Universiteit Leiden

New Decision Diagram-based Techniques for the Simulation of Quantum Computations

Name: Martijn Swenne

Date: 28/07/2021

1st supervisor: Dr. A.W. Laarman
2nd supervisor: Dr. V. Dunjko
Daily supervisor: S.O. Brand, MSc.

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Quantum computing promises fundamental improvements for efficient computability. Impact is expected in scientific simulations in the fields of physics and chemistry, in AI through machine learning and in solving linear equations. One obstacle however is found in the difficulty of designing circuits in these early generations of quantum computers.

This thesis provides new algorithms for simulations of quantum circuits based on decision diagrams, a technique that has shown promising results in practice. In order to scale the technique, we investigate new ways to represent quantum circuits and quantum states in decision diagrams. We also propose new algorithms to balance the trade-off between encoding larger gates in decision diagrams and computing successor states. Our hypothesis is that our methods provide a reduction in the size of the decision diagrams and in the run-time.

We implemented our techniques in a new front-end for the tool Q-Sylvan, a DD-based simulator. For empirical evaluation, we implemented several quantum algorithms. This includes an algorithm not previously used for evaluations called Variational Quantum Classifiers (VQC), a machine learning model that learns to categorise data into classes.

Our experiments confirm a reduction in the size of the decision diagrams. We can also conclude that the accuracy of floating point values used to represent the real values needed for quantum computing is a more pressing matter than previously thought.

Contents

Chapter 1

Introduction

Classical computing brought us many benefits. Algorithms will continue to be improved and are faster and more widely used than people could foresee several decades ago. However, some challenges today will never be solved by classical computing, due to a lack of computational power and time. Several problems encountered nowadays are the large amount of real-time data streams [1] which cannot be processed, large datasets which need to be searched, companies having complex models to predict stock market values or machine learning. To tackle these kinds of problems, we need new ways of computation.

Quantum computing

So called quantum computers, which use quantum mechanical phenomena to perform computations, can solve certain problems substantially faster. The quantum algorithm to solve factorisation, invented by P. Shor [2], can give an almost exponential speedup over classical factorisation algorithms. Another example is Grover's algorithm [3], which can perform a brute force database search with a quadratic speedup over classical brute force. Other applications for quantum computers also include quantum chemistry [4], solving linear equations [5] and quantum machine learning [6].

Quantum circuits are used to perform quantum computations [7]. These circuits consist out of a register of qubits, represented by wires, and gate operations placed on those wires. Gates alter the state of the qubit register. Some gates are parallel with other gates (one gate on each wire) and some are sequential (multiple gates on one qubit). For a gate operation to be executed, all preceding gates must be executed.

Unfortunately, we are still in the Noisy and Intermediate-Scaled Quantum (NISQ) era [8], which means the current quantum computers are very small and noisy. Even though there exist several algorithms that work on NISQ computers [9], there are issues, e.g. verifying the correctness of new quantum algorithms, that create a demand for classical simulation of quantum circuits. The most commonly used method for this is the matrix-vector representation, where the quantum gates are represented by matrices and the quantum states are represented by vectors [10, 11, 12]. However, these representations grow exponentially with the number of qubits used.

DD-based simulations of quantum circuits

A Binary decision diagram (BDD) [13] is data structure, defined as a directed acyclic graph (DAG). A BDD can represent boolean functions in a compact form by exploiting redundancies in these functions. Several logical operations can be performed on BDDs, e.g. a conjunction or a disjunction. All of these BDD operations have run-time proportional to the size of the BDD. This data structure seems well suited as an alternate way for representing quantum states and quantum operations, since the matrix-vector representation usually holds a lot of redundant information.

A different decision diagram, called a quantum multi-value decision diagram (QMDD) [14], can be used to represent both the state of a quantum system (state QMDD) and the gate operations (gate QMDD). Here the gate QMDDs resemble the transition relation encoding known from decision diagrams. Using QMDDs to represent non-entangled quantum states can be done in polynomial time and space with QMDDs, while for the vector representation this is exponential time and space. However, there is not much known about the time and space complexity of QMDDs when representing entangled states. Although the worst case complexity still remains exponential, it has been shown for several practically relevant cases that QMDDs yield substantial performance improvements where the complexity is significantly below the exponential upper bound [14]. This is because a lot of redundancies can be exploited using QMDDs, which is not possible with the matrix-vector representation.

We can perform the simulation of a quantum circuit in two ways. We can directly apply an individual gate QMDD to the state QMDD to compute the next state QMDD. We call this *single-gate simulation*. We can also multiply gate QMDDs beforehand, essentially merging multiple transition-relations into one. We call this *multi-gate simulation*. Most QMDD operations have a run-time linear to their size, so compact QMDDs result in faster QMDD operations.

Problem statement

In this thesis we focus on two subtopics. First, we focus on the fact that single-gate simulation is usually realised by simply applying the first possible gate that still has to be applied. However, there are usually several gates that can be applied at the same time, so applying gates can also be done in a different order. The size of decision diagrams may explode in size after applying a single gate. In the same way, a single gate may reduce the size of decision diagrams drastically. Can we substantially speed up DD-based simulations of quantum circuits by scheduling quantum gates for single-gate simulation?

Second, we look at the fact that it is unknown if DD-based simulations of quantum circuits using multi-gate simulation has an advantage over single-gate simulation or vice versa. The approach in [15] shows promising results for multi-gate simulation, where it has a significant speed-up over single-gate simulation. However, this does come at a cost in terms of space needed. Could a balanced simulation method, which uses both multi-gate simulation and single-gate simulation, provide a substantial speed up over either of the simulation methods individually?

Approach

We propose two new simulation methods. The first simulation method revolves around ordering the gates applied using single-gate simulation in a greedy way. We expect that this *greedy simulation* does not decrease the upper size limit when simulating a quantum circuit, but decreases the number of states having a large size.

Second, we propose a *balanced simulation* method that alternates between our greedy simulation and multi-gate simulation approach in [15]. By alternating between these simulation methods, we expect to avoid large state QMDDs using multi-gate simulation while avoiding storing gate QMDDs by using single-gate simulation when the state QMDDs are small. It is difficult to predict which simulation method should be used in which part of the circuit, which is why we used a simple but effective approach, where we place barriers in the circuit when the active simulation method should be switched with the inactive simulation method.

Both simulation methods proposed above, along with single-gate simulation and multi-gate simulation as proposed in [15], are implemented using a QMDD-based quantum simulator called Q-Sylvan. We provide a front-end for Q-Sylvan, which translates quantum assembly language (QASM) to Q-Sylvan operations. We implemented the simulation methods discussed above in this front-end.

Finally, we provide a tool that can be used for implementing new DD-based simulation methods in Q-Sylvan. By implementing an intermediary representation for quantum circuits in the front-end of Q-Sylvan, we provide a structure around which other simulation methods can be implemented that focus on exploiting the inner structure of circuits. Using the new Q-Sylvan front-end, we implemented several new quantum algorithms for experimentation. Finally we performed an empirical evaluation to determine which simulation method works best.

Experiments show that our simulation methods, compared to single-gate and multi-gate simulations, decrease the average size of the QMDDs during a simulation of a quantum algorithm by several factors. However, the run-time of all simulation methods are very similar and no simulation method has a significantly faster run-time than the other simulation methods.

Outline

The outline of this thesis is as follows. Chapter 2 elaborates on the subjects of quantum computing, decision diagrams and the use of decision diagrams for the simulation of quantum circuits. After this, Chapter 3 defines a formal definition of quantum circuits, which is used to describe existing and propose new simulation techniques. Chapter 4 describes the QMDD-based quantum simulator Q-Sylvan and the Quantum Simulator Benchmarking (QSB)-suite. Finally, Chapter 5 reports the results of the experiments done with our proposed simulation methods, along with a summarized conclusion and speculations about potential future work.

Chapter 2

Background

This chapter will explain a number of basic concepts within both quantum computing, as well as decision diagrams. These subjects are covered in Sections 2.1 and 2.2 respectively. After this, Section 2.3 explains in more detail how decision diagrams can be used to represent quantum states and operations, and how these can be used to simulate the results of a quantum circuit.

2.1 Quantum Computing

Quantum computers perform quantum computations which exploit quantum mechanical phenomena, such as entanglement and superposition. Quantum computing makes use of several quantum versions of their classical counterparts, such as bits, gates and circuits [7]. These alterations are formalised using the Dirac notation and linear algebra.

2.1.1 Quantum bits

Whereas classical bits are boolean variables, where a bit can be either 0 or 1, quantum bits, or qubits, can be in a *superposition* of both states. The state of a qubit can be described as a linear combination of the states 0 and 1. States are usually written in Dirac-notation, or Bra-ket notation. In this notation we use the *ket*, written as $|\ldots\rangle$, to represent the state of a qubit ϕ, which is written as $|\phi\rangle$. $|0\rangle$ and $|1\rangle$ are called basis states. The qubit state is always written as a linear combination of the basis states, which is defined as follows:

$$|\phi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \tag{2.1}$$

Here, α_0 and α_1 are *amplitudes*. Each amplitude can be defined by:

$$\alpha_i \in \mathbb{C} \tag{2.2}$$

When *measuring* a qubit, i.e. we look at what state it is in, the qubit collapses to either $|0\rangle$ or $|1\rangle$. The *Born rule* states that the probability a measurement on a quantum system will yield a given result is proportional to the absolute amplitude squared. Collapsing to either state happens with a certain *probability*; the probability of seeing $|0\rangle$ is $|\alpha_0|^2$ and

the probability of seeing $|1\rangle$ is $|\alpha_1|^2$. Since these probabilities need to add up to 100%, the sum of amplitudes is constraint under:

$$\sum |\alpha_i|^2 = 1 \tag{2.3}$$

A register of qubits can also be defined like Equation 2.1, a linear combination of all possible basis states. These basis states can be composed by using the tensor product between states of the individual qubits. For instance, a qubit register ψ containing two qubits is defined as follows:

$$\begin{aligned} |\psi\rangle &= |\psi_1\rangle \otimes |\psi_2\rangle \\ &= (\alpha_0 |0\rangle + \alpha_1 |1\rangle) \otimes (\beta_0 |0\rangle + \beta_1 |1\rangle) \\ &= \alpha_0\beta_0 |00\rangle + \alpha_0\beta_1 |01\rangle + \alpha_1\beta_0 |10\rangle + \alpha_1\beta_1 |11\rangle \\ &= \alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle \end{aligned} \tag{2.4}$$

Mathematically, the state of a qubit register is represented by a numerical vector containing all amplitudes:

$$|\psi\rangle = \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T \tag{2.5}$$

These state vectors grow exponentially in size. Given an n qubit register, the computational basis states of this register are of the form $|x_1x_2 \dots x_n\rangle$, which means the linear combination of all basis states is specified by 2^n amplitudes. This exponential number of amplitudes is what gives quantum computation fundamentally more power than classical computation.

2.1.2 Quantum gates

Quantum gates are used to modify the state of qubit registers. These gates usually operate on a small number of qubits. Quantum logic gates must be reversible, which means there is no loss of information after applying a gate. This is not the case for classical logic gates. For instance, if we look at a classical AND gate and the output would be a 0, there is no way of knowing the input, so we lost this information.

Quantum gates are usually represented by *matrices*. These matrices must be *reversible* to prevent loss of information and *unitary* to preserve the norm. This means that, given a quantum gate U, there exists a gate W which performs the exact opposite of U. W is defined as the conjugate transpose of U, which will be denoted as $U^\dagger$. Here we can conclude that $UU^\dagger = I$ holds.

Given the state of qubit register ϕ as as described by Equation 2.1, we can simulate the result of applying U, resulting in $U |\phi\rangle$. This is done by multiplying the unitary matrix of the single qubit gate with the state vector of the qubit register:

$$\begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \begin{bmatrix} u_{00}\alpha_0 + u_{10}\alpha_1 \\ u_{01}\alpha_1 + u_{11}\alpha_1 \end{bmatrix} \tag{2.6}$$

The result is a new state vector with possibly new amplitudes. When applying gate U to one of the qubits in the qubit register in Equation 2.4, we need to apply an identity gate

to the other qubit. The resulting matrix W to be applied is a *tensor product* of the two matrices. Applying gate U to the second qubit is written as $W = I \otimes U$, where I is the identity gate, resolves as follows:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} u_{00} & u_{01} & 0 & 0 \\ u_{10} & u_{11} & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} u_{00}\alpha_0 + u_{01}\alpha_1 \\ u_{10}\alpha_0 + u_{11}\alpha_1 \\ u_{00}\alpha_2 + u_{01}\alpha_3 \\ u_{10}\alpha_2 + u_{11}\alpha_3 \end{bmatrix} \tag{2.7}$$

Gates can also be *controlled* by one or more other qubits. When a gate is controlled by a qubit register q, it means the gate is applied if all qubits in q are in the $|1\rangle$ state. The matrix of this gate is constructed by taking the tensor of the applied gate if all qubits in q are in the $|1\rangle$ state and the identity gate otherwise. An example of this is when we apply gate U as in Equation 2.6, but now we control it with the first qubit:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ u_{00}\alpha_2 + u_{01}\alpha_3 \\ u_{10}\alpha_2 + u_{11}\alpha_3 \end{bmatrix} \tag{2.8}$$

Like state vectors, these resulting matrices also grow exponentially. A matrix being applied to a register containing n qubits has the size of $2^n \times 2^n$.

2.1.3 Quantum circuits

Quantum computations are done using quantum circuits. These circuits are a sequence of quantum logic gates working on a qubit register. Circuits are visualised using the quantum circuit representation first presented in [16]. First the n-qubit register is set to an *initial state*, usually the all-zero state $|0^n\rangle$.

$$|0^n\rangle = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix}^T \tag{2.9}$$

A *wire* is a representation of a qubit, usually with a ket in front to show the initial state of the qubit. On a wire, gates are portrayed to show it is applied to this qubit. All gates in a column can be combined with a tensor product and applied to the qubit register, as shown in Equation 2.7. The circuits representing Equation 2.6 and Equation 2.8 being applied are shown in Figure 2.1a and Figure 2.1b respectively.

It is important to know how many qubits you need for your computation. Additional qubits which are used to implement desired unitary transformations, are known as ancillary qubits, or *ancillas*.

Current quantum computers do not have many qubits to work with. This is why it is important to recycle your ancillas. Usually it is not possible to reset ancillas by measuring them. This is because you do not know to which state it will collapse and because collapsing can alter your amplitudes, which is often not a desired outcome.

Uncomputation is a technique for cleaning up temporary side effects on ancillas so they can be re-used. Measuring the ancilla will not suffice, since the amplitudes will be reset

(a) The circuit representation of Equation 2.6.

(b) The circuit representation of Equation 2.8.

Figure 2.1: Two example circuit representations.

and there will be loss of information. For example, we want to use the result on the second qubit of Equation 2.8 to control a gate V being applied to the third qubit. If the results of Equation 2.8 are not needed afterwards, we can uncompute these results and recycle the second qubit. The uncomputation in this case is shown in Figure 2.2.

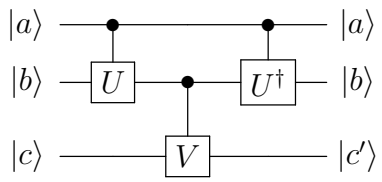


Figure 2.2: An example of using the second qubit as ancilla qubit to store information needed for gate V.

2.1.4 Quantum Assembly Language

Open quantum assembly language (QASM, or openQASM) is an intermediate representation for quantum instructions [17]. The language can be translated to languages of various different back-ends. A *.qasm* file always starts with `OPENQASM X.X;`, where `X.X` is the version. Currently there are only two versions:

- `OPENQASM` 2.0, which is the first version, as described in [17].
- `OPENQASM` 3.0, which is currently in development by Qiskit [18].

In this thesis we will be using `OPENQASM 2.0`. The `.qasm` can contain several different lines of code. Each line is always closed with a semicolon and white spaces are ignored. We can split up the lines into (non-gate) statements and gates. All relevant statements that are possible are listed in Table 2.1.
At the top of a QASM file, the library 'qelib1.inc' is usually imported. This contains all basic quantum gates, so they can be used in the file. All relevant gate-statements are listed in Table 2.2. We can also extend these gate-statements to represent a controlled gate. This is done by adding a 'c' in front of the gate-statement for each controlling qubit, and adding the corresponding qubit as gate-argument. Note that adding more than one 'c' before a gate is not valid in QASM, and thus might not work on some back-ends.

Statement	Description	Example
OPENQASM 2.0;	Denote a file in OpenQASM format	OPENQASM 2.0;
qreg name[size];	Declare a named register of qubits	qreg q[5];
creg name[size];	Declare a named register of bits	creg c[5];
include "filename";	Open and parse another source file	include "qelib.inc";
// Comment text	Comment a line of text	// hello!
measure qubit->bit;	Make a measurement in Z basis	measure q[3]->c[3];
if(bit|creg==int) qop;	Conditionally apply a quantum operation	if(c==5) x q[0];
barrier;	Prevent transformations across this column	barrier;

Table 2.1: Relevant QASM statements that are possible in a `.qasm` file [17].

QASM gates			
gate	**arguments**	**gate**	**arguments**
i	q[...];	h	q[...];
x	q[...];	sx	q[...];
y	q[...];	sy	q[...];
z	q[...];	sz	q[...];
s	q[...];	sdg	q[...];
t	q[...];	tdg	q[...];
rx(α)	q[...];	ry(α)	q[...];
rz(α)	q[...];	cx	q[...], q[...];

Table 2.2: Relevant QASM gates that are possible in a `.qasm` file [17].

2.2 Decision Diagrams

Decision diagrams are a type of data structure that can be used to represent Boolean functions. The decision diagrams that are used in this thesis are quantum multi-value decision diagrams (QMDDs), which are derived from binary decision diagrams (BDDs).

2.2.1 Binary Decision Diagrams

Let us first define a formal description of a Binary Decision Diagram, as stated by Bryant [13]. Let f be a boolean function with n arguments, written as $x_1, \ldots, x_n$. A BDD can be used to represent f. This BDD is a rooted, directed, acyclic function graph with a node set V. This set V contains two types of nodes:

- *non-terminal* nodes, which have as attributes an argument $var(v) \in \{1, \ldots, n\}$

and two children $low(v), high(v) \in V$.

- *terminal* nodes, which have as attribute $value(v) \in \{0, 1\}$.

A BDD having root node v representing a Boolean function f can be recursively defined as follows:

1. If node v is non-terminal with $var(v) = i$, then f can be defined as the Shannon decomposition $f(x_1, \ldots, x_n) = \bar{x}_i f_{low(v)}(x_1, \ldots, x_n) + x_i f_{high(v)}(x_1, \ldots, x_n)$
2. If node v is terminal then $value(v) = f(x_1, \ldots, x_n)$

We can view the set of arguments $x_1, \ldots, x_n$ of f as a description of a path in the BDD. If for node v and argument x_i holds that $var(v) = i$, then we can say that x_i determines the child taken from v. If $x_i = 0$ we take $low(v)$ and if $x_i = 1$ we take $high(v)$. One traversal of the BDD results in one output of f, where we start at the root node v and recursively take one of the children $low(v)$ or $high(v)$ based on the arguments of f until we reach a terminal state. In Figure 2.3 an example is shown of a truth table with its corresponding BDD.

x1	x2	f
0	0	0
0	1	1
1	0	1
1	1	0

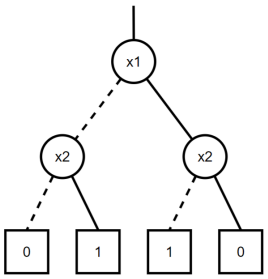


Figure 2.3: A BDD for an XOR function f using two variables. Here the dashed line represents the low edge, and the solid line represents the high edge.

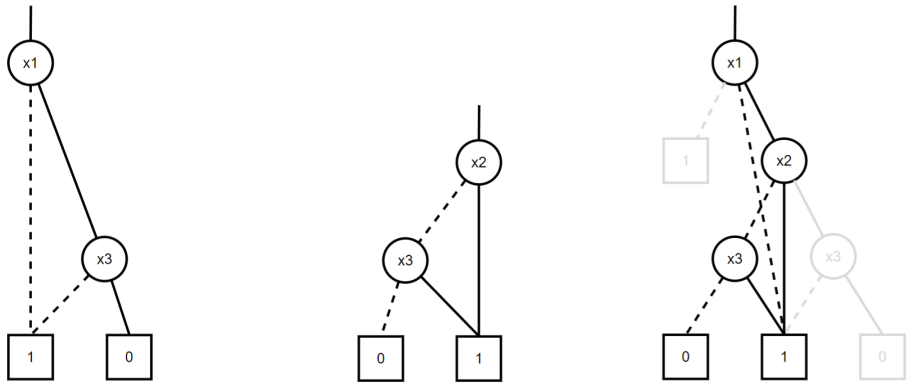
A BDD is said to be *ordered* (OBDD) when for each non-terminal node $v \in V$ holds that $var(v) < var(low(v))$ if $low(v)$ is non-terminal and $var(v) < var(high(v))$ if $high(v)$ is non-terminal.

Applying operations to BDDs

Two BDDs representing the functions $g_1(x_1, \ldots, x_n)$ and $g_2(x_1, \ldots, x_n)$, say G_1 and G_2, can be combined using an operator $<op>$, e.g. the intersection/AND operator or the union/OR operator. The result of an operation applied to these two functions results in a new function $g = g_1 <op> g_2$. The resulting BDD representing the function g, say G, can be created by recursively traversing both G_1 and G_2.

We start at the roots of both BDDs, say v_1 and v_2. We create a new node u and calculate the result of the operation on both nodes: $value(u) = value(v_1) <op> value(v_2)$. If they are both terminal nodes we are done. If at least one of them is a non-terminal node, we must recursively do the above for their children. If $var(v_1) = var(v_2) = i$, then $var(u) = i$. We calculate $low(u)$ by doing the above for $low(v_1)$ and $low(v_2)$, and we

calculate $high(u)$ by doing the above for $high(v_1)$ and $high(v_2)$. If $var(v_1) \neq var(v_2)$ we get the lowest variable, suppose $var(v_1)$. We calculate $low(u)$ by doing the above for $low(v_1)$ and v_2, and we calculate $high(u)$ by doing the above for $high(v_1)$ and v_2. After all nodes in both BDDs have been visited, then G will contain $G_1<op>G_2$. An example of this is shown in Figure 2.4.



(a) A reduced BDD representing $\overline{x_1 \cdot x_3}$.

(b) A reduced BDD representing $x_1 \cdot x_3$.

(c) A BDD representing $\overline{x_1 \cdot x_3} + x_2 \cdot x_3$.

Figure 2.4: The application of an operator (+) on two BDDs. Note that the resulting BDD is not yet reduced.

Most BDD operations have a time complexity proportional to the size of the BDDs being manipulated. This means that BDD operations are exponentially bounded by the number of variables. But as long as the to be represented functions lead to small BDDs, operations on these BDDs are quite efficient.

Reduced ordered BDDs

A full BDD has an exponential number of nodes. However, a BDD can be reduced to a compact function representation by exploiting redundancies in the BDD structure. A BDD is said to be *reduced* (RBDD) if the following two rules have been applied to its graph:

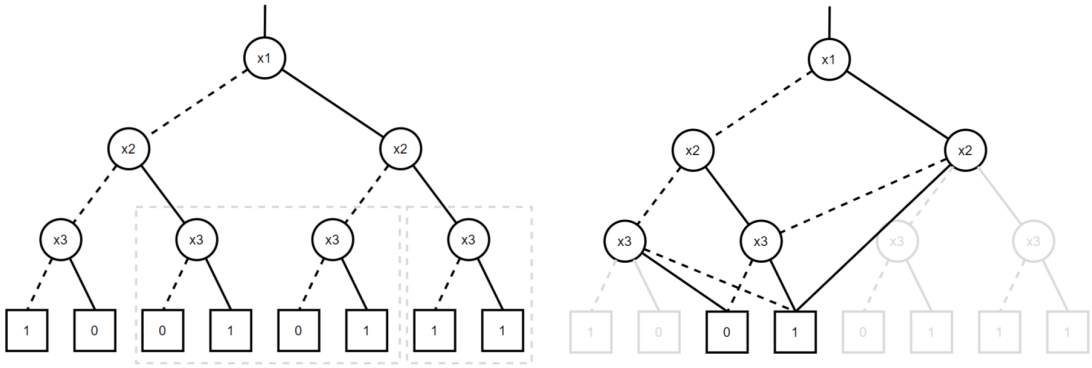
- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

Two (sub)graphs G and G' are isomorphic if there exists a bijective function ω from the nodes of G onto the nodes of G' such that for any node v in G holds that $\omega(v) = v'$. There is currently no known polynomial-time algorithm for checking if two (sub)graphs are isomorphic. However, this can be done efficiently for BDDs. For BDDs, it holds that two (sub)graphs are isomorphic if one of the following rules holds:

- both v and v' are terminal nodes with $value(v) = value(v')$

- both are non-terminal nodes with $var(v) = var(v')$, $\omega(low(v)) = low(v')$ and $\omega(high(v)) = high(v')$

An example of a graph containing isomorphic subgraphs and how to reduce them is shown in Figure 2.5. In this thesis we will only use reduced and ordered decision diagrams. *From now on, when we refer to BDD, we mean an ROBDD.*



(a) A BDD with isomorphic subgraphs.

(b) The same BDD but reduced.

Figure 2.5: Application of the reduction rules onto a BDD. The right box shows a case where rule 1. from the previous paragraph holds. Node 3 is removed and the remaining high edge is connected directly to the terminal node. The left box shows a case where rule 2. of the previous paragraph holds. One of the two subgraphs is removed and the incoming edge of that subgraphs is connected to the remaining subgraph.

A BDD can be reduced at the same time it is created or operations are being applied to it, as described in Section 2.2.1. The reduce operation is almost never used on its own, since doing the reduce operation simultaneously with other operations reduces its complexity significantly.

Time complexity of BDD operations

The BDD operations described above are commonly used to manipulate BDDs. There are some other basic operations which are also commonly used, such as restricting a variable in the function that the BDD represents (restrict), checking if a set of variable assignments satisfies the function (satisfy-one), or counting the number of variable assignments that satisfy the function (satisfy-count). The time complexity of each of these operations is shown in Table 2.3. All BDD operations have a *linear or polynomial time complexity* in the the number of nodes in the BDD.

In the worst-case, the size of a BDD might be exponential in the number of variables. However, BDDs provide a very compact description of many practical functions, such as those encountered in AI and verification [19, 20]

Operation	Time Complexity
Reduce	$O(\|V\| \cdot log\|V\|)$
Apply	$O(\|V_1\| \cdot \|V_2\|)$
Restrict	$O(\|V\| \cdot log\|V\|)$
Satisfy-one	$O(n)$
Satisfy-count	$O(\|V\|)$

Table 2.3: The time complexity of several common BDD operations [13]. Here $|V|$ stands for the size of the set of nodes in our BDD and n stands for the number of variables in the function represented by this BDD.

2.2.2 Algebraic Decision Diagrams

Algebraic decision diagrams (ADDs) [21], also called multi-terminal decision diagrams (MTBDDs) [22], are an extension of the binary decision diagram. Where the BDD has terminal nodes which have as attribute $value(v) \in \{0, 1\}$, the value attribute of terminal nodes in an ADD are defined by $value(v) \in \mathcal{D}$, where often $\mathcal{D} = \mathbb{N}$ or $\mathcal{D} = \mathbb{R}$. The BDD operations and their corresponding time complexities discussed in Section 2.2.1 also apply to ADDs. An example of an ADD is shown in Figure 2.6.

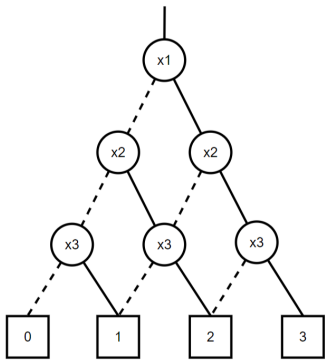


Figure 2.6: An ADD representing the function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$.

2.2.3 Affine Algebraic Decision Diagrams

Affine algebraic decision diagrams (AADDs) [23] are based on ADDs. It is an affine extension to ADDs. A value pair is placed on each edge: an additive value, say c, and a multiplicative value, say b, usually represented as $<c, b>$. AADDs only use one terminal node, with a value of 0. An AADD can be defined as follows:

- If node v is non-terminal with $var(v) = i$, then f can be defined as the Shannon decomposition:
  - $f(x_1, \ldots, x_n) = \bar{x}_i(c_l + b_l f_{low(v)}(x_1, \ldots, x_n)) + x_i(c_h + b_h f_{high(v)}(x_1, \ldots, x_n)$
- If node v is terminal then $value(v) = 0$.

Here, c_h , $c_l \in [0, 1]$ and b_h , $b_l \in (0, 1]$ are real constants representing the additive and multiplicative values of the high and low edge, respectively. Canonicity in decision diagrams is important, because it allows for efficient detection of identical sub-functions during the reduction. To ensure AADDs are canonical, five other constraints are added:

1. G must be ordered.
2. $min(c_h\ c_l) = 0$.
3. $max(c_h + b_h\ , c_l + b_l) = 1$.
4. If $F_h = 0$, then $b_h = 0$ and $c_h > 0$. Similarly for F_l.
5. If $F = 0$ then $b = 0$, otherwise $b > 0$.

An example is shown in Figure 2.7. This AADD represents the same function as used in Figure 2.6, which resulted in a large decision diagram.

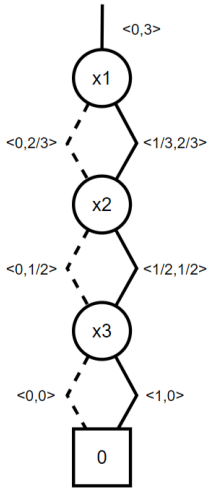


Figure 2.7: An AADD representing the function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$.

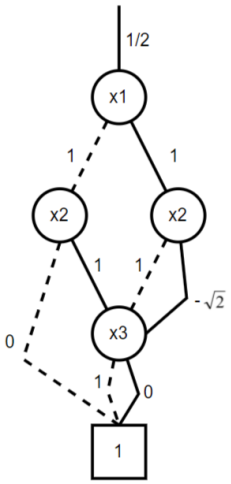


Figure 2.8: A QMDD representing the vector $[0\ 0\ \frac{1}{2}\ 0\ \frac{1}{2}\ 0\ \frac{-1}{\sqrt{2}}\ 0]^T$.

2.2.4 Quantum Multi-value Decision Diagram

Quantum multi-value decision diagrams (QMDD) [24] are a subset of AADDs. In AADDs the value pair on each edge has an additive value, c, and a multiplicative value, b. QMDDs do not use the additive value, which is removed from the edges. Only the multiplicative value is used, and $b \in \mathbb{C}$. Since the last additive value is 0, unlike in AADDs, the value of the single terminal node is set to 1. QMDDs can be used to represent quantum state vectors and gate matrices. In the coming section we will elaborate on this representation, and how it can be used to simulate quantum circuits.

2.3 DD-based simulations of quantum circuits

In this section we will discuss how QMDDs, as described in Section 2.2.4, can be used to simulate the results of a quantum circuit. Simulations of quantum circuits based on QMDDs has first been done by A. Zulehner [14]. Sections 2.3.1 and 2.3.2 will show how a QMDD can represent both quantum state vectors and quantum gates. Section 2.3.3 will also show how these QMDDs can be multiplied, to simulate the application of a gate onto a state vector.

2.3.1 Representing state vectors

As described in Section 2.1.1, the state of a qubit register is usually described using a state vector. This state can also be described using a QMDD. Given a node i in the QMDD, the probabilities of all states in which qubit i is in state $|0\rangle$ are on the low edge, whereas the probabilities of all states in which qubit i is in state $|1\rangle$ are on the high edge. An example in Figure 2.9 shows a QMDD representing a qubit register in the state $|010\rangle$, with numerical representative in the computational basis written as $\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}^T$.

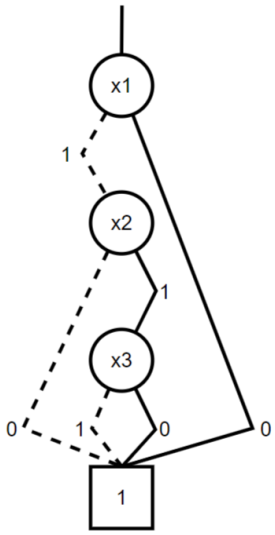


Figure 2.9: A QMDD representing the quantum state $|010\rangle$.

Before simulating a quantum circuit, the state vector is usually initialised in an all-zero state $|0^n\rangle$, where n is the number of qubits. If a different initial state is desired, it can be constructed by applying gates to an all-zero state until the desired state vector is achieved. This will be explained in Sections 2.3.2 and 2.3.3.

Algorithm 1 describes the process of creating an all-zero state vector QMDD having n qubits. The algorithm uses a tensor product between all qubits, where a tensor product between qubit i and qubit $i+1$ in a QMDD means stacking the nodes of these qubits on top of each other.

Algorithm 1: Generating a QMDD representing an all-zero state.

Data: number of qubits
Result: A QMDD representing an all-zero state

```
1 begin
2     create a terminal node v_t with value(v_t) = 1;
3     set v_p to v_t;
4     for i ← n to 1 do
5         create a non-terminal node v with variable(v) = i;
6         set low(v) = v_p and weight(low(v)) = 1;
7         set high(v) = v_t and weight(low(v)) = 0;
8         set v_p = v;
```

2.3.2 Representing gate matrices

A gate matrix for a single qubit gate is a 2×2 matrix. In [14], a single qubit gate QMDD has a node with 4 outgoing edges, one for each entry of the represented 2×2 matrix. Alternatively, a 2×2 matrix can be encoded with two layers of variables, with interleaved primed and unprimed variables. This is very similar to how classical (deterministic) transition relations are often encoded in BDDs [25]. For gate QMDDs the columns and rows are interleaved in a similar way.

A 2×2 matrix is represented by one unprimed node, where the input edge is pointed to, and two primed nodes containing two quadrants of the matrix on their edges each. An example of this is shown in Figure 2.10. **The structure described above will from now on be presented as a single node with 4 outgoing edges.**

$$U = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \tag{2.10}$$

Using the tensor product between single qubit gates acting on different qubits is easily done by stacking each single qubit gate QMDD on top of each other (while keeping the order of the qubits intact). An example of this is shown in Figure 2.11.

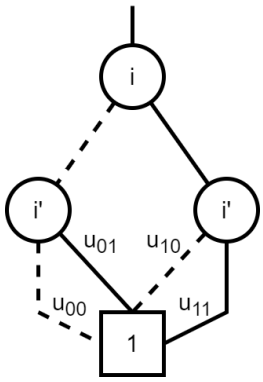


Figure 2.10: A QMDD representing the single qubit gate matrix U on qubit i from Equation 2.10. Note that this is not a normalised QMDD.

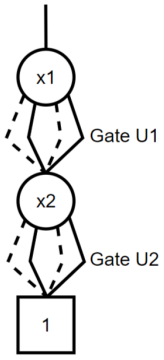


Figure 2.11: A QMDD representing two single qubit gate matrices $U1$ and $U2$ on qubit 1 and qubit 2.

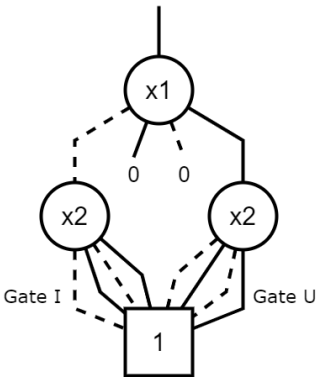


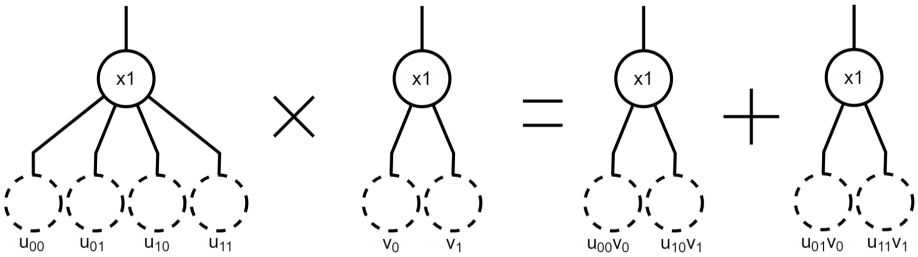
Figure 2.12: A QMDD representing a controlled single qubit gate matrix U on qubit 2, controlled by qubit 1.

There exist multiple qubit gates, which are usually controlled single qubit gates. Any single qubit gate can be controlled by an arbitrary number of qubits. The single qubit gate is only applied if all controlling qubits are in the state $|1\rangle$. Representing these multiple qubit gates as a QMDD is done by splitting the 4 outgoing edges of the controlled qubit. The edge representing matrix entry u_{00} will be connected to the subgraph where gate U is not applied and the edge representing the matrix entry u_{11} will be connected to the subgraph where the single qubit is applied. The other edges will be connected the

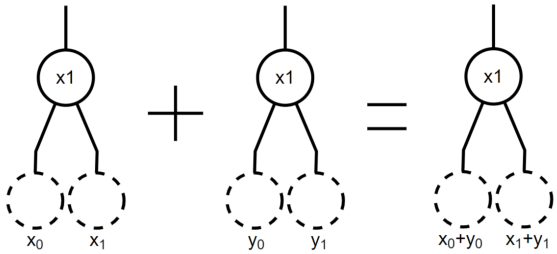
terminal node with an edge weight of 0. These edges are usually represented as seen in the example shown in Figure 2.12.

2.3.3 Simulating a quantum circuit

To create a simulation of a quantum circuit, being able to multiply QMDDs together is crucial. Since a gate matrix can be multiplied with a state vector, as described in Equation 2.6 of Section 2.1.2, a gate QMDD and state vector QMDD must also be able to be multiplied together. Given a gate QMDD of a gate on a single qubit, and a state vector with a single qubit, the low edges of the gate QMDD can be multiplied with the low edge of the state vector QMDD, creating a new node where the results will be stored on the edges. The same is done with the high edges of the gate QMDD and the state vector QMDD. These two nodes must still be added together, which is simply done by adding both low edges of the nodes and both high edges of the nodes together, creating a new low and high edge respectively. The result is the new state vector QMDD. An example of this is shown in Figure 2.13.



(a) The multiplication of a gate QMDD and a state vector QMDD.



(b) The addition of two state vector QMDDs.

Figure 2.13: The tensor product of a gate QMDD with a state vector QMDD.

The method described above can be done for gate QMDDs and state vector QMDDs containing multiple qubits, as long as the QMDDs contain the same number of qubits. Using the same principle, gate QMDDs can also be multiplied with each other, which can result in a gate QMDD representing multiple single qubit gates applied to the same qubit. Gates can be applied directly to the state QMDD in a successive way. We call this **single-gate simulation**. Gate QMDDs resemble the transition relation encoding in BDDs, as described in Section 2.3.2. We can multiply gate QMDDs before applying them to the state QMDD, essentially merging multiple transition-relations into one. We call this **multi-gate simulation**.

2.3.4 Measuring quantum states

After multiplying all gates in the quantum circuit with the initial state vector, the state vector needs to be measured. The easiest way of doing a measurement is on the top qubit, so if any other qubit needs to be measured, we first need to swap this qubit with the top qubit. After this, the probability of measuring either $|0\rangle$ or $|1\rangle$ is defined by Equation 2.11.

$$\begin{aligned} P(q_0 \to |0\rangle) &= \sum_{x \in 0\{0,1\}^{n-1}} |\alpha_x|^2 \\ P(q_0 \to |1\rangle) &= \sum_{x \in 1\{0,1\}^{n-1}} |\alpha_x|^2 \end{aligned} \tag{2.11}$$

After measuring a qubit, the edge weights are altered, where the chosen edge is changed to 1 and the other edge is changed to 0. Finally, the remaining amplitudes must be normalised, which can be done by dividing the incoming edge weight of the root node by $\sqrt{P(q_0 \to |x\rangle)}$, where x is the chosen state.

Chapter 3

New Simulation Algorithms

In order to minimise both the run-time and the needed memory requirements for QMDD-based quantum simulations. Minimising the sizes of all QMDDs needed along the process of those simulations reduces run-time, since most BDD operations have a time complexity proportional to the size of the BDDs being manipulated. Several things can influence the size of these decision diagrams. We can divide our problem into two separate sub-questions:

- Can we substantially speed up DD-based quantum simulations by scheduling quantum gates for single-gate simulation?
- Can a balanced algorithm, which uses both multi-gate simulation and single-gate simulation, have a substantial speed up over either of the simulation methods individually?

3.1 Formal Circuit Notation

We first provide an abstract definition of quantum circuits (Definition 3.1) and the possible operations that can be applied to a quantum circuit (Definition 3.2) in order to describe the existing and new simulation methods described in Sections 3.2, 3.3, 3.4 and 3.5.

Definition 3.1 *A circuit is defined as a set of n qubit wires. Let $q_1, \ldots, q_n$ be n qubits and let $q_{i,1}, \ldots, q_{i,w} \in [w]$ be subsequent locations on the wire of qubit q_i. We can define $Q_j = \{q_{i,j} \mid i \in [n]\}$ as a column across all wires and $Q = \{q \mid q \in Q_j, j \in [w]\}$ as the set of all locations on the wires.*

Definition 3.2 *Let $gate : Q \to G$ be the function that returns the gate positioned at q for $q \in Q$. For gates $g \in G$, we write $Loc(g) = \{q \mid gate(q) = g\}$ as the set of all locations of g. We require all gates $g \in G$ to be limited to columns, i.e. $Loc(g) \subseteq Q_j$ for some $j \in [n]$.*

Definition 3.3 *Let $cc \in [w]^n$ be the circuit counter, i.e. a "program counter" at each wire which tracks all gates that have been executed. Each index in cc is the location of*

the leftmost gate of each wire that has not been executed. A circuit counter cc splits Q into locations prior to cc, written Q_{cc}, and locations after cc, written $Q_{!cc}$, where $cc \subseteq Q_{!cc}$. All gates g where holds that $Loc(g) \subseteq Q_{cc}$ have been executed, whereas all gates g where holds that $Loc(g) \subseteq Q_{!cc}$ have not been executed.

Gates can be multiplied together. We need to keep track of all gates that have not been executed, but have been multiplied with other gates.

Definition 3.4 *Let M be a set of gates that have been merged. Let $gc \in [w]^n$ be the gate counter, i.e. a "program counter" at each wire which tracks all gates that have been merged but not executed. Each index in gc is the location of the leftmost gate of each wire that has not been merged or executed. If $M = \emptyset$, then $gc = cc$. A gate counter gc splits $Q_{!cc}$ into locations prior to gc, written Q_{gc}, and locations after gc, written $Q_{!gc}$, where $gc \subseteq Q_{!gc}$. All gates g where holds that $Loc(g) \subseteq Q_{gc}$ have been merged but not executed, whereas all gates g where holds that $Loc(g) \subseteq Q_{!gc}$ have not been merged nor executed.*

Valid circuit counters depend on the gate function because gates g should be executed in full, i.e. $Loc(g) \subseteq Q_{cc}$ or $Loc(g) \subseteq Q_{!cc}$. Let V be the set of gates that are valid to be executed, i.e. where $Loc(g) \subseteq cc$ holds. Note that cc contains the indices of the leftmost positions in $Q_{!cc}$, which means $cc \subseteq Q_{!cc}$.

Valid gate counters also depend on the gate function in the same manner the circuit counter depends on it. Gates g should be merged in full, i.e. $Loc(g) \subseteq Q_{gc}$ or $Loc(g) \subseteq Q_{!gc}$. Note that gc contains the indices of the leftmost positions in $Q_{!gc}$, which means $gc \subseteq Q_{!gc}$. An example of this is shown in Figure 3.1.

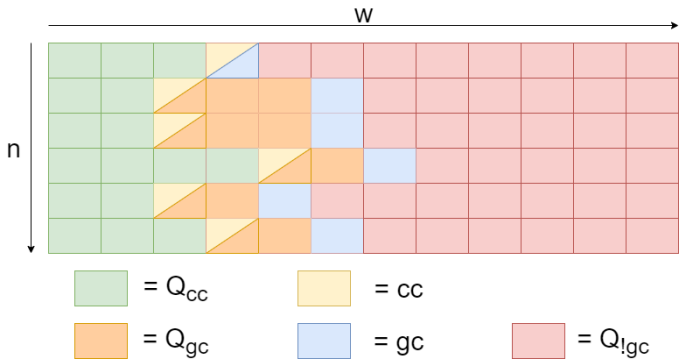


Figure 3.1: A visual representation of all positions in Q of a circuit containing n qubits and w columns. The circuit counter is coloured yellow and the gate counter is coloured blue. The space between the circuit counter and the gate counter is coloured orange, which contains gates that have been merged but not executed, i.e. all gates in M. The green part contains all gates that have been executed. The red part contains all gates that have not been merged nor executed.

Definition 3.5 *Let $ms = (M, gc) \in 2^G \times [w]^n$ be the merge state of a circuit. The merge function be $Merge : (ms, G) \to ms$. A Merge action is a gate g where $Loc(g) \subseteq gc$, such that it transits the system from the current state (M, gc) to a new state (M', gc') with $Merge((M, gc), g) = (M', gc')$. Here $M' = g \cup M$ and $Loc(g) \subseteq Q_{gc'}$ is the state after merging gate g with all other gates in M.*

We need to be able to execute single gates in cc, but we must also be able to apply the resulting gate of the merged set M. Let m be the resulting gate of merging all gates $g \in M$, i.e. $m = \prod_{g \in M} I_1 \otimes \ldots I_{k-1} \otimes g \otimes I_{k+1} \cdots \otimes I_n, k = Loc(g)$ and $Loc(m) = \bigcup_{g \in M} Loc(g)$.

Definition 3.6 *Let $qs \in \mathbb{C}^{2^n}$ be a quantum state $|\psi\rangle$ and let $cs = (qs, cc) \in \mathbb{C}^{2^n} \times [w]^n$ be a circuit state. Let the apply function be $Apply : (s, G) \to s$. An apply action is a gate g where $Loc(g) \subseteq cc$ or $g = m$, such that it transits the systems from the current state (qs, cc) to a successor state (qs', cc') with $Apply((qs, cc), g) = (qs', cc')$. Here, the quantum state qs' is the quantum state qs after applying the gate g, i.e. $|\psi'\rangle = g \times |\psi\rangle$, and cc' is the new circuit counter where $Loc(g) \subseteq Q_{cc'}$. Note that, when $g = m$, it also means that $cc' = gc$.*

Let the initial circuit state be (qs_i, cc_i) with $cc_i = 1^n$ and $qs_i = |0^n\rangle$. Let the initial gate counter be gc_i, where $gc_i = cc_i$. The final circuit state (qs_f, cc_f) is reached when $cc_f = (w+1)^n$, which means $Q_{cc} = Q$ and $Q_{!cc} = \emptyset$. The gate counter can reach the end of the circuit, i.e. $gc_f = (w+1)^n$, after which we cannot further merge gates since $Q!gc = \emptyset$. Note that on all paths, whatever the selection of gates along the way, we end up in the same final circuit state.

3.2 Single-gate simulation

The algorithm implementing single-gate simulation starts at state s_i and while we have not reached s_f, we apply the first gate in the set of all valid gates V. The implementation is shown in Algorithm 2.

Algorithm 2: An algorithm that implements single-gate simulation.

Data: A `.qasm` file as defined in Section 2.1.4

```
1 begin
2     s = s_i;
3     while s ≠ s_f do
4         s = Apply(s, V[0]);
```

3.3 Multi-gate simulation

For multi-gate simulation the `max-size` approach of A. Zulehner [15] is used, of which the implementation is shown in Algorithm 3. This approach consists out of multiplying

gates together until the number of nodes in the resulting gate QMDD reaches a certain threshold. When this happens, the resulting gate QMDD is multiplied with the state vector and the gate QMDD is reset. This is done until all gates in the circuit have been merged and applied to the state vector. From now on, we will refer to multi-gate simulation as *Zulehner's max-size approach*.

Algorithm 3: An algorithm that implements Zulehner's max-size approach.

Data: A `.qasm` file as defined in Section 2.1.4

```
1  gc = gc_i;
2  M = ∅;
3  begin
4      s = (s_i, cc_i);
5      while gc ≠ gc_f do
6          Merge((M, gc), v[0]);
7          if nodecount(m) > threshold then
8              Apply(s, m);
9      if m ≠ ∅ then
10         Apply(s, m);
```

3.4 Optimization of single-gate simulation

The order that gates are applied may have an effect on the size of the intermediary decision diagrams of s. Since the size of decision diagrams may explode after moving a single gate, we want to apply this gate as late as possible. In the same way, a single gate may reduce the size of decision diagrams drastically, which is why we would want to apply this gate as soon as possible. Although this does not decrease the upper limit of nodes when simulating a quantum circuit, the method does decrease the number of intermediary states which contain a large number of nodes. Finding a good algorithm to order the gates may be key to speeding up simulations.

Definition 3.7 *With $nodecount : s \to \mathcal{N}$ we denote the function that returns the number of nodes in the decision diagram that represents s.*

3.4.1 A greedy algorithm

We want to minimise the node count for the successor state of s. This can be achieved by a greedy algorithm. First we initialise two pointers, one to the best state and one the best node count. Then we loop over all valid gates $g \in V$, calculating $Apply(s, g) = s_t$, keeping track of the best s_t with the use of the lowest node count. After we applied all gates in V once, we set our best state as our current state. We do this until we reach the end state s_f. The implementation is shown in Algorithm 4.

Algorithm 4: An algorithm that implements the greedy single-gate simulation.

```
Data: A .qasm file as defined in Section 2.1.4
1  begin
2      s = s_i;
3      while s ≠ s_f do
4          best_state = null;
5          best_count = inf;
6          for g in V do
7              s_t = Apply(s, g);
8              count = nodecount(s_t);
9              if count ≤ best_count then
10                 best_count = count;
11                 best_state = s_t;
12         s = best_state;
```

3.5 Multi-gate vs. single-gate

To implement an algorithm performing multi-gate simulation, as described in Section 3.3, we must also store the intermediary decision diagrams of these gates, instead of only the decision diagram of s.This method might yield large gate representations, but can result in smaller state representations. The approach in [15] shows promising results for multi-gate simulation. However, this does come at a cost in terms of space needed. Combining both simulation methods can lead to using both of their strengths. By alternating between these simulation methods, we can avoid large state QMDDs using multi-gate simulation, while we can avoid storing gate QMDDs by using single-gate simulation when the state QMDDs are small. But first, we must find which regions are better suited for which simulation method. Then, we must find a way to combine both simulation methods, switching based on the regions of the circuit each simulation method is better suited for.

3.5.1 Palindromes

Some regions in the circuit may contain palindromes, for instance where a set of gates is uncomputed (see Section 2.1.3). Combining the gates contained in the palindrome may yield promising results compared to applying each gate separately. We want to use multi-gate simulation on gates inside palindromes, but single-gate simulation on gates outside palindromes. However, there are some complications which makes finding these palindromes difficult. We cannot just use an algorithm that detects palindromes in a string [26], since some gates can be on multiple qubits in the column, which means that from that gate on we must also find palindromes for all qubits connected to that gate.

Besides this is there the problem that some gates commute[1], which means a set of gates that does not hold the characteristics of a palindrome can, in the sense of its result, still be a palindrome.

3.5.2 A balanced algorithm

For this method, we need to alternate between Algorithm 3 and Algorithm 4. But when to do this is still not yet determined. First we need to determine which parts of the circuit work better for which algorithm.

We know that the second half of palindromes, such as uncomputation gates, could possibly reverse some of the complexity that is gained in the first half of the palindrome. This is why merging a palindromes to one resulting gate could lead to skipping complex state vectors. We would like to use Zulehner's max-size approach to merge palindromes and single-gate simulation for all gates outside of palindromes.

Unfortunately, palindromes are very difficult to find. This is why, for the purpose of this thesis, we mark the beginning and end of a palindrome by barriers. Note that two palindromes could be next to each other, which means there must be two barriers in between, one to mark the end of the first palindrome, and one to mark the beginning of the second palindrome. An implementation of this, switching between methods when a barrier is encountered, is shown in Algorithm 5.

Algorithm 5: An algorithm that balances Zulehner's max-size approach and single-gate simulation for quantum circuit simulations.

```
Data: A .qasm file as defined in Section 2.1.4
1  begin
2      s = s_i;
3      is_palindrome = false;
4      while s ≠ s_f do
5          while not barrier do
6              if !palindrome then
7                  s = greedy(s);
8              if palindrome then
9                  s = Zulehner-maxSize(s);
10         is_palindrome = not is_palindrome;
```

[1] Commuting gates can be placed in any order and yield the same results.

Chapter 4

Implementing algorithms in Q-Sylvan

4.1 Q-Sylvan

Q-Sylvan[1] is an extension to the parallel MTBDD library Sylvan[2] [27], adding functionalities for QMDDs. This section gives a brief overview of the features implemented in Q-Sylvan, which are needed to simulate quantum circuits with QMDDs. For the purpose of this thesis, we have implemented a front-end for Q-Sylvan, which makes for easier use of the simulator. This front-end will be described in Section 4.1.4.

4.1.1 Storing complex edge weights

To explain how complex edge weights are added to the decision diagram implementation of Sylvan, it is useful to first explain a bit more about how Sylvan stores decision diagrams without them. Sylvan stores BDD nodes in a hash table where 128 bits are available for a single node [28]. These BDD nodes are effectively a pair of edges (a low and a high edge, 40 bits each) and variable number (24 bits).

For QMDDs, Q-Sylvan maintains a similar bit-structure as Sylvan does for BDDs, but now also allocates a number of bits for the edge weights. Note that storing the actual complex values on the edges would take too much space in Sylvan's current node table implementation: a single complex value consisting of two 64 bit floating point values would already take up 128 bits on its own. Instead the complex values are stored in a separate table, and indices to complex values in this table are stored as part of the nodes in the node table. As a practical optimization we can note that due to normalization of the edge weights, at least one of the edge weights of the child edges of a node will have a weight in $\{0, 1\}$. This makes it so that only a single complex value needs to be stored per pair of edges. As to interfere with Sylvan's parallel computation capabilities as little as possible, the complex values are stored in a lockless hash table, adapted from [29].

Finally, Sylvan's garbage collection, which releases entries in the node hash table if

[1] https://github.com/sebastiaanbrand/q-sylvan
[2] https://github.com/trolando/sylvan

they are no longer used, is extended to also garbage collect the complex value table at appropriate moments.

4.1.2 QMDD operations

Given the two existing simulation methods to simulate quantum circuits, described in Sections 3.2 and 3.3, Q-Sylvan implements the main operations needed to perform both these simulation methods, namely matrix-vector and matrix-matrix multiplication. The functions `qdd_matvec_mult` and `qdd_matmat_mult` are recursive functions as described in Section 2.3.3, which can now benefit from Sylvan's multi-threaded capabilities by computing the results of multiple recursive calls in parallel.

Additionally, Q-Sylvan allows for the application of single qubit gates and controlled gates directly on specified target qubits, via the `qdd_gate` and `qdd_cgate` functions, where the qubits not involved in the gates are left unchanged (as opposed to explicitly being multiplied by identity). This is analogous to the application of partial transition relations with BDDs, where the variables outside the partial relation are left untouched.

Finally, Q-Sylvan allows to do measurements on QMDDs in the computational basis via `qdd_measure_qubit`, which measures one qubit, and `qdd_measure_all`, which measures all qubits. This is implemented as described in 2.3.3, where the QMDD after the measurement is executed is returned. The same QMDD can be measured multiple times if desired, resulting in multiple measurement samples, which would be impossible on a physical quantum computer.

4.1.3 Simulating quantum circuits

To simulate quantum circuits Q-Sylvan has predefined a number of commonly used quantum gates (X, Y, Z, H, S, T, among other), and also supports arbitrary single qubit rotations $R_x(\theta)$, $R_y(\theta)$, and $R_z(\theta)$. All of these single qubit gates can also be used as the target component of controlled gates.

A number of functions to make the construction of desired QMDDs easier have also been added, a few of which are described below.

- `qdd_create_basis_state(int n, bool[] x)` creates the QMDD of an n qubit computational basis state $|x\rangle$.
- `qdd_create_single_qubit_gates(int n, Gate[] U)` create the n qubit matrix QMDD $U_0 \otimes U_1 \otimes \ldots \otimes U_{n-1}$, for a given list of gates U.
- `qdd_tensor_prod(QMDD a, QMDD b)` returns the QMDD encoding $a \otimes b$ for both matrices or vectors, appropriately relabeling the qubit numbers.

4.1.4 Using Q-Sylvan

We have made a front-end Q-Sylvan to make it easier to use. Given a `.qasm` file as described in Section 2.1.4, this front-end can simulate the results of the encoded circuit and return the results. The front-end can be used using terminal commands. The command

`./QASM_to_Sylvan` along with the path to a `.qasm` file runs the encoded circuit using single-gate simulation and prints out the results. Several parameters can also be given, a few of which are denoted below:

- -r runs (int): the number of runs to perform.
- -s seed (int): the randomness seed to be used.
- -m matrix (int): runs the circuit using Zulehner's max-size approach instead. The value given is used as a boundary value for the node count before multiplying with the state vector.
- -g greedy: runs the circuit using a greedy algorithm of the single-gate simulation instead.
- -b balance (int): runs the circuit switching between Zulehner's max-size approach and greedy algorithm instead. The value given is used as a boundary value for the node count before multiplying with the state vector.

For some of the simulation methods a struct is needed that stores the circuit, such that functions can be applied to this **circuit struct**. This struct is a 2-dimensional array as described in Definition 3.1. A gate struct has been defined, as described in Definition 3.2, such that these gates can be placed in the circuit struct. All gates that are implemented by Q-Sylvan have been predefined as gate structs.

4.2 Implementing Quantum Algorithms

To compare different methods for the simulation of quantum circuits, we must use the exact same quantum algorithms to test the speed and memory usage of each simulation method. This means the circuit implementation of a quantum algorithm must always be the same. Several papers testing a simulation method have a non/semi-overlapping set of quantum algorithms, and they do not show (or give a link to) the implementation method of the quantum algorithms, which does not guarantee an identical experimental setup [15, 14].

One solution to this would be a benchmarking tool which can generate circuit implementations of several different quantum algorithms in a consistent way. This benchmarking tool should take in some set of parameters for a quantum algorithm, which can be presented in papers, such that everyone can use an identical circuit implementation as used in those papers.

We have created a Quantum Simulator Benchmarking suite[3], or QSB-suite, which is a python library. It contains several classes, one class for each implemented quantum algorithm. Each class has a generate function. Calling this function, along with the needed parameters, generates a `.txt` file containing the QASM code describing a circuit that implements the algorithm of the class, based on the given parameter values. Currently the quantum algorithms that are implemented are:

[3]https://github.com/MSwenne/QSB-Suite

- A variational quantum classifier (VQC)
- Grover's algorithm [3]
- Grover's algorithm applied to K-SAT
- A Supremacy circuit [30]
- A palindrome circuit generator (synthetic)

Variational quantum classifiers

The variational quantum classifier (VQC) is a quantum machine learning circuit. A VQC circuit can consist out of several different layers. Two basic layers are included in the benchmarking suite: the entanglement layer and the parametrized layer. If needed, the benchmarking suite can easily be extended such that it also supports other VQC layers. The entanglement layers encodes a data point into the qubits. After this, the parametrized layer alters the qubits using rotation gates. The result can be split based on some formula, and result in a predicted label. The rotations in the parametrized layer are usually classical variables that can be altered using machine learning. This way, the circuit "learns" to tune the rotation variables by training on datapoints, such that it predicts the right labels. Note that the QSB-suite only generates the circuit given the rotation variables, whereas the training algorithm must be implemented by the user.

Grover's algorithm

Grover's algorithm is a quantum search algorithm that can, with high probability, find specific answers to black box functions. Given a function $f : \{0, 1, \ldots, 2^n - 1\} \to \{0, 1\}$, Grover's algorithms can find specific results that satisfies $f(x) = 1$. We initialise n qubits in a superposition, giving every possible bitstring an equal chance of appearing. Then the function is encoded into the circuit. This part is called the "oracle". The amplitudes of all bitstrings that satisfy f are negated. Then we use a mean inversion, which makes the amplitudes of the satisfying bitstrings larger, and the amplitudes of the other bitstrings smaller. We need to do the oracle step and the mean inversion step several times, based on the number of qubits and the number of satisfying answers. After the correct number of iterations, the satisfying bitstrings have a high probability of appearing.

Grover's algorithm applied to K-SAT

The boolean satisfiability problem (SAT) is the problem of determining there exists a satisfying solution to a boolean formula. The K-SAT problem only uses boolean formulas in conjunctive normal form (CNF). In a CNF formula, the variables are grouped using the logical 'or'. These groups are called clauses, and all clauses are grouped using a logical 'and'. Each clause can have at most K variables. An example where $K = 3$ is described by Formula 4.1. A K-SAT formula can be encoded into the "oracle" of Grover's algorithm, which returns all satisfying solutions with a high probability.

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge \cdots \wedge (\neg x_2 \vee \neg x_3 \vee x_1) \tag{4.1}$$

A supremacy circuit

Quantum supremacy is a term describing that quantum computer can perform a computational task beyond the capabilities of state-of-the-art classical computers. This circuit tests the capabilities of a quantum computer. It generates a pseudo-random quantum state [30].

Synthetic benchmark: palindrome circuits

The circuit containing palindromes is generated using the random circuit definition above. We create a palindrome by placing gates using the random circuit. Then we place a gate that is being controlled by all other qubits. Now we place the same gates as we placed before the controlled gate, but in reversed order and as a conjugate transpose version. This can be done for one or more times. This circuit is specifically implemented for our experiments to test the efficiency of our balanced algorithm on palindromes.

Chapter 5

Experimental results

In order to test the capabilities of all four simulation methods described in Chapter 3, we performed experimental evaluations using the QSB-suite described in Section 4.2. These evaluations were done using the following quantum algorithms:

- A variational quantum classifier (VQC)
- Grover's algorithm [3]
- Grover's algorithm applied to K-SAT
- A Supremacy circuit [30]
- A palindrome circuit generator

The results of these experimental evaluations are shown in the coming sections. All quantum algorithms were run using each of the four simulation methods. We divided the experiments in two parts:

1. Record the number of nodes of all QMDDs needed while simulating the results of a circuit;
2. Record the time needed to simulate the results of a circuit.

The reason we divided the experiments is because the time taken counting the number of nodes in a QMDD should not be included in the time to run a simulation.

5.1 Parameter settings and hardware specifications

For each circuit several parameters need to be set. Each of the coming subsections will explain what parameter settings were used for the circuits.

All experiments are run on Windows using Windows subsystem for Linux (WSL2) with an Intel i7-9750H CPU having 6 cores and 16GB RAM.

Each simulation method runs one algorithm several times, using a scaling parameter. Which parameter is used for each algorithm is defined in its corresponding section below.

For each value in the defined range we do one run for counting the number of nodes, and 10 runs for measuring the time, of which we take the average as the resulting time taken to run the algorithm.

For each run that is done as described above, we do 10 runs instead, each with a different randomness seed. This randomness seed is needed since some circuits are generated using some form of randomness, which will result in easier or more difficult circuit simulations. From the results of these 10 runs we take the average as the final results of the run. This means 10 runs are done for the node count, resulting in one array that holds the node count along the circuit process. For the time runs, this means we do 100 runs, resulting in 10 average time measurements.

The average node count of the resulting array will be plotted over the range of the scaling parameter, with error bars showing the standard deviation of the node count during the run. The 10 average time measurements are also plotted over the range of the scaling parameter, with error bars showing the standard deviation of these 10 measurements.

5.2 Grover's search

For Grover's search we have two parameters, the number of qubits and the oracle. We ran Grover's search using the qubit number as the scaling parameter, while using the randomness seed as the oracle. This is done because different oracles cause different oracle implementations, which results in easier or more difficult circuit simulations.

For each simulation method we started at 5 qubits, increasing the number of qubits by one until we reach 21 qubits or until the simulator throws an error due to limited memory space. We will explain these memory errors in more detail in the Discussion. It is evident from Figure 5.1 that there is a sharp peak in the greedy simulation method. We will also address this in the Discussion.

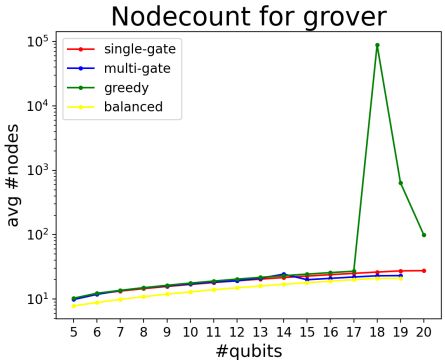


Figure 5.1: The average node count of the state QMDD while simulating the results of Grover's algorithm using multiple different oracles per data point ranged over a certain number of qubits.

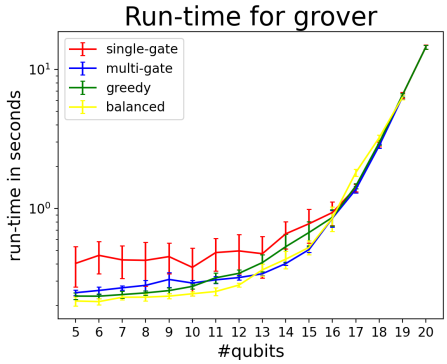


Figure 5.2: The average time needed for each simulation method to simulate Grover's algorithm using multiple different oracles per data point ranged over a certain number of qubits.

5.3 Grover's search for 3-SAT

For Grover's search applied to K-SAT we have several parameters, the number of qubits, the oracle, the value for K and the number of answers the oracle gives. We ran Grover's search using the qubit number as the scaling parameter. For these experiments, we used $K = 3$ and a 3-SAT formula that always results in 1 answer as oracle. Unfortunately, it is difficult to generate random formulas that apply to these constraints, which is why we fixed the oracle to one formula per qubit value for these experiments. The oracle is generated by the number of qubits, always starting with: $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$. This fixes the values x_1, x_2 and x_3 to be `true`. After this, we can append values by simply appending one clause $(\neg x_1 \vee \neg x_2 \vee x_i), i \in [4, n]$. Since x_1 and x_2 are fixed to be `true`, this clause can only be satisfied if x_i is also `true`. This results in an oracle containing the least amount of clauses that result in one solution.

For each simulation method we started at 5 qubits, increasing the number of qubits by one until we reach 21 qubits or until the simulator throws an error due to limited memory space. Note that, for each clause added an extra ancilla is needed. So by increasing the number of qubits by one, the total number of qubits is actually increased by two.

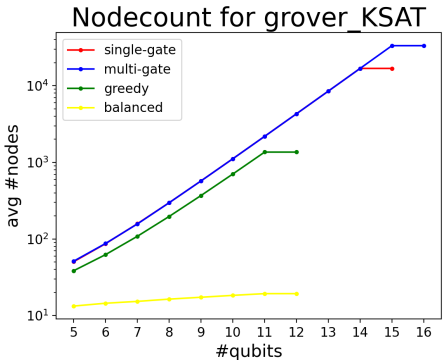


Figure 5.3: The average node count of the state QMDD while simulating the results of Grover's algorithm applied to 3-SAT using a certain number of qubits.

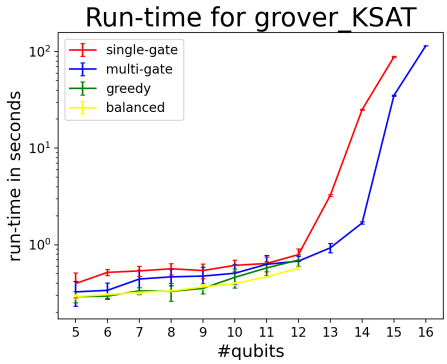


Figure 5.4: The average time needed for each simulation method to simulate Grover's algorithm applied to 3-SAT using a certain number of qubits.

5.4 Palindrome circuits

The palindrome circuits can be generated with several parameters, namely the number of qubits, the number of palindrome sections, the number of gates per palindrome section (and reversed, so twice as many), the ratio of cZ gates and the set of gates to choose from. Again we take the number of qubits as the scaling parameter. The universal gate set is used, containing the gates X, Y, Z, H and t. We generate 5 palindrome sections containing 20 gates (and reversed, so a total of 40 gates per section) and a cZ ratio of

0.3. Here we use a different randomness seed since all generated gates and cZ gates are chosen randomly.

For each simulation method we started at 15 qubits, increasing the number of qubits by one until we reach 41 qubits or until the simulator throws an error due to limited memory space.

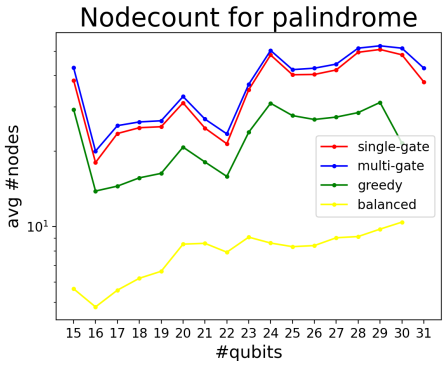


Figure 5.5: The average node count of the state QMDD while simulating a palindrome circuit using a certain number of qubits.

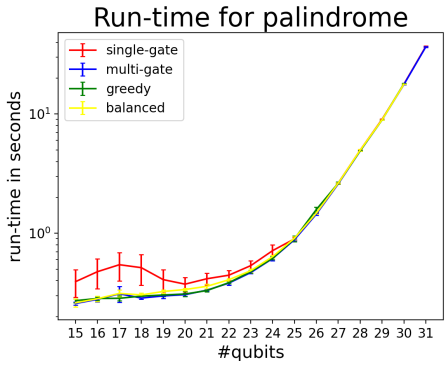


Figure 5.6: The average time needed for each simulation method to simulate a palindrome circuit using a certain number of qubits.

5.5 Supremacy circuits

For the supremacy circuits we choose the depth as the scaling parameter. The number of qubits is fixed, as defined by [30]. We have chosen to use 20 qubits. Several gates are generated randomly, which is why we will use a randomness seed.

For each simulation method we started at a depth of 5, increasing the depth by one until we reach depth 21 or until the simulator throws an error due to limited memory space.

5.6 Variational Quantum Classifiers

For our experiments, training a variational quantum classifier (VQC) is not in our interest. We just want to test the performance of simulating the results of circuits used for VQCs. This is why we use random rotation parameters for each rotation gate. For this, a randomness seed is used. We have fixed the number of entanglement layers and the number of parametrized layers both to 3. The rotation gates used in the parametrized layers are `rY` and `rZ`. The experiments on the VQC circuits are done using the number of qubits as the scaling parameter.

For each simulation method we started at 5 qubits, increasing the number of qubits by one until we reach 21 qubits or until the simulator throws an error due to limited memory space.

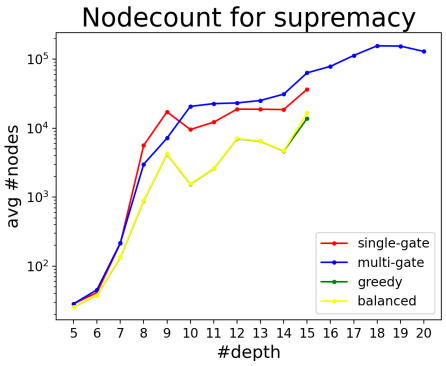


Figure 5.7: The average node count of the state QMDD while simulating a supremacy circuit using a certain depth.

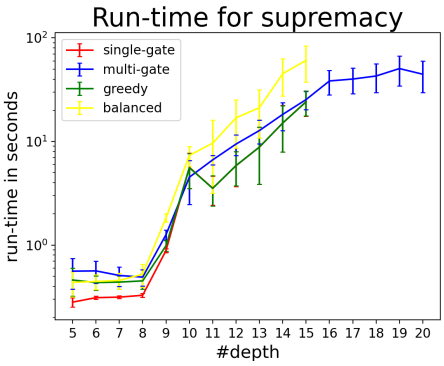


Figure 5.8: The average time needed for each simulation method to simulate a supremacy circuit using a certain depth.

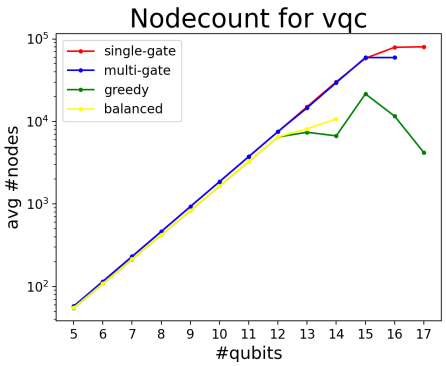


Figure 5.9: The average node count of the state QMDD while simulating a VQC circuit using a certain depth.

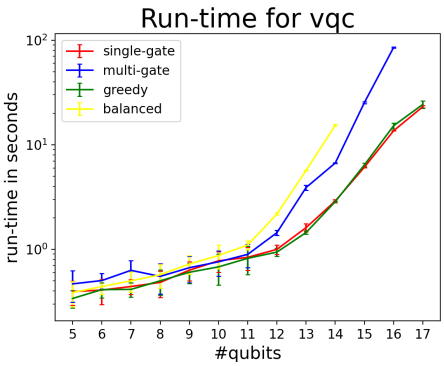


Figure 5.10: The average time needed for each simulation method to simulate a VQC circuit using a certain depth.

5.7 Analysing results

The results shown in each subsection above show several different cases. For Grover's algorithm the balanced simulation method stands out with a smaller average node count than the rest. If we look at the run-time of the simulation, there is unfortunately no large distinction between them, although the balanced simulation method is a bit faster. As for our greedy simulation method, it unfortunately does not give an advantage over the rest for Grover's algorithm.

The same can be seen at Grover's algorithm applied to 3-SAT, although here the difference in the average node count is much more visible. Based on the results of the average node counts, the balanced simulation method is the optimal simulation method to use for this Grover's algorithm applied to 3-SAT. However, if we look at the run-time of the simulation, there is again no large distinction between each method, but the balanced simulation method still remains the fastest of all simulation methods.

The average node count for the palindrome circuit shows promising results for both our proposed simulation methods. The greedy simulation method has a lower average nodes

than single-gate simulation or Zulehner's max-size approach and the balanced simulation method has the lowest average node count. When looking at the run-time, single-gate simulation is the least effective. Other simulation methods have a similar run-time and no simulation method really stands out.

The simulation of the supremacy circuit shows that the average node count of both the greedy and the balanced simulation method are better than single-gate simulation and Zulehner's max-size approach. However, the balanced simulation method has a higher run-time, whereas the greedy simulation method has a low run-time.

The node count for the variational quantum classifiers (VQC) is very similar in each simulation method, although we do see more improvement in our simulation methods. The run-time results for simulating a VQC show that single-gate simulation and the greedy simulation method have the best run-times.

Chapter 6

Conclusion and discussion

The difficulty of designing circuits in these early generations of quantum computers creates a demand for the classical simulation of quantum computations. This thesis has investigated existing simulation methods based on decision diagrams. We proposed two new simulation methods for DD-based simulation of quantum circuits. We have investigated if our simulation methods provide an advantage over the existing simulation methods by performing an empirical evaluation on several quantum algorithms that were implemented in a benchmarking suite. This includes a quantum algorithm not previously used for evaluations called Variational Quantum Classifiers (VQC).

> **Conclusion:** *Our results indicate that our proposed balanced simulation method provides an advantage in terms of computational space. However, this is not the case in terms of run-time, since the run-times of our simulation methods unfortunately do not differ very much from the run-times of the existing methods.*

Limitations: We were missing several datapoints in our results, which was due to a consistent memory error stating our complex value table was full. When looking at Figure 5.1, we can also see a clear spike in the average number of nodes during the greedy simulation. We speculate that is can be caused by the common floating-point accuracy problem. The complex edge weights are represented using floating-point values, which only have a certain amount of precision. If we increase the precision, there is a chance arithmetic errors occur. These arithmetic errors can cause a significant increase in the size of the QMDDs.
Future: It is promising to gain a deeper understanding of the floating-point accuracy problem. Niemann [31] proposes to work with algebraic decision diagrams (ADDs) instead of QMDDs. If ADDs yield promising results in solving the floating-point accuracy problem, it might be beneficial to extend Q-Sylvan to be also able to simulate quantum computations using ADDs.

The Q-Sylvan front-end contains a circuit structure. This structure is suited to be used by simulation methods that can exploit the inner structure of a circuit. For instance, Grover's algorithm does several iterations of the same circuit part. Merging the gates once and re-using the resulting gate for each iteration can result in a faster simulation of Grover's algorithm.

With this thesis we hope to have shown several promising DD-based simulation methods, which have an advantage over matrix-vector simulations. This advantage leads to faster simulations of quantum circuits, which allows for easier circuit designing and quantum algorithm verification.

Bibliography

[1] J. Gama and P.P. Rodrigues. Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer Berlin Heidelberg, 2007.

[2] P.W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, Jan 1999.

[3] L.K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[4] B.P. Lanyon, J.D. Whitfield, G.G. Gillet, M.E. Goggin, M.P. Almeida, I. Kassal, J.D. Biamonte, M. Mohseni, B.J. Powell, M. Barbieri, A. Aspuru-Guzik, and A.G. White. Towards quantum chemistry on a quantum computer. Nature Chemistry 2, 106 - 111 (2009), 2009.

[5] A.W. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for solving linear systems of equations. Phys. Rev. Lett. vol. 15, no. 103, pp. 150502 (2009), 2008.

[6] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, September 2017.

[7] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2009.

[8] J. Preskill. Quantum computing in the nisq era and beyond. Quantum 2, 79 (2018), 2018.

[9] K. Bharti, A. Cervera-Lierta, T.H. Kyaw, T. Haug, S. Alperin-Lea, A. Anand, M. Degroote, H. Heimonen, J.S. Kottmann, T. Menke, W. Mok, S. Sim, L. Kwek, and A. Aspuru-Guzik. Noisy intermediate-scale quantum (nisq) algorithms, 2021.

[10] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. Quipper: A scalable quantum programming language. ACM SIGPLAN Notices 48(6):333-342, 2013, 2013.

[11] D. Wecker and K.M. Svore. Liqui—¿: A software design architecture and domain-specific language for quantum computing, 2014.

[12] M. Smelyanskiy, N.P.D. Sawaya, and A. Aspuru-Guzik. qhipster: The quantum high performance software testing environment, 2016.

[13] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[14] A. Zulehner and R. Wille. Advanced simulation of quantum computations, 2017.

[15] A. Zulehner and R. Wille. Matrix-vector vs. matrix-matrix multiplication: Potential in DD-based simulation of quantum computations. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, March 2019.

[16] A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P.W. Shor, T. Sleator, J.A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, Nov 1995.

[17] A.W. Cross, L.S. Bishop, J.A. Smolin, and J.M. Gambetta. Open quantum assembly language, 2017.

[18] G. Aleksandrowicz et al. Qiskit: An open-source framework for quantum computing, 2019.

[19] G. Audemard and L. Sais. Sat based bdd solver for quantified boolean formulas. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 82–89, 2004.

[20] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. In *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV*, 1995.

[21] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 188–191, 1993.

[22] M. Fujita, P.C. McGeer, and J.C.Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1993.

[23] S. Sanner and D. McAllester. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In *In IJCAI*, pages 1384–1390, 2005.

[24] D.M. Miller and M.A. Thornton. Qmdd: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, pages 30–30, 2006.

[25] Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696, October 2016.

[26] G. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22:346–351, 1975.

[27] T. Van Dijk, A. Laarman, and J. Van De Pol. Multi-core bdd operations for symbolic reachability. *Electronic Notes in Theoretical Computer Science*, 296:127–143, 2013.

[28] T. van Dijk and J. van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696, 2017.

[29] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Formal Methods in Computer Aided Design*, pages 247–255. IEEE, 2010.

[30] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven. Characterizing quantum supremacy in near-term devices. Nature Physics 14, 595-600 (2018), 2016.

[31] P. Niemann, A. Zulehner, R. Drechsler, and R. Wille. Overcoming the tradeoff between accuracy and compactness in decision diagrams for quantum computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4657–4668, Dec 2020.