Informatics

Design und Implementierung des rekursiven Bernstein-Vazirani-Quantenalgorithmus

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Rainer Zachmann, BSc
Matrikelnummer 01526652

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Uwe Egly

Wien, 9. Mai 2023

Rainer Zachmann	Uwe Egly

Informatics

Design and Implementation of the Recursive Bernstein–Vazirani Quantum Algorithm

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Rainer Zachmann, BSc
Registration Number 01526652

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Uwe Egly

Vienna, 9th May, 2023

Rainer Zachmann   Uwe Egly

Erklärung zur Verfassung der Arbeit

Rainer Zachmann, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. Mai 2023

Rainer Zachmann

Kurzfassung

Der Quantenalgorithmus von Bernstein und Vazirani stellt einen Meilenstein in der Komplexitätsforschung von Quantencomputern dar. Es konnte gezeigt werden dass Quantencomputer das Bernstein-Vazirani-Problem, auch bekannt als diskretes Fourier-Sampling, schneller lösen können, als es mit klassischen Rechnern möglich ist.

In der nicht-rekursiven Basisvariante konnte eine lineare Verbesserung durch Quantencomputer gegenüber klassischen Algorithmen beschrieben werden. Darüber hinaus konnte mittels Rekursion eine superpolynomielle Verbesserung gezeigt werden, die Quantenalgorithmen eine mächtigere Komplexitätsklasse als klassischer Programmierung zuordnet.

Die rekursive Variante des Bernstein-Vazirani-Problems wird in dieser Arbeit neu formuliert und nach Problemgröße und Rekursionstiefe parametrisiert. Danach wird der Algorithmus sowohl in der stark typisierten Quantenprogrammiersprache Silq als auch in Qiskit, einer zeitgemäßen Bibliothek zur Quantenprogrammierung, implementiert. Das Programm kann in maschinenlesbarer Form für einen Quantencomputer exportiert oder auf klassischen Rechnern emuliert werden.

Damit können die Ergebnisse von Bernstein und Vazirani bezüglich der Komplexität prinzipiell auch auf Quantencomputern überprüft werden. In dieser Arbeit wird ein neuer Ansatz präsentiert, das rekursive Bernstein-Vazirani-Problem mit einem Kontrollargument zu formulieren. Mit dieser Formulierung wird die Implementierbarkeit des Algorithmus wesentlich verbessert.

Abstract

The Bernstein–Vazirani quantum algorithm serves as an important milestone in the assessment of the computational power of quantum computers. Bernstein and Vazirani showed that the Bernstein–Vazirani problem, alias discrete Fourier sampling, could be solved faster by a quantum algorithm than by any classical formulation.

While the non-recursive, base variant shows a linear improvement of the quantum algorithm over classical ones, a superpolynomial improvement could be achieved by applying recursion to the problem. This recursive Bernstein–Vazirani problem separates the complexity classes of quantum and classical computation.

The recursive variation of the Bernstein–Vazirani problem is reformulated and parametrised by problem size and recursion depth. It is then designed in the strongly typed quantum programming language Silq and also implemented in the state-of-the-art quantum library Qiskit. This program can be exported into machine-readable quantum assembly suitable for a quantum computer or emulated on classical computers.

Using these results, the complexity results of Bernstein and Vazirani can in principle be verified on quantum hardware. We present a new variant of formulating the recursive problem using a control argument that greatly simplifies actual implementations.

Contents

CHAPTER 1

Introduction

The Bernstein–Vazirani problem, also known as discrete Fourier sampling, is well-known in quantum computing. Its basic, non-recursive formulation is an adaption of the Deutsch–Josza problem to a parity function of a secret bit string which uses only a single oracle function call to extract said secret. The relation between the Deutsch–Josza and Bernstein–Vazirani problems is discussed in [SM12, p. 1454].

However, the Bernstein–Vazirani problem uses recursion to further increase the computational complexity. There is currently no implementation of this recursive variant in a quantum programming language available.

Whilst the non-recursive variant is a common example problem for introducing the capabilities of a new quantum programming language, the full recursive problem is often left out, even though its implementation can be a prime example of how to achieve recursion in the demonstrated language.

The result of this thesis is a new formal definition of the recursive Bernstein–Vazirani problem and algorithms for solving it both classically and on a quantum computer. The quantum solution is also presented as a quantum circuit diagram and implemented in the quantum programming languages Silq and Qiskit.

The recursive Bernstein–Vazirani problem provides a separation between computational complexity classes. It is designed in a way that allows quantum computers to solve it in a linear number of calls of a given oracle while a classical computer has to perform a superpolynomial amount of oracle calls.

As such, the problem belongs to the class BQP of problems solvable in polynomial time by a quantum computer relative to an oracle. Bernstein and Vazirani show in [BV97] that it is not possible to solve the problem by classical computation in polynomial time. That means that the class BPP of problems solvable with bounded error probability in polynomial time, again relative to an oracle, by a classical machine is distinct from BQP.

The structure of this thesis is as follows. The next chapter presents the mathematical foundations and notations that are used in this work. These preliminaries define operations on bit strings as well as on quantum states.

Chapter 3 then shows the Bernstein–Vazirani problem and its solution algorithms while discussing different ways of applying recursion. Both classical and quantum algorithms for solving this problem are developed.

The final recursive variant and the non-recursive base case are then implemented in Chapter 4 in the quantum programming languages Silq and Qiskit.

Finally, the last chapter compares the running time between the classical and quantum algorithms.

CHAPTER 2
Preliminaries

In this chapter we are going to establish the prerequisite notations and definitions used in the subsequent chapters of this thesis. The usage of bit strings is formalised as a Boolean vector space, the definition of quantum states is formed in terms of complex Hilbert spaces and quantum circuits are presented as a practical model for quantum computation, which can be visualised as diagrams.

2.1 Boolean Vector Space

Let $\mathbb{B}$ denote the field of integers modulo two, that is the quotient ring $\mathbb{Z}/2\mathbb{Z}$ with two elements 0 and 1. The field operations of addition and multiplication are defined as exclusive disjunction and logical conjunction, respectively. The values of these operations are defined for $x \in \mathbb{B}$ as

$$0 + x = x + 0 = x \qquad \text{and} \qquad 1 + 1 = 0 \tag{2.1}$$

with the additive identity 0 as well as

$$1 \cdot x = x \qquad \text{and} \qquad 0 \cdot x = 0 \tag{2.2}$$

with the multiplicative identity 1. Other notational variants include $x + y = x$ XOR y and $x \cdot y = x$ AND y, respectively.

Since these operations are associative, summation and products of finite sequences $(x_i)_{i \in I}$ can be written as

$$\sum_{i \in I} x_i \qquad \text{and} \qquad \prod_{i \in I} x_i, \tag{2.3}$$

respectively, where for $I = \emptyset$ the empty sum is defined to be the additive identity 0 and the empty product to be the multiplicative identity 1.

Using this Boolean field, for a natural number n the n-dimensional vector space of n-tuples over $\mathbb{B}$ is denoted as $\mathbb{B}^n$. For consistency with zero-based arrays in programming languages, the components of these vectors are numbered from 0 to $n-1$. The notation $\boldsymbol{x}[i]$ shall denote the ith component of $\boldsymbol{x} \in \mathbb{B}^n$ for i in this range from 0 to $n-1$.

The Boolean vector space $\mathbb{B}^n$ is defined with componentwise addition $\boldsymbol{x}+\boldsymbol{y}$ that is defined for i from 0 to $n-1$ as

$$(\boldsymbol{x}+\boldsymbol{y})[i] = \boldsymbol{x}[i] + \boldsymbol{y}[i] \tag{2.4}$$

and scalar multiplication defined as

$$1 \cdot \boldsymbol{x} = \boldsymbol{x} \qquad \text{and} \qquad 0 \cdot \boldsymbol{x} = \boldsymbol{0} \tag{2.5}$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{B}^n$ where $\boldsymbol{0}$ is the zero vector whose components are all 0.

A basis of a vector space is a set of vectors that are linearly independent and span the vector space. In Boolean vector spaces a linear combination of vectors is simply the sum of a subset of vectors.

A set B of the Boolean vectors is linearly independent if and only if the only subset $S \subseteq B$ whose sum of vectors yields the zero vector

$$\boldsymbol{0} = \sum_{\boldsymbol{b} \in S} \boldsymbol{b} \tag{2.6}$$

is the empty set $S = \emptyset$.

B is spanning the Boolean vector space if and only if any vector $\boldsymbol{x} \in \mathbb{B}^n$ can be written as the sum of a subset $S \subseteq B$

$$\boldsymbol{x} = \sum_{\boldsymbol{b} \in S} \boldsymbol{b} \tag{2.7}$$

of these vectors.

The standard basis of $\mathbb{B}^n$ consists of n vectors $\boldsymbol{e}_i$ that have the ith component equal to 1 and the others all set to 0. That is

$$\boldsymbol{e}_i[j] = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{2.8}$$

for all i and j from 0 to $n-1$.

Furthermore, let the bitwise inner product in $\mathbb{B}^n$ be defined as

$$\boldsymbol{x} \odot \boldsymbol{y} = \sum_{i=0}^{n-1} \boldsymbol{x}[i] \cdot \boldsymbol{y}[i] \tag{2.9}$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{B}^n$. It assigns to each pair of Boolean vectors a Boolean value, namely the exclusive disjunction of the components of their componentwise logical conjunction. This is also known as the parity of the bitwise conjunction $\boldsymbol{x}$ AND $\boldsymbol{y}$.

2.2 Hilbert Space

The following definitions for notation used for Hilbert spaces are commonly used in literature. Nielsen and Chuang give more detailed descriptions and explanations in their book entitled *Quantum Computation and Quantum Information* [NC10].

2.2.1 Inner Product Space

A Hilbert space V is a complex vector space whose vectors will be written in Dirac notation as kets $|\varphi\rangle \in V$. It has an inner product $p_V \colon V \times V \to \mathbb{C}$ that for two kets $|\varphi\rangle$ and $|\psi\rangle$ returns a complex number, symbolically written in angle brackets as

$$\langle\varphi|\psi\rangle = p_V(|\varphi\rangle, |\psi\rangle). \tag{2.10}$$

The inner product in our definition is linear in its second argument

$$p_V(|\varphi\rangle, a|\chi\rangle + b|\psi\rangle) = a\langle\varphi|\chi\rangle + b\langle\varphi|\psi\rangle, \tag{2.11}$$

and swapping the arguments results in the complex conjugate result $\langle\psi|\varphi\rangle = \langle\varphi|\psi\rangle^*$ for $|\varphi\rangle, |\chi\rangle, |\psi\rangle \in V$ and $a, b \in \mathbb{C}$.

For any ket $|\varphi\rangle \in V$, furthermore, the inner product with itself $\langle\varphi|\varphi\rangle$ is defined to be non-negative $\langle\varphi|\varphi\rangle \geq 0$ and that it is only zero for the zero vector.

Note that $|0\rangle$ is going to be used to denote the first standard basis vector of the Hilbert space, so the zero vector is written as $|\varnothing\rangle = 0|\varphi\rangle$ here instead.

The norm induced by this inner product is written as $\||\varphi\rangle\| = \sqrt{\langle\varphi|\varphi\rangle}$.

2.2.2 Hermitian Adjoint

The Hermitian adjoint of any ket $|\varphi\rangle \in V$ is a bra $\langle\varphi| = |\varphi\rangle^\dagger$, which is defined as a linear mapping

$$\langle\varphi| \colon V \to \mathbb{C}, \quad |\psi\rangle \mapsto \langle\varphi|\psi\rangle \tag{2.12}$$

and, vice versa, the Hermitian adjoint of a bra is a ket $|\psi\rangle = \langle\psi|^\dagger$. This lets us reinterpret the symbolic bracket form of equation (2.10) as the application of a bra to a ket $\langle\varphi|\psi\rangle = \langle\varphi|(|\psi\rangle)$.

An outer product of the form $|\varphi\rangle\langle\psi|$ denotes an operator that for a ket $|\varphi\rangle \in V$ and a bra $\langle\psi| \colon W \to \mathbb{C}$ results in the linear mapping

$$|\varphi\rangle\langle\psi| \colon W \to V, \quad |\chi\rangle \mapsto \langle\psi|\chi\rangle\, |\varphi\rangle. \tag{2.13}$$

Moreover, the Hermitian adjoint $A^\dagger \colon W \to V$ of any linear mapping $A \colon V \to W$ between to Hilbert spaces V and W is defined by the equation

$$p_W(|\varphi\rangle, A|\psi\rangle) = p_V(A^\dagger|\varphi\rangle, |\psi\rangle) \tag{2.14}$$

where $|\varphi\rangle \in W$ and $|\psi\rangle \in V$.

A linear mapping $U\colon V \to W$ is *unitary* if and only if $U^\dagger U = \mathrm{id}_V$ is the identity mapping on V. Thus, U applied to two kets preserves their inner product

$$p_W(U|\varphi\rangle, U|\psi\rangle) = p_V(U^\dagger U|\varphi\rangle, |\psi\rangle) = \langle\varphi|\psi\rangle. \tag{2.15}$$

2.2.3 Tensor product

The tensor product of two Hilbert spaces V and W is itself a Hilbert space

$$V \otimes W = \left\{ \, |\varphi\rangle \otimes |\psi\rangle \;\middle|\; |\varphi\rangle \in V \wedge |\psi\rangle \in W \, \right\} \tag{2.16}$$

with the properties that

$$\Big(a|\varphi\rangle\Big) \otimes |\psi\rangle = |\varphi\rangle \otimes \Big(a|\psi\rangle\Big) = a\Big(|\varphi\rangle \otimes |\psi\rangle\Big) \tag{2.17}$$

for $a \in \mathbb{C}$, $|\varphi\rangle \in V$ and $|\psi\rangle \in W$ as well as

$$\Big(|\varphi\rangle + |\chi\rangle\Big) \otimes |\psi\rangle = |\varphi\rangle \otimes |\psi\rangle + |\chi\rangle \otimes |\psi\rangle \tag{2.18}$$

and

$$|\varphi\rangle \otimes \Big(|\psi\rangle + |\omega\rangle\Big) = |\varphi\rangle \otimes |\psi\rangle + |\varphi\rangle \otimes |\omega\rangle \tag{2.19}$$

for $|\varphi\rangle, |\chi\rangle \in V$ and $|\psi\rangle, |\omega\rangle \in W$.

Analogously, the tensor product of two linear mappings $A\colon V \to V'$ and $B\colon W \to W'$ is a linear mapping

$$A \otimes B\colon V \otimes W \to V' \otimes W', \quad |\varphi\rangle \otimes |\psi\rangle \mapsto A|\varphi\rangle \otimes B|\psi\rangle. \tag{2.20}$$

Tensor products of more than two Hilbert spaces are defined in the same manner. The n-fold tensor product of a Hilbert space V, a ket $|\varphi\rangle$ and a linear mapping A with itself is denoted by $V^{\otimes n}$, $|\varphi\rangle^{\otimes n}$ and $A^{\otimes n}$, respectively.

2.2.4 Basis

A basis is, like in the previous section, a linearly independent, spanning vector set. That is to say that for a basis B of a Hilbert space V, each ket $|\varphi\rangle \in V$ can be written in terms of complex coordinates $\varphi_b \in \mathbb{C}$ corresponding to the basis vectors $|b\rangle \in B$ as

$$|\varphi\rangle = \sum_{|b\rangle \in B} \varphi_b |b\rangle. \tag{2.21}$$

The basis vectors $|b\rangle \in B$ are linearly independent and hence the only coordinate representation of the zero vector $|\varnothing\rangle$ in this basis has all components equal to zero $\varnothing_b = 0$.

The basis vectors of an orthonormal basis are orthogonal to each other, meaning that the inner product $\langle a|b\rangle$ of two distinct ones $|a\rangle, |b\rangle \in B$ is zero, as well as normalised, such that their norms are one.

Let $\mathcal{H}(A)$ for a finite set A denote the $|A|$-dimensional Hilbert space isomorphic to $\mathbb{C}^{|A|}$ whose orthonormal standard basis vectors are labelled by the elements of A. For example, the Hilbert space $\mathcal{H}(\mathbb{B})$ is the set of vectors written as $a|0\rangle + b|1\rangle$ for $a, b \in \mathbb{C}$.

2.3 Quantum Computation and Quantum Circuits

Using the definitions and notations of Boolean vectors and Hilbert spaces, we can now establish a rigorous description of quantum states and quantum circuits.

2.3.1 Single Qubit Registers

Quantum computation operates on qubits instead of classical bits. A qubit is a unit vector of the Hilbert space $\mathcal{H}(\mathbb{B})$. Unit vectors have a norm of one unit; thus, $|\varphi\rangle = a|0\rangle + b|1\rangle$ for $a, b \in \mathbb{C}$ with $|a|^2 + |b|^2 = 1$ describes the state of a qubit where $|x + \mathrm{i}y| = \sqrt{x^2 + y^2}$ denotes the absolute value of a complex number $x + \mathrm{i}y$.

When a qubit is measured in the standard basis $\{|0\rangle, |1\rangle\}$, it can only ever be observed in one of these two basis states. Superpositions of the two states collapse to a basis state on measurement. Measuring $|\varphi\rangle$ in this basis yields $|0\rangle$ with probability $|a|^2$ and $|1\rangle$ with probability $|b|^2$.

Performing computation on a qubit is possible by applying quantum gates, which are unitary operators on the Hilbert space. Unitarity ensures the preservation of the norm.

The X gate, also known as NOT gate, is a simple quantum gate that operates on a single qubit register. The operator $X\colon \mathcal{H}(\mathbb{B}) \to \mathcal{H}(\mathbb{B})$ is defined as $X = |1\rangle\langle 0| + |0\rangle\langle 1|$ and thus swaps out the components of quantum states with respect to the standard basis. If the ket $|\varphi\rangle = a|0\rangle + b|1\rangle$ is represented by the column vector

$$\begin{bmatrix} a \\ b \end{bmatrix} \in \mathbb{C}^2, \tag{2.22}$$

then the corresponding matrix representation for X is

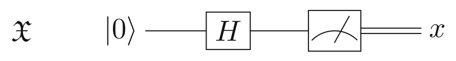
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{2.23}$$

Note that the X gate is self-adjoint, that is $X^\dagger = X$.

Another important quantum gate on a single qubit that we are going to use is the Hadamard gate $H\colon \mathcal{H}(\mathbb{B}) \to \mathcal{H}(\mathbb{B})$ defined as $H = |+\rangle\langle 0| + |-\rangle\langle 1|$ where

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad \text{and} \qquad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{2.24}$$

Figure 2.1 Simple quantum circuit for a random bit.



The matrix representation of H corresponding to the standard basis notation from equation (2.22) is

$$\frac{1}{\sqrt{2}}\begin{bmatrix}+1 & +1\\ +1 & -1\end{bmatrix}. \tag{2.25}$$

Note that the Hadamard gate is also self-adjoint. It is furthermore useful to formulate the Hadamard gate as the function

$$H\colon \mathcal{H}(\mathbb{B}) \to \mathcal{H}(\mathbb{B}), \quad |x\rangle \mapsto 2^{-1/2}\sum_{y\in\mathbb{B}}(-1)^{x\cdot y}|y\rangle. \tag{2.26}$$

The diagram of a single qubit quantum circuit is drawn as a horizontal line with labelled boxes denoting quantum gates, e. g. $\boxed{X}$ or $\boxed{H}$, and a measure symbol for receiving the measured classical value of a qubit. While a single line represents a qubit, a classical bit value is drawn as a double line.

The operations are applied in order from left to right. The simple example circuit in Figure 2.1 performs a Hadamard gate on $|0\rangle$ and measures the resulting $|+\rangle$ state, returning a random bit. The result is equally probable to be 0 or 1 as in this quantum state $|0\rangle$ and $|1\rangle$ have the same probability of $(1/\sqrt{2})^2 = 1/2$.

In order to refer to a qubit within a quantum circuit, we treat its line in the circuit as a quantum register, which we can refer to with a Fraktur letter like $\mathfrak{X}$.

The X gate is also drawn as the alternative symbol $\oplus$ directly on the wire, especially in combination with the notation for controlled gates defined in the following subsection.

2.3.2 Multiple Qubit Registers

Quantum circuits can be constructed by taking the tensor product of qubit states. When we interpret the tensor product of n qubits $\mathcal{H}(\mathbb{B})^{\otimes n}$ as an n-qubit quantum register, it is useful to summarise the tensor product states into the isomorphic Hilbert space $\mathcal{H}(\mathbb{B}^n)$ by combining the Boolean labels of the basis states into n-dimensional Boolean vectors. This isomorphism, transforming for example $|0\rangle \otimes |0\rangle \otimes |1\rangle$ into $|(0,0,1)\rangle$, is defined as

$$\bigotimes_{i=0}^{n-1}|\boldsymbol{x}[i]\rangle \mapsto |\boldsymbol{x}\rangle \tag{2.27}$$

for all $\boldsymbol{x} \in \mathbb{B}^n$.

The order of qubits from left to right in the tensor product as well as top to bottom in quantum circuit diagrams is from the least significant bit $\boldsymbol{x}[0]$ to the most significant one $\boldsymbol{x}[n-1]$, i. e. the corresponding integer to the vector $(0,0,1)$ is $0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 4$.

We can also use this isomorphism to define operations like the n-qubit Hadamard gate H_n isomorphic to $H^{\otimes n}\colon \mathcal{H}(\mathbb{B})^{\otimes n} \to \mathcal{H}(\mathbb{B})^{\otimes n}$ as

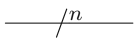
$$H_n\colon \mathcal{H}(\mathbb{B}^n) \to \mathcal{H}(\mathbb{B}^n), \quad |\boldsymbol{x}\rangle \mapsto 2^{-n/2} \sum_{\boldsymbol{y} \in \mathbb{B}^n} (-1)^{\boldsymbol{x} \odot \boldsymbol{y}} |\boldsymbol{y}\rangle \tag{2.28}$$

using the form discovered in equation (2.26). Also note that H_n is self-adjoint as for any $\boldsymbol{x} \in \mathbb{B}^n$

$$\begin{aligned} H_n \left(2^{-n/2} \sum_{\boldsymbol{y} \in \mathbb{B}^n} (-1)^{\boldsymbol{x} \odot \boldsymbol{y}} |\boldsymbol{y}\rangle \right) &= 2^{-n} \sum_{\boldsymbol{z} \in \mathbb{B}^n} \sum_{\boldsymbol{y} \in \mathbb{B}^n} (-1)^{\boldsymbol{x} \odot \boldsymbol{y}} (-1)^{\boldsymbol{y} \odot \boldsymbol{z}} |\boldsymbol{z}\rangle && (2.29) \\ &= 2^{-n} \sum_{\boldsymbol{z} \in \mathbb{B}^n} \sum_{\boldsymbol{y} \in \mathbb{B}^n} (-1)^{\boldsymbol{y} \odot (\boldsymbol{x} + \boldsymbol{z})} |\boldsymbol{z}\rangle && (2.30) \\ &= |\boldsymbol{x}\rangle && (2.31) \end{aligned}$$

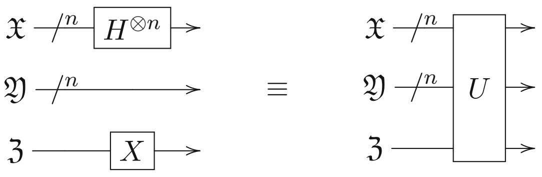
because the sum over all $\boldsymbol{y}$ cancels out to zero for all $\boldsymbol{z} \neq \boldsymbol{x}$ but evaluates to $2^n |\boldsymbol{z}\rangle$ for $\boldsymbol{z} = \boldsymbol{x}$.

Visually, an n-qubit quantum register is marked by a slash



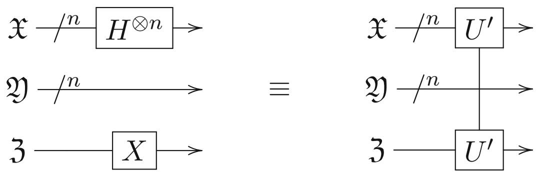
and labelled by a Fraktur letter.

Likewise, unitary operators on multiple qubit registers as well as across quantum registers are used in quantum computation as quantum gates. Gates using adjacent registers are drawn as larger labelled boxes across the affected registers. Such multi-register gates can be defined as tensor products like $U = H^{\otimes n} \otimes I^{\otimes n} \otimes X$ in



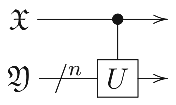
or also as non-separable gates like the controlled gates presented below.

If non-adjacent registers are used as input, multiple boxes with the same label denoting the same gate are used and connected by a vertical line. For example,



means that the gate $U' = H^{\otimes n} \otimes X$ is applied to registers $\mathfrak{X}\mathfrak{Z}$ but not to register $\mathfrak{Y}$, where the juxtaposition of register labels denotes the tensor product of the corresponding quantum states at those registers.

An important tool for using quantum registers are controlled gates. A controlled gate is a quantum gate that is conditionally applied to target registers depending on the state of control registers. Using a single qubit control register $\mathfrak{X}$ and a n-qubit register $\mathfrak{Y}$, any n-qubit quantum gate U can be conditionally applied on $\mathfrak{Y}$ controlled by $\mathfrak{X}$ as depicted by the circuit



where the small black disc denotes the control on state $|1\rangle$. Let the resulting controlled gate $\mathrm{C}_1\, U$ on $\mathfrak{X}\mathfrak{Y}$ from this circuit be defined as

$$\mathrm{C}_1\, U = |0\rangle\langle 0| \otimes \left(\sum_{\boldsymbol{x} \in \mathbb{B}^n} |\boldsymbol{x}\rangle\langle \boldsymbol{x}| \right) + |1\rangle\langle 1| \otimes \left(\sum_{\boldsymbol{x} \in \mathbb{B}^n} U|\boldsymbol{x}\rangle\langle \boldsymbol{x}| \right), \tag{2.32}$$

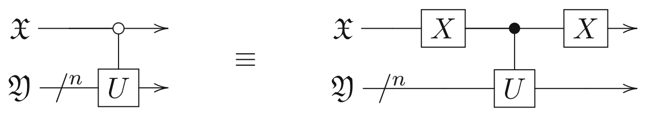
meaning that U is applied if and only if the control register is in state $|1\rangle$.

For example, the controlled NOT gate has the following effect on two qubits.

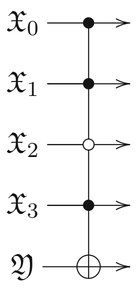
$$\mathrm{C}_1\, X \Big(|x\rangle \otimes |y\rangle \Big) = |x\rangle \otimes |x + y\rangle \tag{2.33}$$

for $x, y \in \mathbb{B}$.

Additionally, a control on state $|0\rangle$ denoted by C_0 does the opposite and applies a gate when the control register is in state $|0\rangle$ instead. This is visualised with a circle on the control register and can be realised by inverting the control register using two X gates.

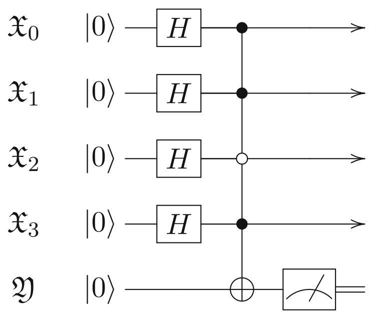


In this definition, the quantum gate U can itself be a controlled gate and hence an operation can be controlled by the value of multiple qubits. For example, let us define a multi-controlled X gate. The $\mathrm{C}_1\,\mathrm{C}_1\,\mathrm{C}_0\,\mathrm{C}_1\, X$ gate



denotes a quantum gate that inverts the value of the register $\mathfrak{Y}$ if and only if the value of the control registers is $(1, 1, 0, 1)$.

Figure 2.2 Quantum circuit with a multi-controlled X gate.



2.3.3 Measurement and Uncomputation

Nielsen and Chuang also present the mathematical definition of quantum measurement in [NC10]. A measurement is defined by a set $\mathcal{M}$ of operators $M \colon V \to V$ such that the sum of their products with their Hermitian adjoints results in the identity of the Hilbert space

$$\sum_{M \in \mathcal{M}} M^\dagger M = \mathrm{id}_V \tag{2.34}$$

and the quantum state of the circuit after measuring $M \in \mathcal{M}$ is transformed from $|\varphi\rangle$ to

$$\frac{M|\varphi\rangle}{\sqrt{\langle\varphi|M^\dagger M|\varphi\rangle}} \tag{2.35}$$

where $\langle\varphi|M^\dagger M|\varphi\rangle$ is the probability of measuring this outcome.

Measuring a single qubit register in the standard basis $\{|0\rangle, |1\rangle\}$ corresponds to the measurement set $\mathcal{M} = \{|0\rangle\langle 0|, |1\rangle\langle 1|\}$.

As a bigger example, let us now apply the multi-controlled X gate from earlier to a known quantum state. The circuit in Figure 2.2 firstly prepares the state

$$|\varphi_1\rangle = |+\rangle \otimes |+\rangle \otimes |+\rangle \otimes |+\rangle \otimes |0\rangle = \bigotimes_{i=0}^{3} |+\rangle \otimes |0\rangle \tag{2.36}$$

using four Hadamard gates. We can rewrite this state using the properties of the tensor product from equations (2.17) to (2.19) as follows.

$$|\varphi_1\rangle = \bigotimes_{i=0}^{3} \left(2^{-1/2} \sum_{x \in \mathbb{B}} |x\rangle \right) \otimes |0\rangle = 2^{-2} \sum_{\boldsymbol{x} \in \mathbb{B}^4} \left(\bigotimes_{i=0}^{3} |\boldsymbol{x}[i]\rangle \right) \otimes |0\rangle \tag{2.37}$$

Then the $\mathrm{C}_1\,\mathrm{C}_1\,\mathrm{C}_0\,\mathrm{C}_1\,X$ gate is applied and yields

$$|\varphi_2\rangle = \mathrm{C}_1\,\mathrm{C}_1\,\mathrm{C}_0\,\mathrm{C}_1\,X\,|\varphi_1\rangle \tag{2.38}$$

$$= 2^{-2}\left(\sum_{\boldsymbol{x}\in\mathbb{B}^4\setminus\{(1,1,0,1)\}}\left(\bigotimes_{i=0}^{3}|\boldsymbol{x}[i]\rangle\otimes|0\rangle\right) + |1\rangle\otimes|1\rangle\otimes|0\rangle\otimes|1\rangle\otimes|1\rangle\right), \tag{2.39}$$

that is, in the controlled state the X gate is applied to $|0\rangle$ and results in $X\,|0\rangle = |1\rangle$, while the other 15 permutations of qubits do not change the register $\mathfrak{Y}$.

Measuring the register $\mathfrak{Y}$ corresponds to the measurement operators

$$M_y = I^{\otimes 4}\otimes|y\rangle\langle y| = \sum_{\boldsymbol{x}\in\mathbb{B}^4}\left(\bigotimes_{i=0}^{3}|\boldsymbol{x}[i]\rangle\langle\boldsymbol{x}[i]|\right)\otimes|y\rangle\langle y| \tag{2.40}$$

for $y\in\mathbb{B}$ which are self-adjoint, i. e. $M_y^\dagger = M_y$, and idempotent, i. e. $M_y^2 = M_y$. This is because the fifth qubit is measured in the standard basis, while the first four qubits are left in their quantum state.

Measuring register $\mathfrak{Y}$ of the state $|\varphi_2\rangle$ yields 1 with a probability of $(1/4)^2 = 1/16$. In case the measurement was 1, the registers $\mathfrak{X}_0\mathfrak{X}_1\mathfrak{X}_2\mathfrak{X}_3$ remain in the state $|1\rangle\otimes|1\rangle\otimes|0\rangle\otimes|1\rangle$. Otherwise, if 0 was measured in $\mathfrak{Y}$, the state of the other registers after the quantum circuit is

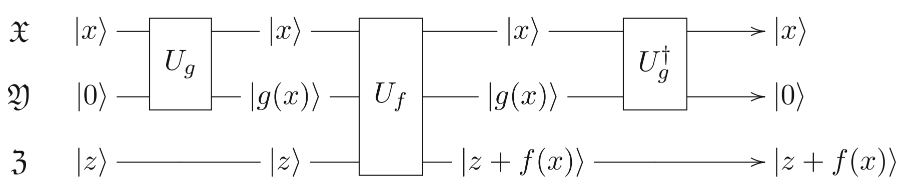
$$15^{-1/2}\sum_{\boldsymbol{x}\in\mathbb{B}^4\setminus\{(1,1,0,1)\}}\left(\bigotimes_{i=0}^{3}|\boldsymbol{x}[i]\rangle\right) \tag{2.41}$$

where the factor $15^{-1/2} = 1/\sqrt{15}$ is the result of dividing the original factor $1/4$ by the square root of the probability of the measured outcome $15/16$ as defined in equation (2.35), preserving the norm of the state vector.

At the end of each quantum circuit all registers are implicitly measured, even when their values are discarded. Thus, if intermediate values are computed whose measurement influences the quantum state of other registers, it is necessary to reverse their computation before the end of the circuit in order to not change the desired result.

This practice of reversing intermediate calculation is called *uncomputation*. Let us say as an example that we want to apply a quantum gate U_f to three registers $\mathfrak{X}\mathfrak{Y}\mathfrak{Z}$ that

Figure 2.3 Example quantum circuit showing uncomputation.



when given $\mathfrak{X}\mathfrak{Y}$ in a state of $|x\rangle \otimes |g(x)\rangle$, for some function g, adds a result of $f(x)$ to the value of register $\mathfrak{Z}$. The state $|x\rangle \otimes |g(x)\rangle$ shall be obtained via application of another quantum gate U_g to $|x\rangle \otimes |0\rangle$.

In this example depicted in Figure 2.3 the register $\mathfrak{Y}$ generally cannot be left in the state of $|g(x)\rangle$ after applying U_f, as it would interfere with the result. Therefore, register $\mathfrak{Y}$ is uncomputed by applying $U_g^\dagger$ so that register $\mathfrak{Z}$ only depends on $\mathfrak{X}$.

CHAPTER 3

Algorithm Design

Using the preliminaries from Chapter 2, we are now presenting the Bernstein–Vazirani problem and discussing different methods of recursion. Algorithms for the classical and quantum versions of the problem are given in pseudocode and the corresponding quantum circuits are visualised.

3.1 The Non-recursive Problem

The Bernstein–Vazirani problem was originally described by Ethan Bernstein and Umesh Vazirani in 1997 as *discrete Fourier sampling* problem in their paper entitled *Quantum Complexity Theory* [BV97]. Given a function f with an input size of n bits of the form

$$f \colon \mathbb{B}^n \to \mathbb{B}, \quad \boldsymbol{x} \mapsto \boldsymbol{x} \odot \boldsymbol{s} \tag{3.1}$$

using the bitwise inner product defined in equation (2.9), where $\boldsymbol{s} \in \mathbb{B}^n$ is a secret, constant n-bit vector, this hidden value $\boldsymbol{s}$ can be computed using the function f and Hadamard transformations.

While a classical, deterministic query algorithm for this value $\boldsymbol{s}$ needs to perform n invocations of the function f, a quantum circuit can measure $\boldsymbol{s}$ with certainty calling an oracle of f only once. Since the function only returns one bit of information per invocation, any classical solution needs to perform at least n calls to retrieve the secret.

An algorithm for querying $\boldsymbol{s}$ classically can use an orthonormal basis B of $\mathbb{B}^n$. The hidden string is queried as

$$\boldsymbol{s} = \sum_{\boldsymbol{b} \in B} f(\boldsymbol{b}) \cdot \boldsymbol{b}, \tag{3.2}$$

which is adding up those basis vectors $\boldsymbol{b} \in B$ whose function value $f(\boldsymbol{b})$ is one. In the simplest case the standard orthonormal basis is used and the solution is found bitwise. Algorithm 3.1 shows that function f is called n times.

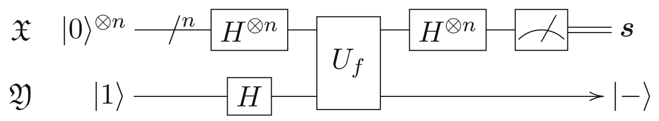
Algorithm 3.1 Classical algorithm for the non-recursive Bernstein–Vazirani problem

```
1  variable $r$ of type $\mathbb{B}^n$
2  for $i = 0 \mathinner{..} n-1$
3      variable $e := \mathbf{0}$ of type $\mathbb{B}^n$
4      $e[i] := 1$                    $e$ is now the $i$th standard basis vector
5      $r[i] := f(e)$                 set $i$th bit of $r$ to $i$th bit of $\boldsymbol{s}$ by calling $f$
6  end for
7  return $r$                         with value $\boldsymbol{s}$
```

Algorithm 3.2 Quantum algorithm for the non-recursive Bernstein–Vazirani problem

```
1  register $\mathfrak{X} := |0\rangle^{\otimes n}$
2  register $\mathfrak{Y} := |1\rangle$
3  apply $H^{\otimes n}$ to $\mathfrak{X}$       resulting in $|+\rangle^{\otimes n}$
4  apply $H$ to $\mathfrak{Y}$                resulting in $|-\rangle$
5  apply $U_f$ to $\mathfrak{X}\mathfrak{Y}$     resulting in $2^{-n/2} \sum_{\boldsymbol{x} \in \mathbb{B}^n} (-1)^{\boldsymbol{x} \odot \boldsymbol{s}} |\boldsymbol{x}\rangle \otimes |-\rangle$
6  apply $H^{\otimes n}$ to $\mathfrak{X}$       resulting in $|s\rangle$
7  discard $\mathfrak{Y}$ as $|-\rangle$         i. e. measuring $\mathfrak{Y}$ does not affect the rest of the circuit
8  measure $\mathfrak{X}$                     with value $s$
```

Figure 3.1 Quantum circuit for the non-recursive Bernstein–Vazirani problem



In order to use the classical function f in a quantum circuit, let U_f be the unitary operator

$$U_f \colon V \to V, \quad |\boldsymbol{x}\rangle \otimes |y\rangle \mapsto |\boldsymbol{x}\rangle \otimes |y + f(\boldsymbol{x})\rangle \tag{3.3}$$

on the Hilbert space $V = \mathcal{H}(\mathbb{B}^n) \otimes \mathcal{H}(\mathbb{B})$ that reversibly queries the function value. Note that this operator is self-adjoint, i. e. $U_f^\dagger = U_f$ holds, since the addition operation in the Boolean field is exclusive disjunction with the property that $f(\boldsymbol{x}) + f(\boldsymbol{x}) = 0$.

Cleve *et al.* give a quantum algorithm for the non-recursive Bernstein–Vazirani problem in their paper entitled *Quantum Algorithms Revisited* [Cle+98]. The quantum oracle U_f is applied to all input qubits in the $|+\rangle$ state and an ancilla register in the $|-\rangle$ state. This algorithm is described in Algorithm 3.2 and shown as a quantum circuit in Figure 3.1.

After preparing the quantum state

$$|\varphi_1\rangle = 2^{-n/2} \sum_{\boldsymbol{x}\in\mathbb{B}^n} |\boldsymbol{x}\rangle \otimes |-\rangle \tag{3.4}$$

the problem can be solved using one invocation of the oracle and n Hadamard gates. Calling the oracle yields the state

$$\begin{aligned} |\varphi_2\rangle &= U_f\,|\varphi_1\rangle \qquad (3.5)\\ &= 2^{-n/2} \sum_{\boldsymbol{x}\in\mathbb{B}^n} (-1)^{\boldsymbol{x}\odot\boldsymbol{s}} |\boldsymbol{x}\rangle \otimes |-\rangle \qquad (3.6) \end{aligned}$$

and the application of the remaining Hadamard gates results in

$$\begin{aligned} |\varphi_3\rangle &= H_n\,|\varphi_2\rangle \qquad (3.7)\\ &= 2^{-n} \sum_{\boldsymbol{y}\in\mathbb{B}^n} \sum_{\boldsymbol{x}\in\mathbb{B}^n} (-1)^{\boldsymbol{x}\odot\boldsymbol{s}+\boldsymbol{y}\odot\boldsymbol{x}} |\boldsymbol{y}\rangle \otimes |-\rangle \qquad (3.8)\\ &= 2^{-n} \sum_{\boldsymbol{y}\in\mathbb{B}^n} \sum_{\boldsymbol{x}\in\mathbb{B}^n} (-1)^{\boldsymbol{x}\odot(\boldsymbol{s}+\boldsymbol{y})} |\boldsymbol{y}\rangle \otimes |-\rangle \qquad (3.9)\\ &= |\boldsymbol{s}\rangle \otimes |-\rangle, \qquad (3.10) \end{aligned}$$

where measuring register $\mathfrak{X}$ reveals the secret bit vector $\boldsymbol{s}$ with certainty. As seen in equations (2.29) to (2.31), the equation from (3.9) to (3.10) holds because for $\boldsymbol{y} = \boldsymbol{s}$ the exponent evaluates to zero, and for all other values of $\boldsymbol{y}$ the exponent $\boldsymbol{x} \odot (\boldsymbol{s} + \boldsymbol{y})$ results in both Boolean values for exactly half of the values of $\boldsymbol{x}$, cancelling out those kets $|\boldsymbol{y}\rangle$.

3.2 Recursive Problem Variants

In order to prove the complexity result of Bernstein and Vazirani that quantum Turing machines are more powerful than classical ones regarding the number of oracle calls needed, researchers have come up with different ways to apply recursion to the non-recursive Bernstein–Vazirani problem.

The following subsections are focused on variants of the recursive Bernstein–Vazirani problem.

3.2.1 The Variant by Bernstein and Vazirani

The original formulation of the *recursive Fourier sampling problem* was presented in [BV97] by adding a second function $g\colon \mathbb{B}^n \to \mathbb{B}$ to the problem and redefining the goal to answer $g(\boldsymbol{s})$ instead of $\boldsymbol{s}$. The authors describe the creation of the recursive problem as follows.

> *For each problem instance of size n, we will replace the 2^n values of f with 2^n independent recursive subproblems of size $\frac{n}{2}$, and so on, stopping the recursion with function calls at the bottom.* [BV97, p. 1458]

That is to say that the problem size n is to be chosen as a power of two and the recursion depth is defined to be its binary logarithm. Let d be this depth of recursion such that $n = 2^d$.

In order to answer $g(\boldsymbol{s})$, the function $f\colon \boldsymbol{x} \mapsto \boldsymbol{x} \odot \boldsymbol{s}$ can no longer be queried directly. Instead, it is promised that for each $\boldsymbol{x} \in \mathbb{B}^n$ there is a subproblem

$$f_{\boldsymbol{x}}\colon \mathbb{B}^{n/2} \to \mathbb{B}, \quad \boldsymbol{x}_1 \mapsto \boldsymbol{x}_1 \odot \boldsymbol{s}_{\boldsymbol{x}} \tag{3.11}$$

with a secret $n/2$-bit string $\boldsymbol{s}_{\boldsymbol{x}}$ and a goal function $g_{\boldsymbol{x}}\colon \mathbb{B}^{n/2} \to \mathbb{B}$, whose solution bit reveals the function value $f(\boldsymbol{x}) = g_{\boldsymbol{x}}(\boldsymbol{s}_{\boldsymbol{x}})$.

For each $\boldsymbol{x}_1 \in \mathbb{B}^{n/2}$ there are in turn subproblems

$$f_{\boldsymbol{x},\boldsymbol{x}_1}\colon \mathbb{B}^{n/4} \to \mathbb{B}, \quad \boldsymbol{x}_2 \mapsto \boldsymbol{x}_2 \odot \boldsymbol{s}_{\boldsymbol{x},\boldsymbol{x}_1} \tag{3.12}$$

with $n/4$-bit secrets $\boldsymbol{s}_{\boldsymbol{x},\boldsymbol{x}_1}$ and goal functions $g_{\boldsymbol{x},\boldsymbol{x}_1}\colon \mathbb{B}^{n/4} \to \mathbb{B}$ promising that $f_{\boldsymbol{x}}(\boldsymbol{x}_1) = g_{\boldsymbol{x},\boldsymbol{x}_1}(\boldsymbol{s}_{\boldsymbol{x},\boldsymbol{x}_1})$, and so forth, until reaching the functions

$$f_{\boldsymbol{x},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{d-1}}\colon \mathbb{B} \to \mathbb{B}, \quad \boldsymbol{x}_d \mapsto \boldsymbol{x}_d \odot \boldsymbol{s}_{\boldsymbol{x},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{d-1}}, \tag{3.13}$$

with 1-bit secrets $\boldsymbol{s}_{\boldsymbol{x},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{d-1}}$. These problem functions $f_{\boldsymbol{x},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{d-1}}$ are given as oracle functions.

Reinterpreting the index parameters as separate function inputs and thus lifting the constants $\boldsymbol{s}$ to functions, the domain of these functions grows like $\mathbb{B}^n \times \mathbb{B}^{n/2} \times \mathbb{B}^{n/4} \times \cdots \times \mathbb{B}^{n/2^k}$ and the definitions can be rewritten as

$$f_k\colon \prod_{i=0}^{k} \mathbb{B}^{2^{d-i}} \to \mathbb{B}, \quad (\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k) \mapsto \boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1}) \tag{3.14}$$

$$\boldsymbol{s}_k\colon \prod_{i=0}^{k-1} \mathbb{B}^{2^{d-i}} \to \mathbb{B}^{2^{d-k}} \tag{3.15}$$

$$g_k\colon \prod_{i=0}^{k} \mathbb{B}^{2^{d-i}} \to \mathbb{B} \tag{3.16}$$

for k from 0 to d with the condition that

$$f_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k) = g_{k+1}\Big(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k, \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\Big) \tag{3.17}$$

for k from 0 to $d-1$ with $\boldsymbol{x}_i \in \mathbb{B}^{2^{d-i}}$.

For $k = 0$ we let the argument to $\boldsymbol{s}_k$ in equation (3.14) and in the following occurrences of the term $\boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})$ denote the empty tuple $()$ and let the empty Cartesian product in equation (3.15) be the singleton set of the empty tuple. $\boldsymbol{s}_0\colon \{()\} \to \mathbb{B}^n$ can then be identified with a constant $\boldsymbol{s}_0 \in \mathbb{B}^n$.

The problem solution may access the oracle function f_d as well as all goal functions g_k in order to answer the goal $g_0(\boldsymbol{s}_0)$.

3.2.2 The Variant by Aaronson

In the paper entitled *Quantum Lower Bound for Recursive Fourier Sampling* [Aar03] Aaronson removes the necessity to half the problem size in each recursion level. He shows that the parameter of recursion depth d can be chosen independently of the input size. Hence, the problem now has two parameters: n and d.

This simplifies the problem's definitions to

$$f_k\colon (\mathbb{B}^n)^{k+1} \to \mathbb{B}, \quad (\boldsymbol{x}_0, \dots, \boldsymbol{x}_k) \mapsto \boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0, \dots, \boldsymbol{x}_{k-1}) \tag{3.18}$$

$$\boldsymbol{s}_k\colon (\mathbb{B}^n)^k \to \mathbb{B}^n \tag{3.19}$$

$$g_k\colon (\mathbb{B}^n)^{k+1} \to \mathbb{B} \tag{3.20}$$

for k from 0 to d with the condition that

$$f_k(\boldsymbol{x}_0, \dots, \boldsymbol{x}_k) = g_{k+1}\Big(\boldsymbol{x}_0, \dots, \boldsymbol{x}_k, \boldsymbol{s}_{k+1}(\boldsymbol{x}_0, \dots, \boldsymbol{x}_k)\Big) \tag{3.21}$$

for k from 0 to $d-1$.

Note that now all parameters $\boldsymbol{x}_0, \dots, \boldsymbol{x}_d$ and all values of $\boldsymbol{s}_k$ are n-bit vectors in $\mathbb{B}^n$. The rule about empty tuples for $k = 0$ from the previous subsection still holds.

Having now all subproblems share the same dimensions, Aaronson suggests that the goal functions' dependencies on $\boldsymbol{x}_0, \dots, \boldsymbol{x}_{k-1}$ can be removed and the resulting function from $\mathbb{B}^n$ to $\mathbb{B}$ can be unified into one function g such that

$$f_k(\boldsymbol{x}_0, \dots, \boldsymbol{x}_k) = g\Big(\boldsymbol{s}_{k+1}(\boldsymbol{x}_0, \dots, \boldsymbol{x}_k)\Big) \tag{3.22}$$

for k from 0 to $d-1$.

For this function he shows that

$$g_{\bmod 3}\colon \mathbb{B}^n \to \mathbb{B}, \quad \boldsymbol{x} \mapsto \begin{cases} 0 & \|\boldsymbol{x}\|_1 \equiv 0 \mod 3 \\ 1 & \text{otherwise,} \end{cases} \tag{3.23}$$

where $\|\boldsymbol{x}\|_1$ denotes the Hamming weight of the bit vector, can be used throughout the recursive problem as an efficiently computable function that is complicated enough to not simplify the problem complexity.

Again only one oracle f_d and the goal function g are directly accessible to the solution algorithm.

3.2.3 The Variant by Johnson

Johnson combines Bernstein's and Vazirani's variant with Aaronson's variant in his dissertation entitled *Upper and Lower Bounds for Recursive Fourier Sampling* [Joh08]. The recursive structure by Aaronson from equations (3.18) to (3.21) is used, while the functions g_k are not unified like in Aaronson's formulation, but are kept separate functions.

By choosing this problem definition, the functions g_k still have access to the parameters $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1}$. This allows for more possibilities in selecting the hidden bit strings $\boldsymbol{s}_k$ as they do not have to satisfy equation (3.22).

3.2.4 A New Variant for Implementations

The aforementioned variants cannot be implemented easily for the secret bit vectors $\boldsymbol{s}_k$ and the goal functions g_k have to fulfil the recursively promised conditions (3.17) and (3.21). The authors of the respective papers provide no straightforward way of constructing such problem instances.

Since the introduction of the goal functions g_k is not necessary in order to perform recursion on the Bernstein–Vazirani problem, the following definition does not use them. Instead the problem functions f_k on all but the last level of recursion receive an additional control argument $\boldsymbol{a} \in \mathbb{B}^n$ that ensures having solved the subproblems.

The depth d is again the number of recursion levels, independent of the problem size n. For k from 0 to $d-1$, let the oracle functions be

$$f_k \colon (\mathbb{B}^n)^{k+2} \to \mathbb{B}, \quad (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k, \boldsymbol{a}) \mapsto \begin{cases} \boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1}) & \boldsymbol{a} = \boldsymbol{s}_{k+1}(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k) \\ 0 & \text{otherwise,} \end{cases} \tag{3.24}$$

and on the last level

$$f_d \colon (\mathbb{B}^n)^{d+1} \to \mathbb{B}, \quad (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_d) \mapsto \boldsymbol{x}_d \odot \boldsymbol{s}_d(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{d-1}) \tag{3.25}$$

with independently choosable hidden bit strings

$$\boldsymbol{s}_k \colon (\mathbb{B}^n)^k \to \mathbb{B}^n \tag{3.26}$$

for all k from 0 to d. All these problem functions f_k including f_d are available as oracles to solution algorithms. The goal is to retrieve the hidden n-bit vector $\boldsymbol{s}_0$.

In this variant the oracle functions f_k replace the role of the goal functions g_k as the means to query information about the secrets. This change in oracle accessibility does not impact the problem complexity because of the added control parameter $\boldsymbol{a}$. The functions f_k now have the same signature as g_{k+1} in equation (3.21).

As the intended way of solving this problem shall always provide $\boldsymbol{a} = \boldsymbol{s}_{k+1}(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k)$ in equation (3.24), the *otherwise* branch need not be defined at all if one allows for partial problem functions f_k. Alternatively, the constant 0 can be changed to any expression not revealing $\boldsymbol{s}_k$.

A classical algorithm for the recursive Bernstein–Vazirani problem can again use vectors of the standard orthonormal basis to query the secret bit vectors, in the same way as in

Algorithm 3.3 Classical algorithm for the recursive Bernstein–Vazirani problem

```
function $RBV_k(e_0\colon \mathbb{B}^n, \ldots, e_{k-1}\colon \mathbb{B}^n)\colon \mathbb{B}^n$ is
                                        $k$ parameters of type $\mathbb{B}^n$, for $k$ from 0 to $d$
    variable $r$ of type $\mathbb{B}^n$
    for $i = 0 \mathinner{..} n-1$
        variable $e_k \coloneqq \mathbf{0}$ of type $\mathbb{B}^n$
        $e_k[i] \coloneqq 1$                        $e_k$ is now the $i$th standard basis vector
        if $k = d$ then
            $r[i] \coloneqq f_d(e_0, \ldots, e_d)$
        else
            $r[i] \coloneqq f_k\big(e_0, \ldots, e_k, RBV_{k+1}(e_0, \ldots, e_k)\big)$
        end if
    end for
    return $r$                                with value $\boldsymbol{s}_k(\boldsymbol{e}_0, \ldots, \boldsymbol{e}_{k-1})$
end function
return $RBV_0()$                              this function call yields the solution value $\boldsymbol{s}_0$
```

Algorithm 3.1. In Algorithm 3.3 the inputs $\boldsymbol{x}_k$ are set to standard basis vectors that are stored in the variables e_k and the hidden strings $\boldsymbol{s}_k$ are then computed bitwise stored in variables r.

Correctness of this classical algorithm can be shown as follows. In the base case for $k = d$ the function $RBV_d(e_0, \ldots, e_{d-1})$ computes

$$\boldsymbol{s}_d(e_0, \ldots, e_{d-1}) = \sum_{\boldsymbol{b} \in B} f_d(e_0, \ldots, e_{d-1}, \boldsymbol{b}) \cdot \boldsymbol{b} \tag{3.27}$$

bitwise by choosing the standard basis for B the same way as in the non-recursive case.

In the induction step we show that for k from $d-1$ to 0, given the induction hypothesis that $RBV_{k+1}(e_0, \ldots, e_k)$ returns $\boldsymbol{s}_{k+1}(e_0, \ldots, e_k)$, the function $RBV_k(e_0, \ldots, e_{k-1})$ shall return

$$\boldsymbol{s}_k(e_0, \ldots, e_{k-1}) = \sum_{\boldsymbol{b} \in B} f_k\Big(e_0, \ldots, e_{k-1}, \boldsymbol{b}, RBV_{k+1}(e_0, \ldots, e_k)\Big) \cdot \boldsymbol{b} \tag{3.28}$$

where B is again the standard basis of $\mathbb{B}^n$.

Thus, $RBV_0()$ returns $\boldsymbol{s}_0$ after d induction steps.

As each invocation of RBV_k iterates over n values of i calling f_k once per iteration and recursing each time for $k < d$, this algorithm calls the problem functions f_k exactly n^{k+1}

times for k from 0 to d; thus,

$$\sum_{k=0}^{d} n^{k+1} = \frac{n}{n-1} \cdot (n^{d+1} - 1) \tag{3.29}$$

oracle calls are made in total.

A quantum algorithm for solving this recursive variant is Algorithm 3.4, which is also shown in Figure 3.2 as a quantum circuit diagram. This algorithm defines n-qubit registers $\mathfrak{X}_k$ for k from 0 to d and one ancilla register $\mathfrak{Y}$. The initial state

$$|\varphi_1\rangle = 2^{-(d+1)n/2} \bigotimes_{l=0}^{d} \left(\sum_{\boldsymbol{x}_l \in \mathbb{B}^n} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle \tag{3.30}$$

is prepared using Hadamard transformations similar to the non-recursive version.

Then the order of oracle function invocations and Hadamard gates follows a recursive definition. The structure was formalised by McKague in his paper entitled *Interactive proofs with efficient quantum prover for recursive Fourier sampling* [McK12, p. 4, Algorithm 2].

The procedure *RBV* and its corresponding recursively defined quantum gate can be described as follows. In the base case for $k = d$ the procedure $RBV(d)$ applies the gate U_{f_d} to all registers, which performs the computation

$$|\varphi_2\rangle = U_{f_d} |\varphi_1\rangle \tag{3.31}$$

$$= 2^{-(d+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_d \in \mathbb{B}^n} \left((-1)^{\boldsymbol{x}_d \odot \boldsymbol{s}_d(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{d-1})} \bigotimes_{l=0}^{d} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle \tag{3.32}$$

using the tensor product's properties from equations (2.17) to (2.19), effectively introducing a factor of $(-1)^{\boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})}$. For k from $d-1$ to 0 we can now compute the control argument by calling $RBV(k+1)$, assuming that it adds $(-1)^{\boldsymbol{x}_{k+1} \odot \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)}$ as a factor

$$RBV(k+1)\, 2^{-(d+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_d \in \mathbb{B}^n} \left(\bigotimes_{l=0}^{d} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle = \tag{3.33}$$

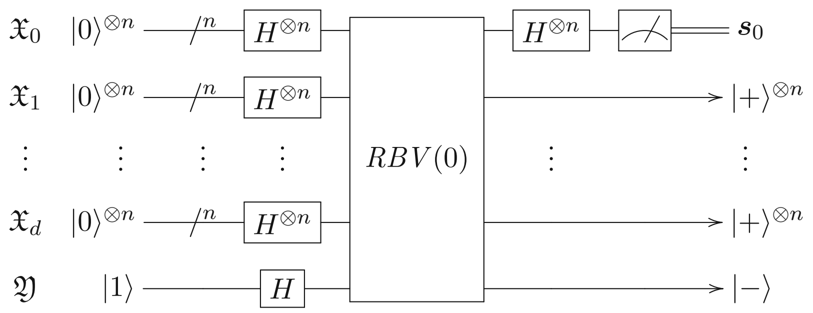
$$2^{-(d+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_d \in \mathbb{B}^n} \left((-1)^{\boldsymbol{x}_{k+1} \odot \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)} \bigotimes_{l=0}^{d} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle$$

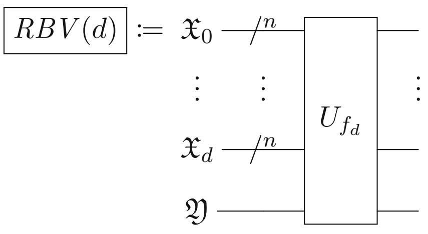
and applying Hadamard gates to the register $\mathfrak{X}_{k+1}$ after the $RBV(k+1)$ gate, performing

$$H_n\, 2^{-n/2} \sum_{\boldsymbol{x}_{k+1} \in \mathbb{B}^n} (-1)^{\boldsymbol{x}_{k+1} \odot \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)} |\boldsymbol{x}_{k+1}\rangle = |\boldsymbol{s}_{k+1}(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k)\rangle \tag{3.34}$$

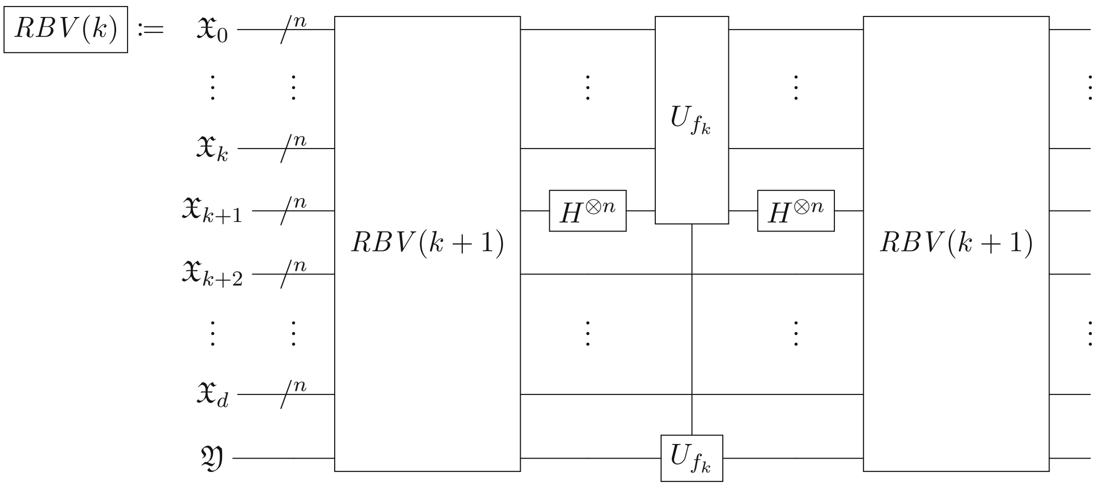
applying the pattern described in equations (2.29) through (2.31).

Figure 3.2 Quantum circuit for the recursive Bernstein–Vazirani problem





for k from 0 to $d-1$:



Algorithm 3.4 Quantum algorithm for the recursive Bernstein–Vazirani problem

```
for $k = 0 \mathinner{..} d$
    register $\mathfrak{X}_k := |0\rangle^{\otimes n}$
    apply $H^{\otimes n}$ to $\mathfrak{X}_k$                resulting in $|+\rangle^{\otimes n}$
end for
register $\mathfrak{Y} := |1\rangle$
apply $H$ to $\mathfrak{Y}$                              resulting in $|-\rangle$
procedure $RBV(k\colon \mathbb{N})$ is                   shift phase by $(-1)^{\boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})}$
    if $k = d$ then
        apply $U_{f_d}$ to $\mathfrak{X}_0 \cdots \mathfrak{X}_d \mathfrak{Y}$
    else
        call $RBV(k+1)$
        apply $H^{\otimes n}$ to $\mathfrak{X}_{k+1}$        resulting in $|\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle$
        apply $U_{f_k}$ to $\mathfrak{X}_0 \cdots \mathfrak{X}_{k+1} \mathfrak{Y}$
        apply $H^{\otimes n}$ to $\mathfrak{X}_{k+1}$        uncomputing $|\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle$
        call $RBV(k+1)$
    end if
end procedure
call $RBV(0)$
for $k = 1 \mathinner{..} d$
    discard $\mathfrak{X}_k$ as $|+\rangle^{\otimes n}$
end for
discard $\mathfrak{Y}$ as $|-\rangle$
apply $H^{\otimes n}$ to $\mathfrak{X}_0$                resulting in $|\boldsymbol{s}_0\rangle$
measure $\mathfrak{X}_0$                                 with value $\boldsymbol{s}_0$
```

Then U_{f_k} can be applied to registers $\mathfrak{X}_0 \cdots \mathfrak{X}_{k+1}\mathfrak{Y}$ as expected to yield

$$U_{f_k}\, 2^{-(k+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k \in \mathbb{B}^n} \left(\bigotimes_{l=0}^{k} |\boldsymbol{x}_l\rangle \otimes |\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle \right) \otimes |-\rangle = \tag{3.35}$$
$$2^{-(k+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k \in \mathbb{B}^n} \left((-1)^{\boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})} \bigotimes_{l=0}^{k} |\boldsymbol{x}_l\rangle \otimes |\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle \right) \otimes |-\rangle,$$

which computes the factor $(-1)^{\boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})}$. However, the control argument cannot be left in the state $|\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle$ because it depends on other registers that we do not want to measure here. Therefore, register $\mathfrak{X}_{k+1}$ is uncomputed by applying Hadamard gates as

$$H_n\, |\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)\rangle = 2^{-n/2} \sum_{\boldsymbol{x}_{k+1} \in \mathbb{B}^n} (-1)^{\boldsymbol{x}_{k+1} \odot \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)} |\boldsymbol{x}_{k+1}\rangle \tag{3.36}$$

and another procedure call $RBV(k+1)$ to cancel out the factor

$$RBV(k+1)\, 2^{-(d+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_d \in \mathbb{B}^n} \left((-1)^{\boldsymbol{x}_{k+1} \odot \boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_k)} \bigotimes_{l=0}^{d} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle = \tag{3.37}$$

$$2^{-(d+1)n/2} \sum_{\boldsymbol{x}_0,\ldots,\boldsymbol{x}_d \in \mathbb{B}^n} \left(\bigotimes_{l=0}^{d} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle$$

using the fact that the RBV gates are self-adjoint, since adding a second copy of the factor multiplies to one. So overall $RBV(k)$ adds a factor of $(-1)^{\boldsymbol{x}_k \odot \boldsymbol{s}_k(\boldsymbol{x}_0,\ldots,\boldsymbol{x}_{k-1})}$ to the quantum state.

Hence, the call of $RBV(0)$ computes a state of

$$2^{-n/2} \sum_{\boldsymbol{x}_0 \in \mathbb{B}^n} (-1)^{\boldsymbol{x}_0 \odot \boldsymbol{s}_0} |\boldsymbol{x}_0\rangle \otimes \bigotimes_{l=1}^{d} \left(\sum_{\boldsymbol{x}_l \in \mathbb{B}^n} |\boldsymbol{x}_l\rangle \right) \otimes |-\rangle, \tag{3.38}$$

where the solution $\boldsymbol{s}_0$ can be obtained via another Hadamard transformation on the $\mathfrak{X}_0$ register.

This quantum algorithm applies the U_{f_k} gate 2^k times; thus, it uses

$$\sum_{k=0}^{d} 2^k = 2^{d+1} - 1 \tag{3.39}$$

invocations of oracle functions in total, independent of the problem size n. Recall that the number of oracle calls of the classical solution in equation (3.29) did in fact depend on n.

The significance of this result regarding computational complexity is discussed in Chapter 5.

CHAPTER 4

Implementation

In this chapter the Silq and Qiskit implementations of the Bernstein–Vazirani algorithm in both non-recursive and recursive forms are presented and discussed.

Silq is a quantum programming language that is based on a strong type system. It is used to verify that all used intermediate values are correctly uncomputed and do not interfere with the intended measurement. The Silq implementation emulates quantum computation on classical computers in a type-safe manner.

As presented in the publication [Bic+20], Silq supports types for interpreting qubits or classical bits as fixed-width integers of arbitrary width as well as tuples, arrays and vectors.

Qiskit [Qis22] is a software development kit for Python that provides a way to define quantum circuits, compile them into quantum assembly instructions and run them on an emulator or on quantum hardware. Furthermore, quantum circuit diagrams can be generated from the circuit implementations.

4.1 Concepts and Features of Silq

Information about the features of the programming language Silq is taken from the paper [Bic+20] as well as Silq's website [Bic+23]. While the paper does not explain all the features of Silq that are needed in order to use this language, the website provides us with more usage examples.

All classical types are marked with an exclamation mark `!` in front of the data type. The data types we are going to use are the following.

Boolean values are of the type `B` for qubits and `!B` for classical bits, with the operator `xorb`, short for *XOR binary*, performing Boolean addition. Classical natural numbers denoted by `!N` can be arbitrarily large non-negative integers.

Tuples can be defined by parenthesised expressions like `(0, 3, 1) : !B x !N x !N` or built by concatenation, e. g. `(0,) ~ (3, 1)` where the tuple with one element is distinguished by a trailing comma. A vector of type `T^n` for some `n : !N` is a fixed-length `n`-tuple of values of the same type `T : *` where the asterisk denotes the kind of any data type.

A sequence of elements of a type `T : *` with a variable number of elements is an array `T[]`, whose instances are constructed by the use of brackets, e. g. `[0] ~ [3, 1] : !N[]` concatenates two lists of naturals resulting in a 3-element array.

Vectors of `n : !N` bits or qubits can also be viewed as fixed-size integers. We are going to use unsigned integers of the type `uint[n]` which treat a vector $\boldsymbol{x} \in \mathbb{B}^n$ as its corresponding binary value

$$\sum_{i=0}^{n-1} \boldsymbol{x}[i] \cdot 2^i \tag{4.1}$$

with the least significant bit at the lowest index.

Functions can have special annotations related to quantum computation. Quantum type parameters which are not marked `const` have to be consumed in order to prevent implicit measurement, that is to say that their state has to be used up or uncomputed manually. On the other hand, `const` parameters are uncomputed automatically.

A function is `qfree` if and only if it neither introduces nor destroys superpositions, i. e. it can be described classically. Finally, the annotation `lifted` after the list of parameters in a function definition is a shorthand to annotate all parameters as `const` and mark the function itself as `qfree`.

Generic parameters in brackets like `[n : !N]` can be used to define parametrically polymorphic functions by using type variables in their definitions. These generic parameters can be omitted in function calls if they are redundant with the type information.

There is currently no way to export the quantum circuit as an interchange format nor to run the program on a quantum computer. The main accomplishment of Silq is to type-check quantum programs and to emulate them on classical hardware returning quantum states in Dirac notation.

4.2 Algorithm Modelling in Silq

The following Silq scripts are going to reference utility definitions from `util.slq` for n-qubit Hadamard gates as well as array and vector builder functions. Listing 4.1 provides the source of these reusable definitions.

The function `Hn` applies the Hadamard transformation to all components of its `B^n` argument. An array of values is returned by `arrOf` given the desired length and a function that determines the value for each index. Arrays can be coerced into vectors

Listing 4.1 Silq utility functions `util.slq` used in the algorithm implementations

```
// apply H to n qubits
def Hn [n : !N] (x : B^n) : B^n
{
  for i in [0 .. n)
  {
    x[i] := H (x[i]);
  }
  return x;
}

// array of supplied classical values
def arrOf [T : *] (n : !N, f : !N !-> !T) : !T[]
{
  v := [] : !T[];
  for i in [0 .. n)
  {
    v ~= [f (i)];
  }
  return v;
}

// vector of supplied classical values
def vecOf [T : *] (n : !N, f : !N !-> !T) : !T^n
{
  return arrOf (n, f) coerce !T^n;
}

// random classical n-bit vector
def randomVec [n : !N] () : !B^n
{
  return vecOf (n, lambda (i : !N) => measure (H (0 : B)));
}
```

of the same length as done by the function `vecOf`. An easy way to obtain random bit vectors in Silq is implemented as `randomVec`.

The algorithms are implemented in Silq in order to test them on actual values for the secret bit vectors. The following programs select random secrets and then run the Bernstein–Vazirani algorithm.

Firstly, the non-recursive Bernstein–Vazirani algorithm is implemented. The bitwise inner product $f(\boldsymbol{x}) = \boldsymbol{x} \odot \boldsymbol{s}$ is modelled exactly like in its definition in equation (2.9) as the sum of bitwise logical conjunctions. This lets us define the function `f` that computes this dot product, which is then used to define the function `main` resembling the pseudocode from Algorithm 3.2. The source code is available in Listing 4.2.

The problem size parameter n can be adapted easily at line 14.

This implementation of the non-recursive Bernstein–Vazirani algorithm chooses a random secret string `s` each time. A possible output from running this script is

```
(1,1,0,0,1)
(1+0i)·|(1,1,0,0,1)⟩
```

where the first line is the value of `s` that was chosen and the second line represents the quantum state of the register `x` returned after the computation, which shows the expected result.

This program can now be extended towards the recursive Bernstein–Vazirani algorithm as presented in Algorithm 3.4. In the most general case, we use a lookup table of random values to store the information for the secrets $\boldsymbol{s}_k(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1})$ from equation (3.26) used to define the oracle functions of the recursive Bernstein–Vazirani problem. For each k from 0 to d there are 2^{kn} secret values. These correspond to all possible argument values $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1}$ for $k > 0$.

The algorithm implementation is given in Listing 4.3. Its output is of the same form as the non-recursive algorithm.

This implementation defines a vector of arrays with the random secret values at lines 79 to 81. The value of `sktab[`k`][`X`]` here denotes the secret $\boldsymbol{s}_k(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1})$ where X is the unsigned integer representation of the concatenated argument vectors $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1}$. This index

$$X = \sum_{i=0}^{k-1} \sum_{j=0}^{n-1} \boldsymbol{x}_i[j] \cdot 2^{in+j} \tag{4.2}$$

is computed by the function `concat`, which builds a kn-bit vector from k separate n-bit vectors and casts it to a kn-bit unsigned integer.

Having this integer X determined by the state of quantum registers, the array is accessed by comparing this index to classical values in function `sk`. For the first index `i` such that its integer representation compares equal to the concatenated quantum state of `x`, the

Listing 4.2 Silq implementation of the non-recursive Bernstein–Vazirani algorithm

```
import util; // defines randomVec, Hn

// f(x) = x . s
def f [n : !N] (s : !B^n) (x : B^n) lifted : B
{
  r := 0 : B;
  for i in [0..n)
  {
    r xorb= x[i] & s[i];
  }
  return r;
}

n := 5; // problem size parameter

def main ()
{
  s := randomVec [n] ();
  print (s);

  x := vector (n, 0 : B);
  x := Hn (x);
  y := H (1 : B);

  y xorb= f (s) (x);

  x := Hn (x);
  y := H (y);
  forget (y = 1);

  assert (measure (x == s));
  return x;
}
```

Listing 4.3 Silq implementation of the recursive Bernstein–Vazirani algorithm

```
import util; // defines randomVec, arrOf, vecOf, Hn

// concat x0 .. x[k-1] as uint[k*n]
def concat [n : !N, k : !N] (x : (B^n)^k) lifted : uint[k*n]
{
  arr := [] : B[];
  for i in [0 .. k)
  {
    arr ~= x[i]; // concatenate all qubits into an array
  }
  return (arr coerce B^(k*n)) as uint[k*n];
}

// sk(x0,..,x[k-1]) = skarr[concat (x0 .. x[k-1])]
//   where skarr = sktab[k]
def sk [n : !N, k : !N] (skarr : !(B^n)[])
  (x : (B^n)^k) lifted : B^n
{
  for i in [0 .. 2^(k*n))
  {
    if (i as !uint[k*n]) == concat (x)
    {
      return skarr[i] : B^n;
    }
  }
  assert (false);
}

// fd(x0,..,xd) = xd . sd(x0,..,x[d-1])
def fd [n : !N, d : !N] (sktab : !(B^n)[]^(d+1))
  (x : (B^n)^d, xd : B^n) lifted : B
{
  s := sk [n, d] (sktab[d]) (x);
  r := 0 : B;
  for i in [0 .. n)
  {
    r xorb= xd[i] & s[i];
  }
  return r;
}
```

```

// fk(x0,..,xk,a) = xk . sk(x0,..,x[k-1]) [ a == s[k+1](x0,..,xk) ]
def fk [n : !N, k : !N, d : !N] (sktab : !(B^n)[]^(d+1))
  (x : (B^n)^k, xk : B^n, a : B^n) lifted : B
{
  s := sk [n, k] (sktab[k]) (x);
  r := 0 : B;
  if a == sk [n, k+1] (sktab[k+1]) (x ~ (xk,))
  {
    for i in [0 .. n)
    {
      r xorb= xk[i] & s[i];
    }
  }
  return r;
}

def rbv [n : !N, d : !N] (sktab : !(B^n)[]^(d+1))
  (xk : (B^n)^(d+1), y : B, k : !N) : (B^n)^(d+1) x B
{
  if k == d
  {
    y xorb= fd (sktab) (xk[0 .. d], xk[d]);
    return (xk, y);
  }
  (xk, y) := rbv (sktab) (xk, y, k + 1);
  xk[k+1] := Hn (xk[k+1]);
  y xorb= fk [n, k, d] (sktab) (xk[0 .. k], xk[k], xk[k+1]);
  xk[k+1] := Hn (xk[k+1]);
  (xk, y) := rbv (sktab) (xk, y, k + 1);
  return (xk, y);
}
```

```

n := 2; // problem size parameter
d := 2; // recursion depth parameter

def main ()
{
  genSecrets := lambda (i : !N) => randomVec [n] ();
  genArrays  := lambda (k : !N) => arrOf (2^(k*n), genSecrets);
  sktab := vecOf (d + 1, genArrays) : !(B^n)[]^(d+1);
  print (sktab[0][0]);

  xk := vector (d + 1, vector (n, 0 : B));
  for k in [0 .. d+1)
  {
    xk[k] := Hn (xk[k]);
  }
  y := H (1 : B);

  (xk, y) := rbv (sktab) (xk, y, 0);

  (x0,) ~ xk := xk;
  x0 := Hn (x0);

  for k in [0 .. d)
  {
    xk[k] := Hn (xk[k]);
  }
  forget (xk = vector (d, vector (n, 0)));
  y := H (y);
  forget (y = 1);

  assert (measure (x0 == sktab[0][0]));
  return x0;
}
```

lookup table entry is returned. It is not possible for this search to fail because the whole index space from 0 to $2^{kn} - 1$ is covered.

While the function `fd` is written similar to function `f` in the non-recursive implementation, `fk` has to additionally check for the control argument $\boldsymbol{a}$ as specified in the defining equation (3.24). If the parameter `a` were not equal to the corresponding value in the lookup table, a state of $|0\rangle$ would be returned.

Note that `(xk,)` denotes the singleton tuple of `xk` with the trailing comma distinguishing it from other parenthesised expressions. In Silq the concatenation operator `~` applies to both arrays and vectors.

Finally, the recursive procedure from Algorithm 3.4 is implemented as the function `rbv`. Note that the index slice expression `xk[0 .. k]` yields the vector at the indices from 0 to $k - 1$.

The parameters n and d can be changed at lines 74 and 75, respectively, but the defaults are low for performance reasons.

4.3 Program Implementation in Qiskit

While Silq is a useful programming language to evaluate these algorithms, it does not provide a possibility for assembling quantum circuits or exporting the program into other formats. We use Qiskit to accomplish this task and show how the algorithm defines an implementable quantum circuit that can in principle be verified on quantum computers.

Hence, the non-recursive and recursive Bernstein–Vazirani algorithms are implemented in Qiskit and emulated on a classical computer, while having the option to directly compile the scripts into machine code for quantum hardware.

Again the non-recursive algorithm is implemented first. We chose to define the U_f gate as a compound quantum gate as constructed at lines 18 to 22 in Listing 4.4. This is achieved defining a sub-circuit `Uf_qc` that is then translated into a quantum gate using Qiskit's `to_gate` method. With this definition the program outputs diagram images of the quantum circuit with different levels of abstraction.

Before and after each oracle invocation barriers are put in place to prevent the transpiler from using the oracle implementation instance when optimising the circuit. These barriers are not shown in the diagrams, though, to avoid clutter.

Running the program prints two lines consisting of the random secret $\boldsymbol{s}$ and the emulation results, for example

```
s = 10110
{'10110': 1024}
```

showing that all 1024 shots, i. e. repetitions of the emulation, resulted in the correct value measured. Besides the textual output, the generated images serve as an overview of the

Listing 4.4 Qiskit implementation of the non-recursive Bernstein–Vazirani algorithm

```python
from qiskit import *
from random import getrandbits

n = 5
fmt = dict(output='mpl', plot_barriers=False)

def indexed(x):
    return enumerate([b == '1' for b in reversed(f'{x:0{n}b}')])

s = getrandbits(n)
print(f's␣=␣{s:0{n}b}')
x = QuantumRegister(n, 'x')
y = AncillaRegister(1, 'y')
c = ClassicalRegister(n, 'c')
qc = QuantumCircuit(x, y, c)

# f(x) = x . s
Uf_qc = QuantumCircuit(x, y)
for i, bit in indexed(s):
    if bit:
        Uf_qc.cx(x[i], y)
Uf = Uf_qc.to_gate(label='$U_f$')

qc.h(x)
qc.x(y)
qc.h(y)
qc.barrier()

qc.append(Uf, x[:] + y[:])
qc.barrier()

qc.h(x)
qc.measure(x, c)
qc.draw(**fmt, filename='1bv-circuit.png')
qc.decompose('*U*').draw(**fmt, filename='1bv-decomposed.png')

sim = Aer.get_backend('aer_simulator')
tp = transpile(qc, sim)
tp.draw(**fmt, filename='1bv-transpiled.png')
print(sim.run(tp, shots=1024).result().get_counts())
```

circuit `1bv-circuit.png`, a decomposition of the oracle gate U_f in `1bv-decomposed.png` as well as the transpiled version `1bv-transpiled.png` of how the emulator receives the circuit. Examples of these images are shown in Figures 4.1 to 4.3 where Qiskit also produces a $U_2(-\pi,-\pi)$ gate. This gate $U_2(\varphi,\lambda)\colon \mathcal{H}(\mathbb{B}) \to \mathcal{H}(\mathbb{B})$ for $\varphi,\lambda \in \mathbb{C}$ is defined as

$$U_2(\varphi,\lambda) = 2^{-1/2}\Big(|0\rangle\langle 0| + \mathrm{e}^{\mathrm{i}\varphi}|1\rangle\langle 0| - \mathrm{e}^{\mathrm{i}\lambda}|0\rangle\langle 1| + \mathrm{e}^{\mathrm{i}(\varphi+\lambda)}|1\rangle\langle 1|\Big) \tag{4.3}$$

and used to encode the composition of H after X as $HX = U_2(-\pi,-\pi) = |-\rangle\langle 0| + |+\rangle\langle 1|$.

The default transpiler backend does not constraint register connectivity and it allows for a plethora of basic gates including multi-controlled gates. Transpiling for a quantum computer would require a reduced set of basic gates and a mapping of the available quantum registers instead. Note that transpiling also simplifies gate combinations that are not restricted by barriers resulting in the aforementioned U_2 gate.

Analogously to the Silq scripts, this non-recursive implementation is extended towards a full implementation of the recursive Bernstein–Vazirani algorithm in Qiskit. The randomly chosen values of the secrets $\boldsymbol{s}_k(\boldsymbol{x}_0,\dots,\boldsymbol{x}_{k-1})$ are once again stored as a lookup table. The final program is given in Listing 4.5.

The focus of this script lies on the implementation of conditionals as controlled quantum gates. The definition of U_{f_d} at lines 25 to 31 has to account for all instances of $\boldsymbol{x}_0,\dots,\boldsymbol{x}_{d-1}$ and therefore iterates over 2^{dn} values for these qubits. Each of these values additionally controls the controlled X gate, added via the `control` method, resulting in gates with $d \cdot n + 1$ control qubits.

The gates U_{f_k} for k from 0 to $d-1$ at lines 34 to 46 also iterate over 2^{kn} values of $\boldsymbol{x}_0,\dots,\boldsymbol{x}_{k-1}$ but then additionally consider the 2^n options for $\boldsymbol{x}_k$ and check for the control register value $\boldsymbol{a}$ to equate to $\boldsymbol{s}_{k+1}(\boldsymbol{x}_0,\dots,\boldsymbol{x}_k)$. This definition requires multi-controlled X gates with $(k+2)\cdot n$ control qubits for all values of $\boldsymbol{x}_0,\dots,\boldsymbol{x}_k$ where the ith component of both $\boldsymbol{x}_k$ and $\boldsymbol{s}_k(\boldsymbol{x}_0,\dots,\boldsymbol{x}_{k-1})$ is 1.

Thus, this implementation is a close translation of the presented Silq code into Qiskit.

The outputs of running this Qiskit script are similar to the non-recursive variant. While the text is of the same form, the diagram images show the increased complexity of the quantum gates. An example for the circuit overview image is given in Figure 4.4.

Note that the decomposed and transpiled output images are too detailed to be displayed here. Figures 4.5 and 4.6 show the first two rows of these diagrams.

This general implementation of the oracle functions U_{f_k} using lookup tables requires an exponential number of gates with respect to both parameters n and d. However, treating these oracles as black boxes, the findings about this algorithm's complexity still hold as discovered in the previous chapter.

Figure 4.1 Example output image `1bv-circuit.png` of the program in Listing 4.4

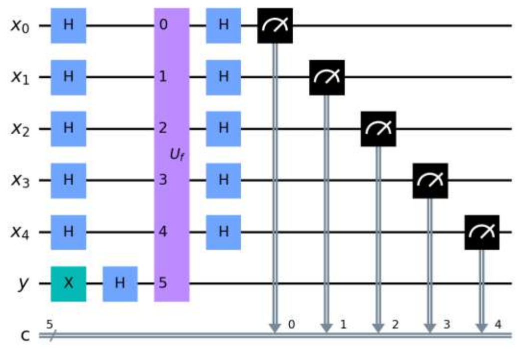


Figure 4.2 Example output image `1bv-decomposed.png` of the program in Listing 4.4

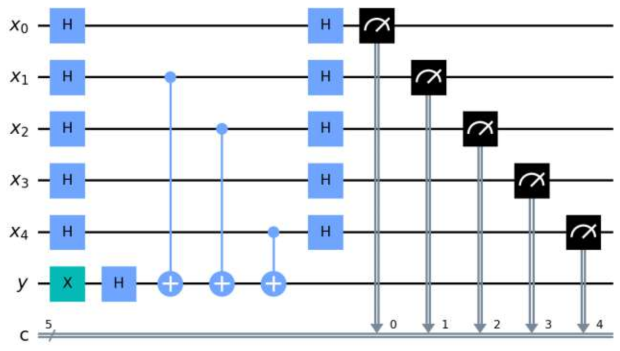
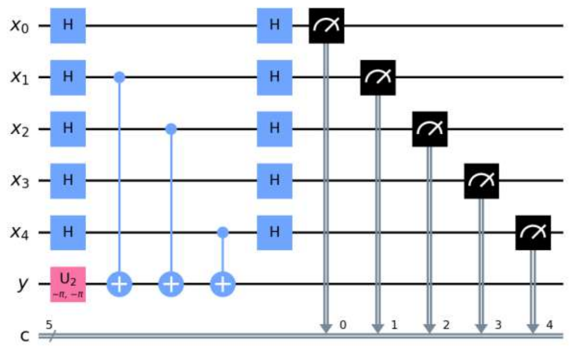


Figure 4.3 Example output image `1bv-transpiled.png` of the program in Listing 4.4



Listing 4.5 Qiskit implementation of the recursive Bernstein–Vazirani algorithm

```
from qiskit import *
from qiskit.circuit.library import CXGate, MCXGate
from random import getrandbits

n = 2
d = 3
fmt = dict(output='mpl', plot_barriers=False, fold=18)

def indexed(x):
    return enumerate([b == '1' for b in reversed(f'{x:0{n}b}')])

def concat(xss):
    return [x for xs in xss for x in xs]

# sk(x0,..,x[k-1])
sk = [[getrandbits(n) for x in range(2**(k*n))] for k in range(d+1)]
print(f's0␣=␣{sk[0][0]:0{n}b}')

xk = [QuantumRegister(n, f'x{k}') for k in range(d+1)]
y = AncillaRegister(1, 'y')
c = ClassicalRegister(n, 'c')
qc = QuantumCircuit(*xk, y, c)

# fd(x0,..,xd) = xd . sd(x0,..,x[d-1])
Ufd_qc = QuantumCircuit(*xk, y)
for ix in range(1 << d*n):
    for i, bit in indexed(sk[d][ix]):
        if bit:
            gate = CXGate().control(d*n, ctrl_state=ix)
            Ufd_qc.append(gate, concat(xk[:d]) + [xk[d][i]] + y[:])
Ufd = Ufd_qc.to_gate(label=f'$U_{{f_{{{d}}}}}$')
```

```

# fk(x0,..,xk,a) = xk . sk(x0,..,x[k-1]) [ a == s[k+1](x0,..,xk) ]
Ufk = []
for k in range(d):
    Ufk_qc = QuantumCircuit(*xk[:k+2], y)
    for ix0 in range(1 << k*n):   # x0,..,x[k-1]
        for ixk in range(1 << n): # xk
            for i, bit in indexed(sk[k][ix0]):
                if bit and ixk & (1 << i):
                    ix = (ixk << k*n) + ix0
                    gate = MCXGate(n, ctrl_state=sk[k+1][ix])
                    gate = gate.control(n, ctrl_state=ixk)
                    gate = gate.control(k*n, ctrl_state=ix0)
                    Ufk_qc.append(gate, Ufk_qc.qubits)
    Ufk.append(Ufk_qc.to_gate(label=f'$U_{{f_{{{k}}}}}$'))

for k in range(d+1):
    qc.h(xk[k])
qc.x(y)
qc.h(y)
qc.barrier()

def rbv(k):
    if k == d:
        qc.append(Ufd, concat(xk[:]) + [y])
        qc.barrier()
        return
    rbv(k + 1)
    qc.h(xk[k+1])
    qc.barrier()
    qc.append(Ufk[k], concat(xk[:k+2]) + [y])
    qc.barrier()
    qc.h(xk[k+1])
    qc.barrier()
    rbv(k + 1)

rbv(0)

qc.h(xk[0])
qc.measure(xk[0], c)
qc.draw(**fmt, filename='rbv-circuit.png')
qc.decompose('*U*').draw(**fmt, filename='rbv-decomposed.png')
```

```

sim = Aer.get_backend('aer_simulator')
tp = transpile(qc, sim)
tp.draw(**fmt, filename='rbv-transpiled.png')
print(sim.run(tp, shots=1024).result().get_counts())
```

Figure 4.4 Example output image `rbv-circuit.png` of the program in Listing 4.5

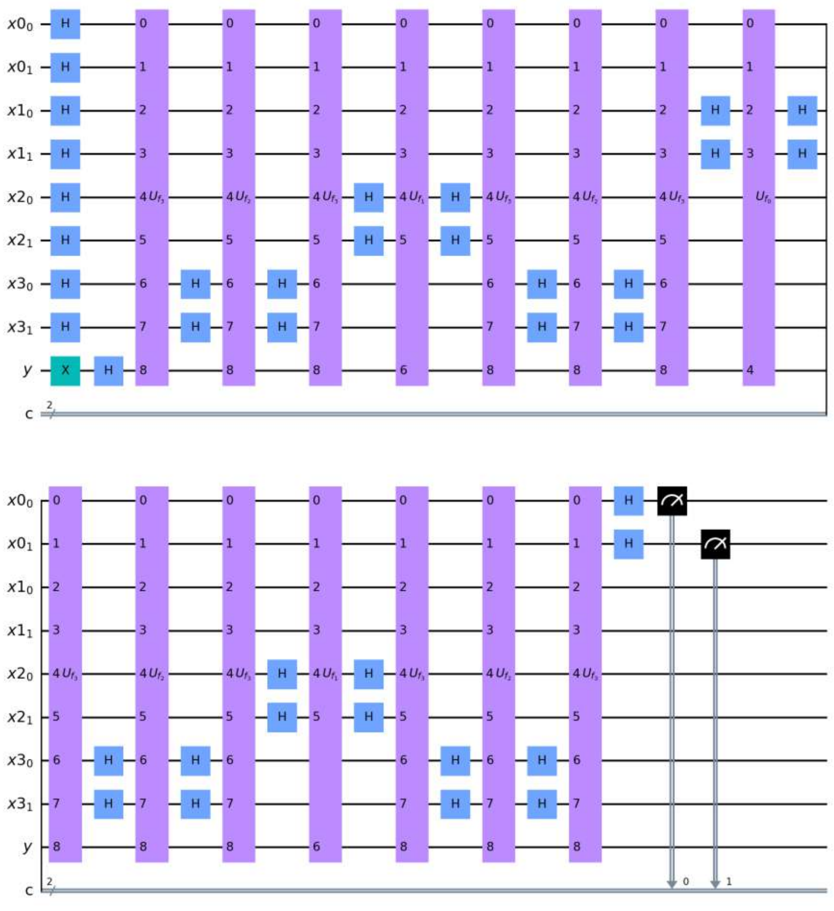
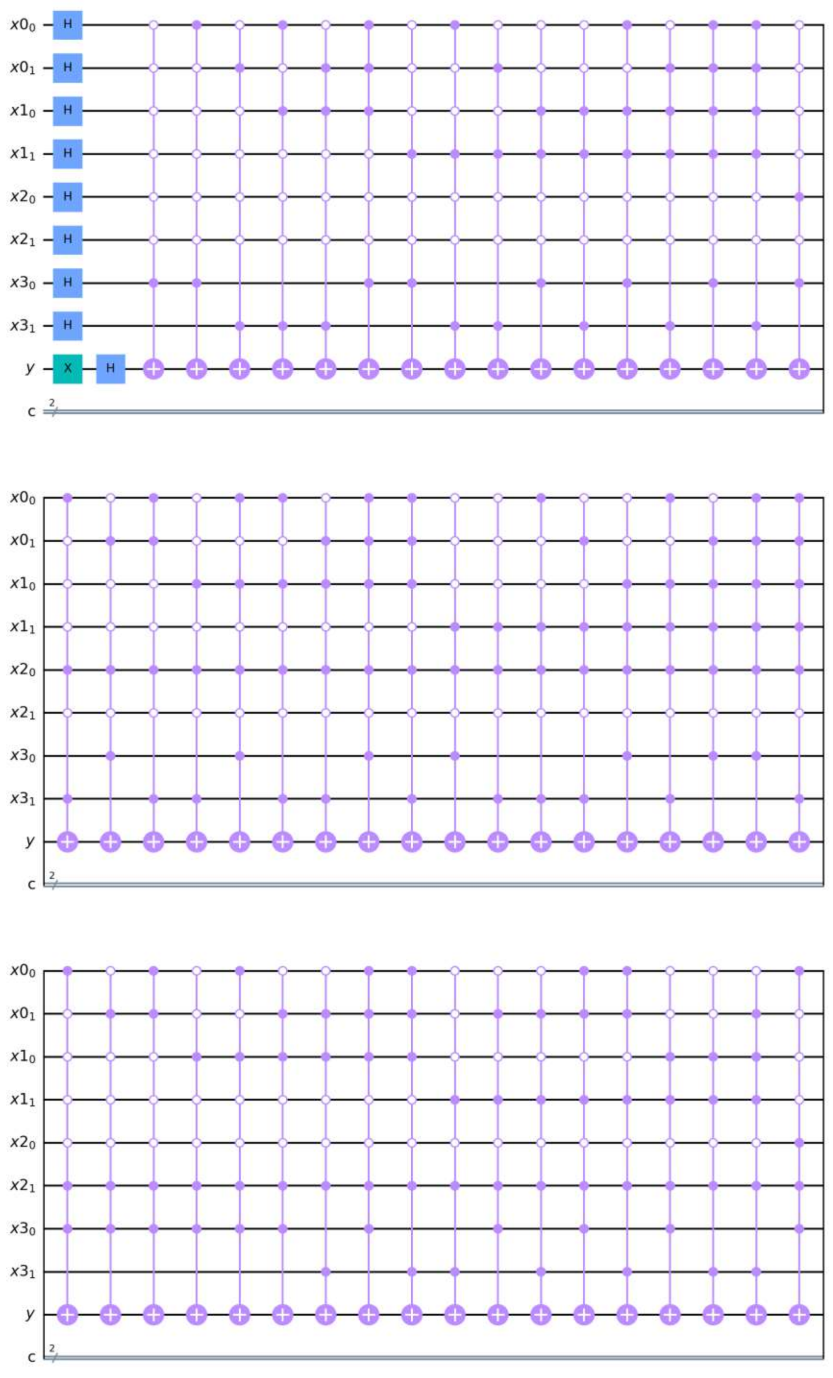
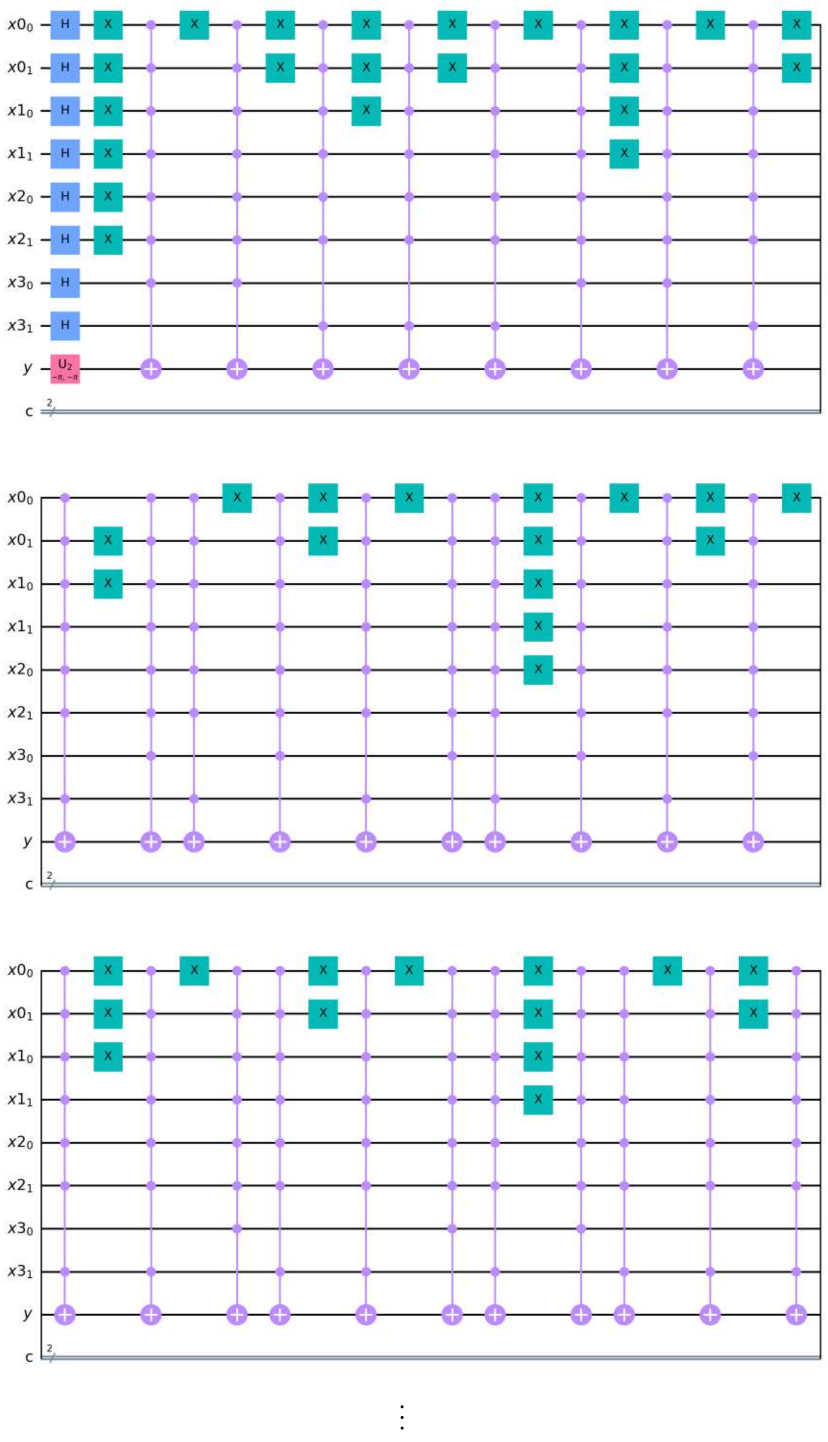


Figure 4.5 Initial part of an example output image `rbv-decomposed.png` of the program in Listing 4.5



⋮

Figure 4.6 Initial part of an example output image `rbv-transpiled.png` of the program in Listing 4.5



CHAPTER 5

Discussion

The Bernstein–Vazirani problem is one of the most important problems for examining the power of quantum computers and comparing it against the class of problems that are efficiently computable on classical computers.

The complexity classes BPP and BQP denote the classes of decision problems solvable with bounded error probability in polynomial time by classical and quantum computers, respectively. As shown in [BV97], the recursive Bernstein–Vazirani problem belongs to the complexity class BQP but not to the class BPP relative to the given oracle, assuming that one-way functions exist and thus that $\mathsf{P} \neq \mathsf{NP}$.

A one-way function is computable in polynomial time but cannot be inverted easily, that is to say that there is no procedure that gives an input for any given output of this function in polynomial time. If P is equal to NP, one-way functions cannot exist as shown in *Foundations of Cryptography* by Oded Goldreich [Gol01, Chapter 2].

Therefore, implementing the recursive Bernstein–Vazirani algorithm in modern quantum programming languages helps to verify the power of quantum computation with respect to computational complexity.

Recall the number of oracle invocations of the recursive algorithm. The classical implementation used

$$\frac{n}{n-1} \cdot (n^{d+1} - 1) \tag{5.1}$$

oracle calls, while the quantum version only made

$$2^{d+1} - 1 \tag{5.2}$$

calls. For $d = 0$ we get the non-recursive version as well.

Since we dropped the original constraint between n and d, we can revisit the case when $n = 2^{d+1}$ or equally $d = \operatorname{ld} n - 1$ where ld denotes the logarithm to the base 2. In this case the classical problem solution makes

$$\frac{n}{n-1} \cdot (n^{\operatorname{ld} n} - 1) \tag{5.3}$$

oracle calls and the quantum program uses only $n-1$ invocations. Thus, the quantum computer solves the problem in a polynomial number of calls of the given oracle, whereas a classical machine needs a superpolynomial number of calls with respect to the input size to run the algorithm.

Hence, we could demonstrate the computational power of quantum computers by designing and implementing the recursive Bernstein–Vazirani algorithm in modern quantum programming languages.

Bibliography

[Aar03] Scott Aaronson. *Quantum Lower Bound for Recursive Fourier Sampling.* In: *Quantum Information and Computation* 2003, 3(2), pp. 165–174. DOI: 10.48550/arXiv.quant-ph/0209060.

[Bic+20] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. *Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics.* In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 286–300. ISBN: 9781450376136. DOI: 10.1145/3385412.3386007.

[Bic+23] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. *What is Silq?* Feb. 1, 2023. URL: https://silq.ethz.ch/.

[BV97] Ethan Bernstein and Umesh Vazirani. *Quantum Complexity Theory.* In: *SIAM J. Comput.* 1997, 26(5), pp. 1411–1473. ISSN: 0097-5397. DOI: 10.1137/S0097539796300921.

[Cle+98] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. *Quantum Algorithms Revisited.* In: *Proc. R. Soc. A* 1998, 454(1969), pp. 339–354. DOI: 10.1098/rspa.1998.0164.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2001. ISBN: 978-0-511-54689-1. DOI: 10.1017/CBO9780511546891.

[Joh08] Benjamin Edward Johnson. *Upper and Lower Bounds for Recursive Fourier Sampling.* PhD thesis. University of California, Berkeley, 2008. ISBN: 978-0-549-83340-6.

[McK12] Matthew McKague. *Interactive proofs with efficient quantum prover for recursive Fourier sampling.* In: *Chicago Journal of Theoretical Computer Science* Sept. 2012, 2012(6), pp. 1–10. DOI: 10.4086/cjtcs.2012.006.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2010. ISBN: 978-1-107-00217-3.

[Qis22] Qiskit contributers (listed at https://github.com/Qiskit/qiskit/blob/master/AUTHORS). *Qiskit: An Open-source Framework for Quantum Computing.* 2022. DOI: 10.5281/zenodo.2573505.

[SM12] Jamie Smith and Michele Mosca. *Algorithms for Quantum Computers*. In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1451–1492. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_43.