (finite or infinite) sequence of states of this automaton is called a *trace*, just as in the classical sense. Let $A = A_0, A_1, \ldots \in P$ be an element of

a linear-time property P, i.e., each A_j is a conjunction of atomic propositions. Then a trace $\pi = |\varphi_0\rangle, |\varphi_1\rangle, \ldots$ of an automaton satisfies A if $|\varphi_t\rangle \in X$ for each $X \in A_t$ at each time $t \geq 0$. An automaton satisfies a given predicate P if all the automaton's possible traces satisfy some $A \in P$. Ying, Li and Feng provide an algorithm for checking whether an automaton satisfies a given invariant.

D'Hondt and Panangaden formulate a notion of pre- and postconditions which takes a slightly different approach than the propositional logic approach above [100]. Their assertions are Hermitian operators P satisfying $0 \leq P \leq \mathbb{I}$.[¶] For a given state $|\varphi\rangle$ and an assertion P (i.e., a Hermitian operator), D'Hondt and Panangaden argue that the value $0 \leq \langle\varphi| P |\varphi\rangle \leq 1$ should be interpreted, informally speaking, as the probability that $|\varphi\rangle$ satisfies P. This is analogous to a classical probabilistic statement: if μ is a probability distribution and P is a classical predicate, then the state satisfies P with probability $0 \leq \mu.P \leq 1$. With this in mind, D'Hondt and Panangaden go on to define that, for a given quantum gate U and Hermitian operators P, Q, the triple $\{P\}U\{Q\}$ constitutes a valid Hoare triple if, for all states $|\varphi\rangle$, it holds that $P \leq U^\dagger Q U$. Consequently, if $\{P\}U\{Q\}$ is a valid Hoare triple, then a quantum state will satisfy Q with probability p after application of the gate U if it satisfied P with probability p before applying the gate. This formalization naturally lends itself to mixed states in addition to pure states.

Ying uses the Hoare triples of D'Hondt and Panangaden to establish a full-fledged quantum Hoare logic for quantum while-programs [359]. He gives deduction rules for this logic and formulates and proves soundness and completeness for partial and total correctness. Relative to the quantum Hoare logic of Chadha et al. [79], the quantum Hoare logic of Ying is based on D'Hondt and Panangaden's approach in which an assertion is a Hermitian operator, rather than a statement in EQPL [217], as in Chadha et al.'s case. Zhou et al. consider the special case of this Hoare logic in which the assertions are restricted to be projections, rather than general Hermitian operators [367]. They argue that this is a common use case and suffices to prove correctness of several quantum programs. Since such assertions may be easier to generate and check, this justifies the penalty to expressiveness.

Software tools. We first list tools that model check specifications given in a temporal logic, after which we list tools that check equivalence of two quantum circuits or programs.

[¶]Here $A \leq B$ means $\langle\varphi| A |\varphi\rangle \leq \langle\varphi| B |\varphi\rangle$ for all quantum states $|\varphi\rangle$. Therefore, requiring that $0 \leq A \leq \mathbb{I}$ ensures that $0 \leq \langle\varphi| A |\varphi\rangle \leq 1$ for all states $|\varphi\rangle$.

First, Gay and Nagarajan [128] build a model checker which can check QCTL properties of a given quantum protocol. The quantum protocols that are considered are restricted: the only quantum gates that are allowed are the Clifford gates. Since the Clifford gate set is not universal for quantum computing, not all quantum protocols can be expressed in this tool.

Honarvar and Nagarajan [161] implement a model checker for programs written in the Q# quantum programming language [306]. They allow a user to formulate a precondition and postcondition for any program statement, inspired by the quantum Hoare logics of Ying et al. [359] and Zhou et al. [367] above.

Feng et al. [116] build QPMC, which can check QCTL properties (see [30]) of quantum programs. To this end, they develop a quantum programming language by extending the PRISM guarded-command based language [191]. QPMC extends the probabilistic model checker IScasMC [147] and, contrary to the model checker of Gay and Nagarajan [128], can check general quantum programs, as it does not restrict the gate set. Internally, QPMC models a program as a quantum Markov chain, a notion introduced by Feng et al. [117].

Liu et al. [205] formalize the quantum Hoare logic of Ying [359] in the proof assistant Isabelle/HOL [244]. They use this formalization to prove the correctness of an implementation of Grover's algorithm.

Ardeshir-Larijani, Gay and Nagarajan build two equivalence checking tools: first, an equivalence checker for quantum programs and protocols [17] that are written in the *Quantum Programming Language* (QPL, introduced by Selinger [285]); and second, an equivalence checker for concurrent quantum programs [18]. In [18], Ardeshir-Larijani et al. develop a new quantum programming language for concurrent quantum processes. This programming language is inspired by qCCS of Ying et al., an algebra of concurrent quantum processes [362]. Just as in [128], the quantum protcols that are considered restricted to use only Clifford gates.

Burgholzer et al. provide two tools for quantum circuit equivalence checking. First, given circuits $U = U_1 \cdots U_m$ and $V = V_1 \ldots V_\ell$, Burgholer and Wille [75] use QMDDs to construct the unitary matrix $U^\dagger V$. If the circuits are equivalent, i.e., if $U = \lambda V$, then $U^\dagger V = \lambda \mathbb{I}$, which is a matrix with a very simple QMDD. With this in mind, they construct the matrix $U^\dagger V$ by iteratively constructing matrices of the form $(U_1 \cdots U_a)^\dagger \cdot (V_1 \cdots V_b)$ for some $a \leq m, b \leq \ell$, finding empirically that these intermediate matrices often remain close to the identity and hence, have small QMDDs.

Second, Burgholzer, Kueng and Wille [71] use QMDDs to simulate two given circuits on a randomly chosen stabilizer state. If the circuits are found to yield different output states, they conclude the circuits are not equivalent. Although this method is sound but not complete, they show that the probability of finding a counterexample is quite good if U and V are "very different," in a precise sense. Specifically, they consider the *average fidelity* $\mathcal{F}_{\text{avg}}(U, V)$, of two unitary matrices, defined below. They show that for a random stabilizer state $|g\rangle$, and unitaries U, V, it holds that

$$\mathbb{E}_{|g\rangle}[\langle g| U^\dagger V |g\rangle] \approx \mathcal{F}_{\text{avg}}(U, V) = \frac{1}{2^n + 1}\left(1 + 2^n \left|\text{tr}(U^\dagger V)\right|^2\right) \tag{2.16}$$

Using Markov's inequality, we immediately obtain a bound on the probability that the circuits yield the same result:

$$\mathbb{P}_{|g\rangle}[|\langle g| U^\dagger V |g\rangle|^2 = 1] \leq \mathcal{F}_{\text{avg}}(U, V) \tag{2.17}$$

Notably, the expectation value achieved by random stabilizer states in Equation 2.16 is the same as that achieved by a random Haar state. This is a consequence of the fact that the set of stabilizer states is a complex projective 3-design [190]. In this narrow sense, therefore, using random stabilizer states is *optimal* in the black box setting. However, random stabilizer states are not optimal in the sense that the probability of obtaining a counterexample is not as high as for a random Haar state. To achieve this, one would need to choose as random inputs a different set of states, which is a projective t-design for some large t.

Lastly, Hong et al. [162] implement a tool for quantum circuit equivalence checking in the presence of noise. This equivalence checking of noisy circuits is more general than the equivalence checking problem studied by Burgholzer et al. above [71, 75]. The noisy circuit is modeled by a superoperator, rather than a unitary matrix. They represent the noisy circuit as a tensor network using the Tensor Decision Diagram (TDD); they then contract the tensor network using operations on the TDD.