6-2-2022

Methodologies for Quantum Circuit and Algorithm Design at Low and High Levels

Edison Tsai
Portland State University

Methodologies for Quantum Circuit and Algorithm Design at Low and High Levels

by

Edison Tsai

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Marek A. Perkowski, Chair
Xiaoyu Song
Fu Li
Steven Bleiler

Portland State University
2022

Abstract

Although the concept of quantum computing has existed for decades, the technology needed to successfully implement a quantum computing system has not yet reached the level of sophistication, reliability, and scalability necessary for commercial viability until very recently. Significant progress on this front was made in the past few years, with IBM planning to create a 1000-qubit chip by the end of 2023, and Google already claiming to have achieved quantum supremacy. Other major industry players such as Intel and Microsoft have also invested significant amounts of resources into quantum computing research.

Any viable computing system requires both hardware and software to work together harmoniously in order to perform useful computations. While the achievements of IBM and other companies represent a large step forward for quantum hardware, many gaps remain to be filled with respect to the corresponding software. Specifically, there is currently no clear path towards a complete process for translating quantum algorithms into physical operations that are directly executable on quantum hardware. Such a process is analogous to a compiler that translates programs written in a high-level language into executable machine instructions on a conventional digital computer, and it is necessary if quantum computers are to be harnessed to perform practically useful computations. Existing work has addressed individual components of this process, but so far no unified method for translating the whole of a quantum algorithm into executable operations has been described.

I make substantial progress towards filling this gap by describing a set of high-level and low-level quantum circuit design techniques, which when taken together reduce the need of a circuit designer to be concerned with low-level details. On the high-level side, I describe

an approach or strategy to designing quantum oracles for Grover's algorithm that allows it to be applied to several types of problems. This approach involves designing oracles in terms of high-level blocks such as counters, multiplexers, comparators, and arbitrary Boolean functions. The implementations of these blocks in terms of lower-level quantum gates are demonstrated in a way that makes it clear that scaled-up versions of them can be generated in a completely automated fashion. For a specific class of problems, which I call state-space path planning problems, I also introduce a paradigm for quantum oracle design that involves representing the problem in terms of individual states and moves. Problems of this sort have applications in robotics and games.

Low-level techniques that I introduce include methods for realizing both single-output and multiple-output Boolean functions, as well as reversible functions with multiple-valued inputs and outputs, on the quantum gate level. These realization methods can be used to translate the Boolean functions used as high-level blocks in quantum oracle design into low-level gates. The low-level gates used are two-qubit controlled gates such as controlled-NOT and controlled-V whose physical realizations have been extensively studied. In particular, I demonstrate that realizing symmetric functions, which are a subset of Boolean functions, directly using these low-level gates can give better results than the usual method of using higher-level Toffoli gates as intermediates. I also demonstrate that the problem of realizing a reversible Boolean function with many inputs and many outputs "in place", that is, using the same qubits to hold the inputs and outputs of the function, can be converted into realizing a sequence of single-output Boolean functions. I describe the realization methods in sufficient detail for a skilled programmer to implement them as part of a CAD tool for quantum circuit design.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction and Preliminaries

Proposals to harness quantum phenomena for computing and information processing go back to at least as far as 1982 [1]. It has been known for a long time that a working quantum computational device of sufficient scale, if one could be built, would be able to exceed the theoretical algorithmic-complexity limits for a classical computer on certain tasks [2, 3]. This ability for a quantum computer to outperform a classical computer is known as *quantum supremacy*. The well-known algorithms due to Shor [4, 5] and Grover [6, 7, 8] provide theoretical examples of quantum supremacy for computational tasks of practical interest. Shor's algorithm allows a quantum computer to find the prime factorization of an integer in polynomial time relative to its number of digits, while Grover's algorithm allows a quantum computer to search an unsorted collection of N items in $\mathcal{O}\left(\sqrt{N}\right)$ time where a classical computer requires $\mathcal{O}(N)$ time on average.

Despite decades of theoretical interest in quantum computation, practical issues with the physical realization of a quantum computing device remained unsolved for a long time. However, recent advances have made much more likely the prospect of a commercially viable quantum computing system being available in the near future. IBM has plans for a 1000-qubit chip by the end of 2023 [9], while Google announced the achievement of quantum supremacy in 2019 [10], although this claim turned out to be controversial [11].

Assuming that a working quantum computational device of practical scale will become available in the future, there still remains the question of how quantum algorithms will be implemented in practice. There has been little published work demonstrating the application of a quantum algorithm to solve a practical problem in full detail. For instance, Grover's algorithm requires a quantum circuit known as a *quantum oracle* to define the search criteria that it uses, so in order to use Grover's algorithm to solve a given problem, one must first design a quantum oracle for that problem, and the design must ultimately be decomposable into low-level gates.

Modern digital hardware and software design both involve a high degree of automation. In particular, CAD tools for very-large-scale-integrated (VLSI) circuit design allow human designers to only be concerned with high-level specifications, with the CAD tool automating the process of generating low-level primitives like individual gates or transistors from these specifications. Similarly, most modern software is written in high-level programming languages, with a compiler being responsible for automatically generating executable machine instructions from the high-level code. In contrast, current quantum languages like Qiskit [12] are low-level, similar to assembly language, and describe quantum circuits on the level of individual gates. The creation of CAD tools for quantum circuits and algorithms, *i.e.*, "quantum compilers", is currently an underdeveloped area of research. Such tools will be necessary, or at least very useful, to create complicated quantum circuits containing tens of thousands of gates or more.

In order to create CAD tools for quantum circuits, and to create a partially-automated workflow for quantum circuit design, at least two ingredients are necessary: the ability to realize functions in terms of low-level gates, and the ability to express a problem to be solved in terms of well-defined functions. The objective of this dissertation is to contribute to the

creation of CAD tools for quantum circuits by showing how both of these requirements can be met. First, I demonstrate new methods for realizing non-reversible and reversible functions using quantum circuits. My realization method for non-reversible functions realizes them directly using low-level quantum gates which have no classical counterpart, unlike most existing approaches which are often based on techniques from classical digital logic design and not specifically optimized for a quantum setting. I show that my approach gives better results than classically-based techniques for a specific class of functions known as symmetric functions. For reversible functions, I introduce a new concept of *distance gates*, which are used as building blocks in a cycle-based algorithm for realizing functions expressed as permutations. Distance gates have the very useful property that they are easily generalized to multiple-valued quantum circuits, which have attracted attention because multiple-valued qudits are a more natural model than binary qubits for some quantum systems. It is also known that using multiple-valued qudits to augment an otherwise binary system can allow for certain binary operations to be performed more efficiently. My synthesis method using distance gates is well-suited to such systems because it naturally has the ability to work with multiple-valued quantum systems as well as with quantum systems that mix qudits and qubits.

The second main achievement of this dissertation is to demonstrate the application of Grover's algorithm to problems of practical interest in full detail. "Full detail" means that I show how the quantum oracles used in Grover's algorithm can be designed using functional blocks such as incrementers, multiplexers, and comparators, and then show how to implement these functional blocks on a lower level. These functional blocks can then be reused for other problems as well. In this way, I therefore create an approach for solving certain types of problems using Grover's algorithm: the quantum oracle for the problem is first expressed

in terms of high-level blocks, which can then be automatically translated into low-level gates. This approach contributes to the future development of a partially-automated workflow for solving problems using Grover's algorithm, in which a human designer only needs to design a quantum oracle using high-level functions and does not need to directly deal with low-level gates.

In Sections 1.1 through 1.4, I review some preliminary background knowledge on quantum computing and define terminology that will be used in the remainder of this dissertation. Only the topics most relevant to this dissertation are covered; for a more thorough treatment of quantum computing and quantum information theory, the reader is invited to consult [13]. Section 1.5 gives an overview of the organization of the remainder of this dissertation.

1.1 Quantum information

1.1.1 Qubits and quantum states

In a conventional digital computer, the fundamental unit of information is a bit (a contraction of "binary digit"). A bit is an entity that may be in one of two states, which can be denoted 0 and 1. The quantum analog of a bit is a qubit (a contraction of "quantum bit"), which is the fundamental unit of quantum information. A qubit is an entity that may take on states analogous to those of a bit, denoted $|0\rangle$ and $|1\rangle$. However, a qubit may also take on a continuum of infinitely many other possible states. Specifically, according to the postulates of quantum mechanics, a qubit that can take on one of two independent states $|0\rangle$ and $|1\rangle$ can also take on any superposition, or linear combination, of $|0\rangle$ and $|1\rangle$. In other words, the state of a qubit can be described by the expression

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

where $|\psi\rangle$ is the state and α and β are complex numbers. The notation $|\cdot\rangle$ is called a *ket* and is often used in quantum mechanics to denote the state of a quantum system. Mathematically, kets represent vectors in a Hilbert space, which contains all the possible states of a given quantum system. When the system is a single qubit, the possible states consist of linear combinations of $|0\rangle$ and $|1\rangle$, as seen in (1.1), meaning that the Hilbert space is two-dimensional with $|0\rangle$ and $|1\rangle$ forming a basis for it.

The postulates of quantum mechanics also hold that there is no physical difference between states that are scalar multiples of one another. In mathematical terminology, the space of the possible states of a quantum system is a projective space. Thus, if the state of a qubit is $\alpha|0\rangle+\beta|1\rangle$, this state is equally well described by any vector of the form $c\big(\alpha|0\rangle+\beta|1\rangle\big)$, where c is any nonzero complex number. The zero vector is excluded from the state space completely; in other words, the zero vector is not a valid representation of any quantum state. Since all quantum states are represented by nonzero vectors, they may be normalized by scaling them to a vector magnitude of 1. For example, the vector $3|0\rangle + 4|1\rangle$ has a vector magnitude of $\sqrt{|3|^2 + |4|^2} = 5$ and is a non-normalized representation of a quantum state. This vector may be normalized by dividing by its magnitude, giving $\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$. From now on, I will assume that all vector representations of quantum states are normalized.

Superpositions between the two basis states $|0\rangle$ and $|1\rangle$ of a qubit cannot be directly observed. When a qubit is measured—that is, examined to determine its state—the measurement can only return a result of either $|0\rangle$ or $|1\rangle$. Furthermore, quantum measurements disturb the qubit being measured, collapsing its state to either $|0\rangle$ or $|1\rangle$ (depending on the result returned by the measurement) and destroying any superposition. Quantum measurement is postulated to be a probabilistic process that is inherently unpredictable. In other words, it is in general impossible to predict the outcome of any one particular measurement, but it is

possible to predict the overall frequencies with which the two basis states will be observed if a large collection of qubits with identical states is measured. If $\alpha|0\rangle + \beta|1\rangle$ is a normalized representation of the state of a qubit just prior to measurement, then the probability that the measurement returns $|0\rangle$ is $|\alpha|^2$ and the probability that the measurement returns $|1\rangle$ is $|\beta|^2$. Thus, if a qubit in the state $\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$ is measured, the probability of observing either outcome is $\frac{1}{2}$, whereas if the qubit is instead in the state $\frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle$, then the probability of observing $|1\rangle$ is $\frac{3}{4}$ and that of observing $|0\rangle$ is only $\frac{1}{4}$.

Referring to the right-hand side of (1.1), I will call the individual terms $\alpha|0\rangle$ and $\beta|1\rangle$ *components* of the state $|\psi\rangle$ and the coefficients α and β their corresponding *amplitudes*. Since the amplitudes are complex numbers, two states can behave indistinguishably with respect to measurement even though they might be distinguishable by other means. For instance, if one qubit is in the state $\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$ and another is in the state $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$, measurement of either qubit returns $|0\rangle$ with probability 0.5. However, these two states are said to differ in their relative phase because the complex phase difference between the $|0\rangle$ and $|1\rangle$ components is zero for the first qubit but 180 degrees for the second qubit. These two states can be distinguished by measurement if the measurement is preceded by action of an appropriate quantum gate; quantum gates are discussed in Section 1.2. The states $\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$ and $\frac{1}{\sqrt{2}}\big(-|0\rangle - |1\rangle\big)$, though, are not distinguishable by any means and are in fact just different representations of the same state because they are scalar multiples of each other, as discussed earlier. This scalar multiple can be made explicit by writing

$$\frac{1}{\sqrt{2}}\big(-|0\rangle - |1\rangle\big) = -1 \cdot \frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big). \tag{1.2}$$

The -1 on the right-hand side of (1.2) is referred to as a *global phase factor*, or simply a *global phase* for short, since it carries only a complex phase and does not affect the

normalization of the state, having a magnitude of 1. Unlike relative phase, a global phase can always be ignored since it is just a particular type of scalar multiple.

1.1.2 Joint states

When two qubits come together to form a single system, their states can be combined to form a *joint state*, which is a single mathematical entity that describes the states of both qubits simultaneously. The joint state of two initially independent qubits is given by an operation known as the tensor product or Kronecker product. The tensor product, denoted $\otimes$, of two quantum states may be computed by multiplying them in the same fashion as multiplying polynomials, and then combining all possible pairs of basis states from the two qubits to create a new set of four basis states. For instance, if two qubits are in the states $\alpha|0\rangle + \beta|1\rangle$ and $\gamma|0\rangle + \delta|1\rangle$, respectively, then their joint state is given by

$$\begin{aligned}&\big(\alpha|0\rangle + \beta|1\rangle\big) \otimes \big(\gamma|0\rangle + \delta|1\rangle\big) \\ &\qquad = \alpha\gamma|0\rangle \otimes |0\rangle + \alpha\delta|0\rangle \otimes |1\rangle + \beta\gamma|1\rangle \otimes |0\rangle + \beta\delta|1\rangle \otimes |1\rangle \end{aligned} \tag{1.3}$$

where $|0\rangle \otimes |0\rangle$, $|0\rangle \otimes |1\rangle$, $|1\rangle \otimes |0\rangle$, $|1\rangle \otimes |1\rangle$ form a set of basis states for the combined two-qubit system. These four basis states correspond to the four possible states for a system of two classical bits. Usually, the product $|0\rangle \otimes |0\rangle$ is abbreviated as $|00\rangle$, and similarly for the other three basis states, so that the joint state in the previous example could also be written

$$\alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle. \tag{1.4}$$

Joint states are similarly defined for systems containing more than two qubits: the joint state of an n-qubit system is given by the tensor product of the individual qubits' states. For

instance, given a three-qubit system, where the individual qubits are in the states $\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle$, $|1\rangle$, and $\frac{4}{5}|0\rangle - \frac{3}{5}|1\rangle$, respectively, the joint state of the system is

$$\begin{aligned}\Big(\frac{3}{5}|0\rangle + \frac{4}{5}|1\rangle\Big) \otimes |1\rangle \otimes \Big(\frac{4}{5}|0\rangle - \frac{3}{5}|1\rangle\Big) \\ = \frac{1}{25}\big(12|010\rangle - 9|011\rangle + 16|110\rangle - 12|111\rangle\big).\end{aligned} \tag{1.5}$$

The basis states for the three-qubit system are $|000\rangle$, $|001\rangle$, $|010\rangle$, and so on up to $|111\rangle$, where $|abc\rangle$ is a synonym for $|a\rangle \otimes |b\rangle \otimes |c\rangle$ as in the two-qubit case. In general, an n-qubit system has 2^n basis states, which are given by $|00 \ldots 0\rangle$ through $|11 \ldots 1\rangle$. Given a joint state of an n-qubit system, I will refer to a term containing a single basis state, such as $\alpha\gamma|00\rangle$ in (1.4), as a component of the state. The coefficient of a component, such as $\alpha\gamma$, will be called the component's amplitude, just as in the single-qubit case.

An alternative notation for the basis states of an n-qubit system uses $|0\rangle$, $|1\rangle$, $|2\rangle$, and so on up to $|2^n - 1\rangle$. In this notation, if i is a natural number less than 2^n, then $|i\rangle$ denotes the basis state corresponding to the base-two representation of i. For instance, in a three-qubit system, $|5\rangle$ in this notation is a synonym for $|1\rangle \otimes |0\rangle \otimes |1\rangle$ because the base-two representation of 5 is 101. An arbitrary state of an n-qubit system can then be written as a sum of components

$$a_0|0\rangle + a_1|1\rangle + a_2|2\rangle + \cdots + a_{2^n-1}|2^n - 1\rangle = \sum_{i=0}^{2^n-1} a_i|i\rangle. \tag{1.6}$$

I will make use of this last notation later when discussing Grover's search algorithm.

1.1.3 Entangled states

As described in Section 1.1.2, the states of two or more independent qubits may be combined via the tensor product to obtain the joint state of the qubits considered as a single system.

The reverse operation is not always possible: a state of a multi-qubit system cannot always be factorized into the tensor product of single-qubit states. For instance, consider a two-qubit system in the state

$$\frac{1}{\sqrt{2}}\big(|01\rangle + |10\rangle\big). \tag{1.7}$$

Suppose that this state could be factorized into the tensor product of two single-qubit states $\alpha|0\rangle + \beta|1\rangle$ and $\gamma|0\rangle + \delta|1\rangle$. Then, comparing (1.4) with (1.7), we have $\alpha\gamma = 0$, $\alpha\delta = 1$, $\beta\gamma = 1$, and $\beta\delta = 0$. Since $\alpha\gamma = 0$, at least one of α or γ must be zero. But α cannot be zero since $\alpha\delta = 1$, and γ cannot be zero either since $\beta\gamma = 1$. This is a contradiction, showing that the state represented by (1.7) cannot be factorized into the tensor product of two single-qubit states. Such a state is called an *entangled* state: the two qubits' states are inseparably connected, and one cannot meaningfully speak of the state of one of the qubits on its own, but only of the state of the whole two-qubit system.

As a further indication that the state (1.7) is entangled, one can also observe that the results of measurement of the two qubits are correlated. If a measurement of the first qubit returns $|0\rangle$, then a subsequent measurement of the second qubit must return $|1\rangle$, because $|00\rangle$ is not a possible outcome for a measurement of the state (1.7). On the other hand, if a measurement of the first qubit returns $|1\rangle$, then a subsequent measurement of the second qubit must return $|0\rangle$, because $|11\rangle$ is not a possible outcome either. In other words, a measurement of one of the qubits affects the possible outcomes for a measurement of the other qubit. This phenomenon is counterintuitive and would not occur if the two qubits constituted independent systems.

I will not make direct use of entanglement phenomena in this dissertation, but it plays a role in Grover's algorithm, which I discuss and rely on later. Entanglement is also critical to other applications of quantum information processing, such as quantum teleportation. For

details, the reader is invited to consult [13] or any other similar source.

1.1.4 The Bloch sphere

The Bloch sphere is a geometrical device that assists in visualizing the space of possible states for a single qubit. Any state of the form $\alpha|0\rangle + \beta|1\rangle$ may be represented by a point on the surface of the Bloch sphere. By convention, the poles of the Bloch sphere, at the points of intersection with the z-axis, are taken to represent the basis states $|0\rangle$ and $|1\rangle$, with $|0\rangle$ being at the north and $|1\rangle$ at the south pole. The spherical colatitude (that is, the angle from the positive z-axis) of a point on the sphere determines the relative magnitudes of the $|0\rangle$ and $|1\rangle$ components, while the azimuthal angle (that is, the angle in the xy-plane) determines their relative phase. Specifically, if a point on the Bloch sphere has spherical coordinates (ϕ, θ), where ϕ is the colatitude and θ is the azimuthal angle, then this point represents the quantum state

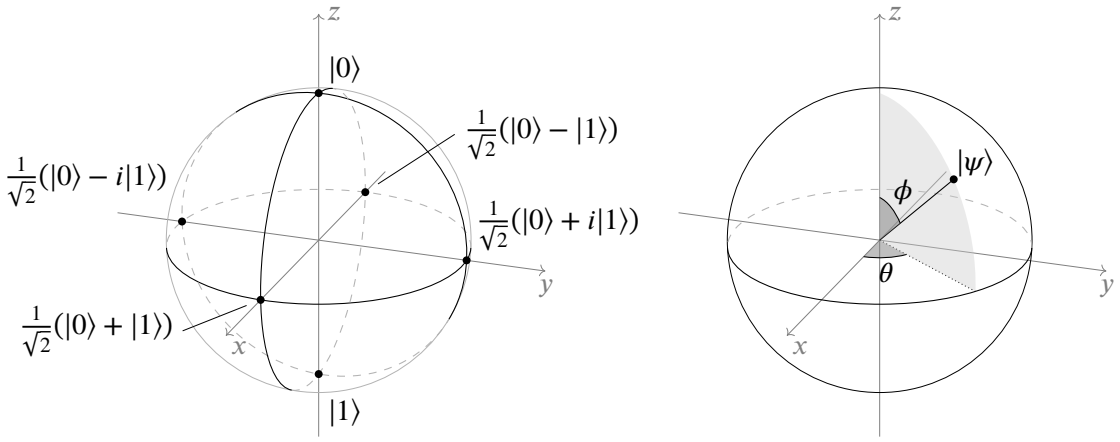
$$|\psi\rangle = \cos\frac{\phi}{2}\,|0\rangle + \sin\frac{\phi}{2}\,e^{i\theta}\,|1\rangle. \tag{1.8}$$

A depiction of the Bloch sphere with a few points labeled is shown in Figure 1.1a. The angles ϕ and θ defining an arbitrary quantum state $|\psi\rangle$ are shown in Figure 1.1b.

1.1.5 Column vector representation of quantum states

Since kets represent vectors, the state of a single qubit can also be written in conventional column-vector form. Specifically, we can identify the kets $|0\rangle$ and $|1\rangle$ with the standard basis vectors for a two-dimensional complex vector space:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \text{and} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1.9}$$



(a) Representation of the basis states and some superpositions. **(b)** Representation of an arbitrary state.

Figure 1.1: The Bloch sphere.

Upon making this identification, we can then write

$$\alpha|0\rangle + \beta|1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{1.10}$$

so the state of a single qubit can be represented by a two-dimensional complex vector whose components are just the state's probability amplitudes.

In column-vector form, tensor products can be computed by distributing the second vector over the components of the first vector:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \otimes \begin{bmatrix} \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \\ \beta \begin{bmatrix} \gamma \\ \delta \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}. \tag{1.11}$$

The column-vector representation is useful when working with matrix representation of quantum gates, which are discussed in Section 1.2.

1.1.6 Multiple-valued qudits

The overwhelming majority of digital computers in use today are based on the bit as the fundamental unit of information, since an information-carrying unit with two possible states is the minimum required for a useful information processing system. However, experimental processors that use information-carrying units with more than two possible states also exist. Analogously, in principle, a quantum computational system can use quantum information units with more than two basis states rather than qubits. Such quantum information units are sometimes called *qudits*, a contraction of "quantum digit". For instance, if we consider a quantum system with three states, $|0\rangle$, $|1\rangle$, and $|2\rangle$, then an arbitrary state of this system takes the form

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle + \gamma|2\rangle, \tag{1.12}$$

where α, β, and γ are complex numbers, not all equal to zero. A quantum information unit with exactly three states can also be called a *qutrit*. A quantum computational system that makes use of qudits is described as *multiple-valued*. The phrase "multiple-valued qudit" is technically redundant, but I will sometimes use it for emphasis.

The mathematical representations of qudits behave very similarly to their qubit counterparts. Given a vector representation of the state of a qudit, any nonzero scalar multiple of that vector is also considered to represent the same state. Vector representations can be normalized to have a magnitude of 1, and from now on all such representations are assumed to be so normalized. A measurement of a qudit always returns one of the basis states for the qudit and collapses the qudit's state to the result that was observed, destroying any superposition. The probabilities of the possible measurement results are given by the squares of the magnitudes of the corresponding components of the state. For instance, if a qutrit is in the

state $\frac{1}{2}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle + \frac{1}{2}|2\rangle$, then a measurement of this qutrit returns $|0\rangle$ with probability $\frac{1}{4}$, $|1\rangle$ with probability $\frac{1}{2}$, and $|2\rangle$ with probability $\frac{1}{4}$. The state of a system containing multiple qudits is described by the tensor product of the individual qudits' states.

There is unfortunately no easily visualizable analog of the Bloch sphere for multiple-valued qudits. However, the ket representation of a qudit's state is easily rewritten as a column vector in the same manner as for a qubit. Given a qutrit with basis states $|0\rangle$, $|1\rangle$, and $|2\rangle$, we make the identifications

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \text{ and } |2\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{1.13}$$

which leads to the expected representation of an arbitrary qutrit's state as a column vector:

$$\alpha|0\rangle + \beta|1\rangle + \gamma|2\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \tag{1.14}$$

Similar representations can be used for qudits with four, five, or any higher number of basis states. For a qudit with n possible basis states, we will follow the convention of numbering the states starting from 0, giving $\{|0\rangle, |1\rangle, \ldots, |n-1\rangle\}$ as the set of basis states. The number n will be called the *radix* of the qudit, so for instance a quantum computing system using qutrits can also be said to use qudits of radix 3. A qudit of radix n can also be referred to as an n-valued qudit. It is possible for a single quantum computing system to mix qudits of different radices; such systems are briefly discussed in Section 1.2.10 and the design of quantum circuits for them is considered in Chapter 6.

1.2 Quantum gates

Quantum gates are the operations that act on qubits and qudits in order to perform useful computations. A quantum gate may operate on any number of qubits/qudits; however, gates that operate on many qubits/qudits simultaneously are unlikely to be primitive (that is, directly realizable as a low-level physical operation) and may require a large number of primitive operations to emulate. All quantum gates are reversible: given the output state of any quantum gate, it is always possible to reconstruct the original input state prior to the gate's operation. Equivalently, every quantum gate must have a corresponding inverse gate that undoes its action.

In addition, every quantum gate can also be represented by a matrix. The matrix representation of a quantum gate completely determines all aspects of the gate's behavior. Given a gate G represented in matrix form, the effect of the gate on an input state $|\psi\rangle$ may be determined by matrix-vector multiplication: $G|\psi\rangle$ gives the gate's output state where $|\psi\rangle$ has been expressed in vector form as described in the preceding subsection. Any matrix that represents a quantum gate must be *unitary*, which is a stronger condition than simply being invertible. A matrix U is said to be unitary if it satisfies the property $U^\dagger U = I$, where I denotes the identity matrix and $U^\dagger$ denotes the *Hermitian adjoint* of the matrix U, defined as the complex conjugate of the transpose of U.

1.2.1 The identity gate

A no-op can be considered a quantum "gate" that simply leaves a qubit's state unchanged. Mathematically, it is sometimes useful to treat the lack of an operation as a quantum gate in

and of itself, the identity gate, which is represented by the 2×2 identity matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{1.15}$$

In a schematic diagram for a quantum circuit, an identity gate amounts to just a "wire", although, as will be explained later, these are not actual physical wires. Quantum circuits are discussed in Section 1.3.

1.2.2 The inverter or NOT gate

The quantum inverter or NOT gate operates on a single qubit and is represented by the following matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{1.16}$$

Multiplying this matrix by the vector $\begin{bmatrix} \alpha & \beta \end{bmatrix}^{\mathsf{T}}$, we see that the inverter acts on an arbitrary input state $\alpha|0\rangle + \beta|1\rangle$ to produce $\beta|0\rangle + \alpha|1\rangle$. In other words, the inverter simply exchanges the amplitudes of the two components of its input state. In particular, the inverter acts on the state $|0\rangle$ to produce $|1\rangle$, and on $|1\rangle$ to produce $|0\rangle$. This behavior is consistent with that of an inverter or NOT gate in classical digital logic, therefore allowing the quantum inverter to be considered an extension of the classical inverter to a quantum setting.

The action of the inverter can also be geometrically visualized using the Bloch sphere: inverting a qubit corresponds to a 180-degree rotation of the Bloch sphere around the x-axis. This visualization is helpful when considering the V gate and other root-of-NOT gates, discussed in Sections 1.2.4 and 1.2.5.

1.2.3 The Hadamard gate

The Hadamard gate operates on a single qubit and is represented by the following matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{1.17}$$

Unlike the inverter, the Hadamard gate has no classical analog, so it can be considered an "inherently quantum" gate. The Hadamard gate lacks a classical analog because it acts on basis states to produce superpositions, which do not exist in a classical setting. Using matrix multiplication, it is easy to verify in particular that the Hadamard acts on $|0\rangle$ to produce $\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$ and acts on $|1\rangle$ to produce $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$. The Hadamard gate is therefore useful for initializing a qubit or qubits to a superposition state, such as in Grover's algorithm, which is discussed in Section 1.4.

The Hadamard gate is a self-inverse gate: if two Hadamard gates act consecutively on the same qubit, the second gate undoes the action of the first and restores the qubit to its original state. This property can be concisely expressed as $H^2 = I$. The Hadamard gate is denoted by the schematic symbol shown in Figure 1.2b.

(a) The inverter. **(b)** The Hadamard gate. **(c)** The V gate. **(d)** The $V^\dagger$ gate.

Figure 1.2: Several single-qubit gates.

1.2.4 The V and $V^\dagger$ gates

The V gate is similar to the Hadamard gate in that it acts on basis states to produce superpositions. The V gate is represented by the following matrix:

$$V = \frac{1}{2}\begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}. \tag{1.18}$$

Unlike the Hadamard gate, the V gate is not self-inverse. Instead, when two V gates act consecutively on the same qubit, the cumulative effect on the qubit's state is the same as that of the inverter. This property can be expressed as $V^2 = X$, and because of it, the V gate can be called a "square-root-of-NOT" gate. The fact that $V^2 = X$ also implies that $V^4 = I$; that is, four V gates acting consecutively on one qubit leaves the qubit in the same state in which it started. These properties of the V gate are used in Chapters 2 through 4 for realizing symmetric functions using two-qubit gates.

Since the V gate is not self-inverse, it must have a corresponding distinct inverse gate. This inverse gate is the $V^\dagger$ gate, where, as mentioned before, the $\dagger$ sign denotes the Hermitian adjoint (the complex conjugate of the transpose) of a matrix. Since all quantum gates are represented by unitary matrices, the Hermitian adjoint of the matrix is also its inverse. Therefore, the $V^\dagger$ gate is represented by the following matrix:

$$V^\dagger = \frac{1}{2}\begin{bmatrix} 1-i & 1+i \\ 1+i & 1-i \end{bmatrix}. \tag{1.19}$$

Because of the inverse relationship between the V and $V^\dagger$ gates as well as the fact that $V^4 = \mathrm{I}$, exactly four distinct quantum states may be reached by starting from a basis state and applying sequences of V and $V^\dagger$ gates. These four states may be arranged into a circle,

as shown in Figure 1.3. One application of a V gate is equivalent to moving 90 degrees counterclockwise around this circle while one application of a $V^\dagger$ gate is equivalent to moving 90 degrees clockwise. This circle corresponds to a circle on the Bloch sphere, specifically, the circle of intersection of the Bloch sphere with the yz-plane. Therefore, the V gate acts as a 90-degree rotation of the entire Bloch sphere in this plane, or equivalently around the x-axis. The square-root-of-NOT property of the V gate then has a clear geometric interpretation: the composition of two 90-degree rotations around the x-axis is equivalent to a single 180-degree rotation around that same axis.

The V and $V^\dagger$ gates are denoted by the schematic symbols shown in Figures 1.2c and 1.2d.

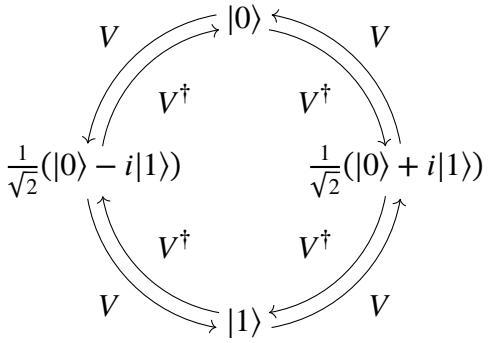


Figure 1.3: States reachable by applying V and $V^\dagger$ gates to $|0\rangle$ and $|1\rangle$, shown as a cross-section of the Bloch sphere in the yz-plane.

1.2.5 Rotation gates and higher roots of NOT

As previously mentioned in Sections 1.2.2 and 1.2.4, the actions of the NOT and V gates can be represented on the Bloch sphere by rotations of 180 and 90 degrees, respectively, around the x-axis. Rotations through smaller angles are also possible and can be used to describe higher root-of-NOT gates. In general, the gate $R_x(\theta)$, whose action corresponds to

a rotation through the angle θ around the x-axis of the Bloch sphere, is represented by the unitary matrix

$$R_x(\theta) = \exp\left(\frac{-i\theta}{2}X\right) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \tag{1.20}$$

where X is the matrix representation of the NOT gate given in (1.16).

The NOT and V gates can be considered special cases of (1.20) for $\theta = \pi$ and $\theta = \pi/2$, respectively. However, the careful reader will notice a small discrepancy: when these values of θ are substituted into (1.20), the resulting matrices do not match (1.16) and (1.18) exactly, but rather are

$$R_x(\pi) = \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix} \tag{1.21}$$

and

$$R_x\left(\frac{\pi}{2}\right) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -i \\ -i & 1 \end{bmatrix}. \tag{1.22}$$

Although these matrices appear different from the ones given in (1.16) and (1.18), closer examination shows that the differences are only constant multiplicative factors, which amount to introducing extra global phases when these gates act on single qubits. Since global phase factors are physically irrelevant, the discrepancy can be ignored if we are considering a single-qubit system. However, in the context of a multi-qubit system, the apparent global phase factor can become a relative phase factor, as will be explained in Section 1.3.9. Therefore, I introduce rotation gates with an additional phase-correction factor, defined by

$$\tilde{R}_x(\theta) = e^{i\theta/2}\,R_x(\theta). \tag{1.23}$$

The expression given in (1.23) then has the property that $\tilde{R}_x(\pi) = X$ and $\tilde{R}_x(\pi/2) = V$ exactly. We also have $\tilde{R}_x(2\pi) = I$ and $\tilde{R}_x(-\pi/2) = V^\dagger$, the latter corresponding to the fact

that the $V^\dagger$ acts by rotating the Bloch sphere by 90 degrees around the x-axis in the opposite direction to the V gate.

Using (1.23), for any positive integer n we can define an n-th root-of-NOT gate as the instance of $\tilde{R}_x(\theta)$ where $\theta = \pi/n$. In Chapters 2 through 4, I will in particular use n-th root-of-NOT gates where n is a power of two. It will be convenient to have a compact, specialized notation for such gates, so I will use the notation X_k as a synonym for $\tilde{R}_x(\pi/2^k)$. In other words, X_k will denote a 2^k-th root-of-NOT gate. The p-th power of this gate, X_k^p, is then a $\tilde{R}_x(\pi p/2^k)$ gate.

To give a few examples of the notation just introduced in the previous paragraph, a X_0 gate is the same as an inverter, a X_1 gate is the same as a V gate, and a X_2 gate would be a fourth-root-of-NOT gate, with $X_2^4 = X$. A $V^\dagger$ gate can be represented as either $-X_1$ or $3X_1$, since, as Figure 1.3 shows, it can be treated as either a $-90°$ or $270°$ rotation (in radians, $-\pi/2$ or $3\pi/2$), where a counterclockwise angle in this figure is considered to be positive.

Root-of-NOT gates were previously used for quantum circuit design by Szyprowski and Kerntopf [14]. The X_k gates defined here are identical to what in Szyprowski and Kerntopf's notation would be denoted as R_{2^k}.

1.2.6 The controlled-NOT gate

The controlled-NOT gate, also known as the Feynman gate or CNOT gate for short, is a two-qubit gate. One of the qubits is designated the *control* qubit and the other, the *target* qubit. The gate operates by inverting the target qubit when the control qubit is in the state $|1\rangle$, while the target qubit remains unchanged if the control qubit is in the state $|0\rangle$. Since the controlled-NOT gate acts on the joint state of two qubits, it is represented by a 4×4 matrix,

which is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{1.24}$$

The controlled-NOT gate may be thought of as computing the exclusive-OR of its two inputs. Specifically, if the control and target qubits of the gate begin in the states $|a\rangle$ and $|b\rangle$, respectively, where $a, b \in \{0, 1\}$, then the outputs from the gate are $|a\rangle$ and $|a \oplus b\rangle$, respectively, where $\oplus$ denotes the exclusive-OR operation of Boolean algebra. (The case where one or both qubits are not in basis states is discussed in Section 1.3.9.) It is easy to verify that this is equivalent to the previously given definition in which the the target qubit is inverted if the control qubit is in the state $|1\rangle$.

The controlled-NOT gate is denoted by the schematic symbol shown in Figure 1.4. In this symbol, the small black dot identifies the control qubit while the other qubit is the target qubit.

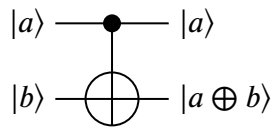


Figure 1.4: The controlled-NOT gate.

1.2.7 The controlled-NOT gate with negative-polarity control

A controlled-NOT gate may also be set up with a "negative-polarity control", or simply "negative control" for short: the target qubit is inverted when the control qubit's state is $|0\rangle$, instead of when it is $|1\rangle$. This variant of the controlled-NOT gate is represented by the

matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{1.25}$$

A negative control may be indicated in a schematic diagram by replacing the filled dot of the standard controlled-NOT gate with an empty dot, as shown in Figure 1.5. If needed for disambiguation, the ordinary controlled-NOT gate can also be called a "positive-polarity control" or "positive-control" gate.

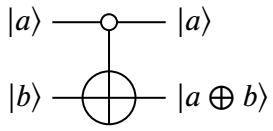


Figure 1.5: The negative-control Feynman gate.

Both positive- and negative-control variants of the controlled-NOT gate may additionally appear in a "target-up" configuration as shown in Figure 1.6. These are the same gates as the ones shown in Figures 1.4 and 1.5, with the only difference being that the target qubit is now considered the more-significant qubit, resulting in a different ordering of basis states and therefore a different matrix representation. The matrix representation of the target-up positive-control CNOT gate is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{1.26}$$

and that of the target-up negative-control CNOT gate is

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{1.27}$$

(a) The positive-control variant. **(b)** The negative-control variant.

Figure 1.6: "Target-up" configurations of controlled-NOT gates.

1.2.8 The Toffoli and multiple-control Toffoli gates

The Toffoli gate is a three-qubit gate. Two of the qubits are designated as control qubits and the remaining one is designated the target qubit. The Toffoli gate inverts the target qubit only if both control qubits are in the state $|1\rangle$. It can be represented by the matrix

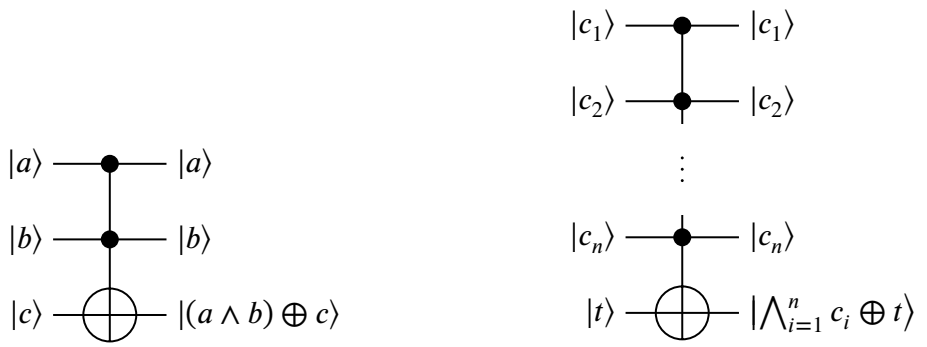
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}. \tag{1.28}$$

The Toffoli gate is denoted by the schematic symbol shown in Figure 1.7a, where the control qubits are indicated by the two small dots.

Like all the variants of the controlled-NOT gate, the Toffoli gate's action can also be expressed using Boolean algebra. If the control and target qubits all begin in basis states, with values $|a\rangle$, $|b\rangle$ for the control qubits and $|c\rangle$ for the target qubit, then the state of the target qubit becomes $|(a \wedge b) \oplus c\rangle$ after operation of the gate.[1] Figure 1.7a shows this interpretation of the Toffoli gate's operation in terms of Boolean algebra.

The multiple-control Toffoli gate is a generalization of the Toffoli gate for a variable

[1] As with the controlled-NOT gate, we assume for now that all qubits begin in basis states.



(a) The Toffoli gate.

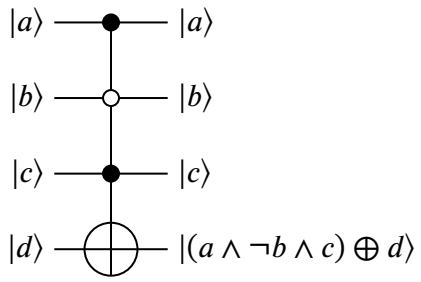
(b) A multiple-control Toffoli gate.

Figure 1.7: Toffoli and multiple-control Toffoli gates.

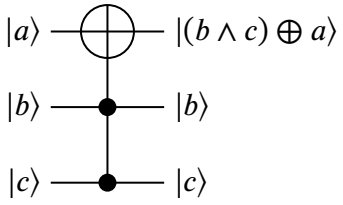
number of qubits. Strictly speaking, it is not a single gate but rather a family of gates, all of which operate conceptually in the same manner. All but one of the input qubits to a multiple-control Toffoli gate are designated as control qubits, while the remaining one is designated as the target. The gate inverts the target qubit only if all control qubits are in the state $|1\rangle$. The multiple-control Toffoli gate is denoted by the schematic symbol shown in Figure 1.7b. Its action has a representation in Boolean algebra analogous to that of the Toffoli gate: if the control qubits begin in states $|c_1\rangle$ through $|c_n\rangle$, and the target qubit begins in state $|t\rangle$, then the gate changes the target qubit's state to $\left|\bigwedge_{i=1}^{n} c_i \oplus t\right\rangle$, as depicted in Figure 1.7b.

As with the Feynman gate, the Toffoli and multiple-control Toffoli gates may have any of their control inputs modified to be negative controls that are "active" when the corresponding qubit's state is $|0\rangle$ rather than $|1\rangle$. For instance, the gate shown in Figure 1.8a will invert the target qubit only if the first, second, and third control qubits are in the states $|1\rangle$, $|0\rangle$, and $|1\rangle$, respectively. In terms of Boolean algebra, this corresponds to taking the exclusive-OR of the target qubit with $a \wedge \neg b \wedge c$, where $|a\rangle$, $|b\rangle$, and $|c\rangle$ are the states of the control qubits. Just as the Feynman gate may appear as a "target-up" variant, the Toffoli and multiple-control

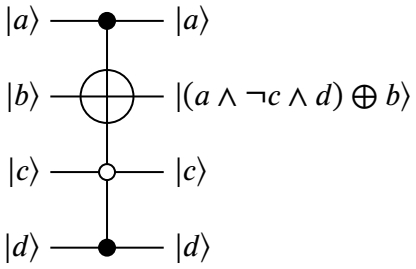
Toffoli gates may also appear with any of the qubits in a circuit being the target qubit, and any such variant may further be modified to have negative control polarities. Figures 1.8b and 1.8c display examples of these variant configurations of Toffoli and multiple-control Toffoli gates.



(a) A multiple-control Toffoli gate with a negative-polarity control.



(b) A Toffoli gate in "target-up" configuration.



(c) A multiple-control Toffoli gate with target qubit in the middle and a negative-polarity control.

Figure 1.8: Toffoli and multiple-control Toffoli gates with varying control polarities and target qubits.

The Toffoli and multiple-control Toffoli gates are often used to compute the logical AND of any number of inputs. In Figure 1.7b, if we let $t = 0$, meaning that the target qubit is initialized to the state $|0\rangle$, then the final state of this qubit will simply be $\left|\bigwedge_{i=1}^{n} c_i\right\rangle$, the logical AND of the input values c_1 through c_n. These same gates can also be used to compute a logical OR by taking advantage of duality in Boolean algebra: since $\bigvee_{i=1}^{n} c_i = \neg\left(\bigwedge_{i=1}^{n} \neg c_i\right)$, a Toffoli or multiple-control Toffoli gate with all-negative control polarities and with the

target qubit initialized to $|1\rangle$, as shown in Figure 1.9, computes the logical OR of its control inputs. Specifically, the final state of the target qubit in Figure 1.9 is seen to be

$$\left(\bigwedge_{i=1}^{n} \neg c_i\right) \oplus 1 = \neg\left(\bigwedge_{i=1}^{n} \neg c_i\right) = \bigvee_{i=1}^{n} c_i. \tag{1.29}$$

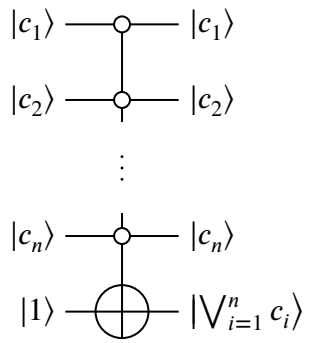


Figure 1.9: A multiple-control Toffoli gate computing the logical OR of n variables.

The ability of the Toffoli and multiple-control Toffoli gates to compute logical ANDs and ORs means that they provide a simple path to computational universality, since any Boolean function may be computed using logical NOT, AND, and OR operations. However, as Section 1.3.5 explains, this straightforward approach does not always perform very well in practice, because computing Boolean functions using AND and OR operations implemented by Toffoli gates requires additional qubits to be added to the circuit, increasing its size. In Chapters 2 through 4 I propose an alternative path to universality which does not rely on multiple-control Toffoli gates at all but instead entails realizing functions directly using two-qubit gates.

Since the multiple-control Toffoli gates are simply extensions of the Toffoli gate with more control qubits, it is often convenient to refer to them both as simply "Toffoli gates", and I will often do so. If there is a need for disambiguation, the number of control qubits or the

total number of qubits (including the target qubit) can be specified, *e.g.*, a 3-control Toffoli gate or a 4-qubit Toffoli gate. The Toffoli gate shown in Figure 1.7a is then a 2-control or 3-qubit Toffoli gate. Both the NOT and CNOT gates can also be considered degenerate Toffoli gates: the NOT gate is a 0-control Toffoli while the CNOT is a 1-control Toffoli.

1.2.9 Generic controlled gates

The controlled-NOT, Toffoli, and multiple-control Toffoli gates are clearly closely related. Both the controlled-NOT and Toffoli gates can be considered special cases of the multiple-control Toffoli gate with one and two control qubits, respectively. Conversely, the Toffoli and multiple-control Toffoli gates can be thought of as extensions of the controlled-NOT gate to larger numbers of qubits.

Controlled gates are a further extension of the type of operation performed by the controlled-NOT Toffoli, and multiple-control Toffoli gates. For an arbitrary single-qubit gate U, which I will call the *target gate*, a controlled-U gate is defined as a two-qubit gate, acting on one control qubit and one target qubit, whose behavior is analogous to that of the controlled-NOT gate. If the control qubit is in the state $|1\rangle$, the gate U is applied to the target qubit, whereas if the control qubit is in the state $|0\rangle$, the target qubit is unaffected. Such a gate is depicted in Figure 1.10a.

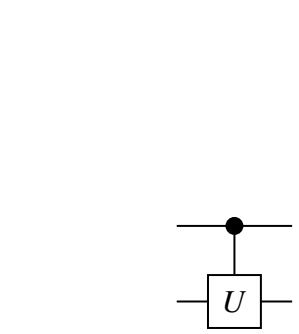
With a controlled-U gate being analogous to a controlled-NOT gate, a multiple-control U gate is then similarly defined by analogy with the Toffoli and multiple-control Toffoli gates. Specifically, a multiple-control U gate operates by applying the U gate to the target qubit only if all control qubits are in the state $|1\rangle$. Its schematic symbol is shown in Figure 1.10b. It is also possible to define controlled-U and multiple-control U gates with negative control polarities.

Multiple-control U gates may be yet further generalized by allowing an arbitrary Boolean function to determine whether the U gate is applied to the target qubit, instead of requiring that all control qubits be in the state $|1\rangle$. I call such a gate an f-controlled U gate, where f is the aforementioned Boolean function that determines whether the U gate is applied. In other words, the f-controlled U gate operates by applying the U gate to the target qubit if and only if f evaluates to 1 with the control qubits' states as its inputs. This form of controlled gate is illustrated in Figure 1.10c.

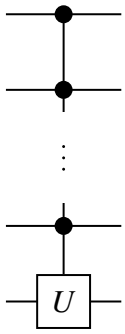
Finally, we may also relax the condition that U be a single-qubit gate, thus allowing for f-controlled U gates where U acts on multiple qubits. For such a gate, all the qubits acted on by U are designated as target qubits, and the U gate is only actually applied if f evaluates to 1, just as before. This, the most general type of controlled gate, is shown in Figure 1.10d.

I refer to a controlled gate (of any of the previously described types) as "active" when the control qubits' states are such that the target gate is applied, and "inactive" otherwise. This terminology has already been used several times without comment. For instance, a positive-control CNOT gate is active when its control qubit has state $|1\rangle$ and inactive when its control qubit has state $|0\rangle$, while the reverse is true for the negative-control CNOT gate.

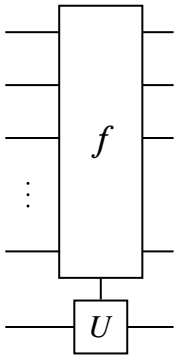
There is one special case of f-controlled-U gate that is of particular interest to us, namely, the case where U is an inverter. An f-controlled inverter provides a method by which a single-output function f can be realized in a reversible fashion. Specifically, if one lets U be an inverter and initializes the target qubit to $|0\rangle$ in Figure 1.10c, then the final state of the target qubit will be $|f(x_1, \ldots, x_n)\rangle$, as shown in Figure 1.11. In this respect, the f-controlled inverter is analogous to the Toffoli gate, which provides a method of realizing the logical AND operation of Boolean algebra in a reversible fashion. Gates of this type are also crucial to the operation of Grover's algorithm, which is discussed in Section 1.4. Realization of



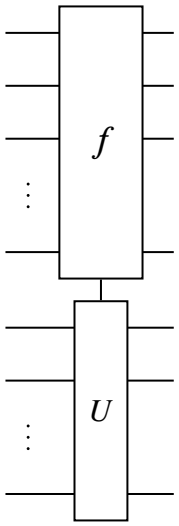
(a) A controlled-U gate.



(b) A multiple-control U gate.



(c) An f-controlled U gate.



(d) An f-controlled U gate where U acts on multiple qubits.

Figure 1.10: Controlled gates with increasing generality.

Boolean functions using quantum circuits is discussed further in Section 1.3.4.

1.2.10 Multiple-valued gates

All of the gates introduced thus far operate only on qubits. Gates that operate on qudits function similarly, but they take on a much larger number of variants due to the additional available states. For instance, one can define an "inverter-like" gate that operates on a qutrit by exchanging the $|0\rangle$ and $|1\rangle$ components of its state while leaving the $|2\rangle$ component

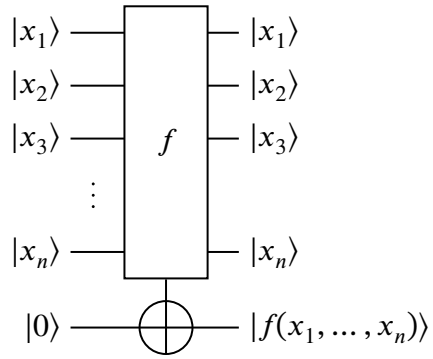


Figure 1.11: An f-controlled inverter, which provides a reversible realization of the otherwise non-reversible function f.

unchanged. This gate is represented by the following matrix:

$$T_{01} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.30}$$

However, two other "inverter-like" gates are also possible: the gate could instead exchange the $|0\rangle$ and $|2\rangle$ components of the input state while leaving the $|1\rangle$ component unchanged, or it could exchange the $|1\rangle$ and $|2\rangle$ components and leave the $|0\rangle$ component unchanged. These two gates are represented by the following matrices:

$$T_{02} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{1.31}$$

$$T_{12} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{1.32}$$

For a qudit with more than three basis states, even more "inverter-like" gates are possible: for every pair of basis states, one can define a gate that exchanges the components corresponding to those two basis states while leaving all other components unchanged. Therefore,

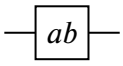
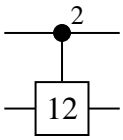


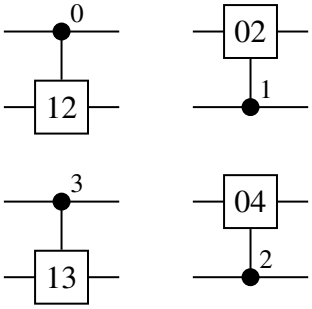
Figure 1.12: A transpositional gate T_{ab}.

with four basis states, six "inverter-like" gates are possible, with five basis states, ten are possible, and in general, with n basis states, ${}_nC_2 = n(n-1)/2$ "inverter-like" gates are possible. There does not appear to be a standard name for these gates, but I will refer to them as *transpositional* gates from now on since their operation can be described as a transposition of two of the basis states of a qudit. In general, I will use the notation T_{ab}, which was previously used without comment, to represent a transposition gate that exchanges the states $|a\rangle$ and $|b\rangle$. However, in circuit schematics, I will simply use ab to represent a T_{ab} gate, as shown in Figure 1.12. This notation has the useful property of being radix-agnostic. In other words, a symbol such as T_{02} is meaningful for any qudit with radix $n \geq 3$, regardless of the precise value of n, and represents a gate that exchanges the $|0\rangle$ and $|2\rangle$ states of the qudit. Strictly speaking, a T_{02} gate for a qutrit and a T_{02} gate for a ququart (qudit of radix 4) are two different gates, because one is represented by a 3×3 and the other by a 4×4 matrix. However, it will prove convenient to think of these gates as conceptually the "same" operation because it will be helpful in Chapter 6 for designing families of circuits that can be easily adapted for qudits of arbitrary radix. This way of thinking is reinforced by the T_{ab} notation, as well as by our convention (first introduced at the end of 1.1.6) of labeling the computational basis states of an n-valued qudit as $|0\rangle$ through $|n-1\rangle$.

The concept of controlled gates also generalizes to qudits in a straightforward manner. One can define an analog of the controlled-NOT gate that operates on qutrits as follows: if the control qutrit is in the state $|2\rangle$, then the transpositional gate T_{12} is applied to the target qutrit; otherwise, the target qutrit's state remains unchanged. Figure 1.13a shows the schematic diagram used to represent such a *controlled-transposition* gate.



(a) A controlled-transposition gate operating on two qutrits.



(b) Some other possible controlled-transposition gates.

Figure 1.13: Examples of controlled-transposition gates.

Controlled-transposition gates may have different control types similar to the positive- and negative-control variants of the CNOT gate. However, instead of having positive- and negative-control variants, controlled-transposition gates operating on qudits can be set up with a variety of *control values* and *target gates*. The control value defines which state of the control qudit is required to activate the gate, while the target gate represents the operation to be performed on the target qudit when the control qudit's state matches the control value. Like the CNOT gate, target-up variants of controlled-transposition gates also exist. Figure 1.13b shows some of these possible variants. Since there are n possible control values for a control qubit of radix n, the control value is simply written as a numeral beside the control dot in the schematic symbol, rather than using filled and empty dots as in the binary case.

Like transpositional gates, controlled-transposition gates can operate on qudits of any radix that is high enough for the basis states that the gate operates on to exist. The control

and target qudits can even be of different radices. For instance, the gates depicted in the upper row of Figure 1.13b have control values of 0 and 1, but their target gates both involve the basis state $|2\rangle$. Therefore, for both gates, the target qudit must have radix at least 3 but the control qudit only needs to have radix 2. Both gates can operate with a qubit as the control input and a qutrit as the target input, although they can also operate on two qutrits, or on qudits of any higher radices. The gate in the lower left of Figure 1.13b requires qudits with a radix of at least 4 for both its control and target inputs, since it has a control value of 3 and the target gate also involves $|3\rangle$. The last gate in the lower right of Figure 1.13b requires a target qudit of radix at least 5, but its control qudit may be of a radix as low as 3.

In Section 1.2.9 it was mentioned that Toffoli and multiple-control Toffoli gates may be thought of as extensions of the controlled-NOT gate. In a similar way, controlled-transposition gates may be extended with additional control qudits to create *multiple-control transpositional* gates. Such a gate acts on any number of control qudits together with one target qudit, and is described by specifying a control value for each control qudit together with a target gate. A multiple-control transpositional gate thus described is acts by applying its target gate to the target qudit if and only if all control qudits' states match their corresponding control values. For instance, the leftmost gate shown in Figure 1.14 applies a T_{01} gate to the target qudit if and only if the control qudits' states are $|1\rangle$ and $|2\rangle$, respectively, and the middle gate applies a T_{02} gate to the target qudit if and only if the three control qudits' states are $|0\rangle$, $|2\rangle$, and $|1\rangle$, in that order. The third gate shown in Figure 1.14 targets a qudit other than the bottommost one in the circuit. Like controlled-transposition gates, multiple-control transpositional gates can operate on qudits of mixed radices, and they are subject to the same minimum-radix requirements as controlled-transposition gates. For instance, the first gate shown in Figure 1.14 requires qudits with a minimum radix of 2, 3, and 2, in order from top

to bottom.

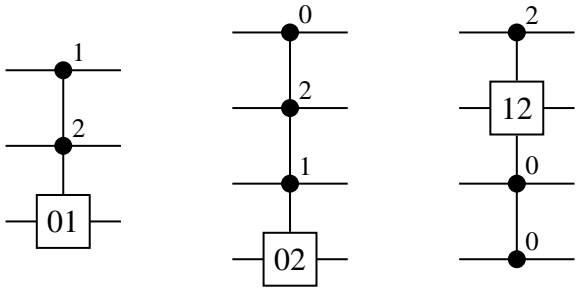


Figure 1.14: Examples of multiple-control transpositional gates.

1.3 Quantum circuits

1.3.1 The structure of quantum circuits

A *quantum circuit* consists of a sequence of quantum gates designed to perform some computational task. Quantum circuits are represented in schematic form by using horizontal lines to connect the output of one quantum gate to the next. These horizontal lines then represent the qubits or qudits that participate in the circuit's operation. Figure 1.15 illustrates the typical structure of a quantum circuit and its appearance in schematic form.

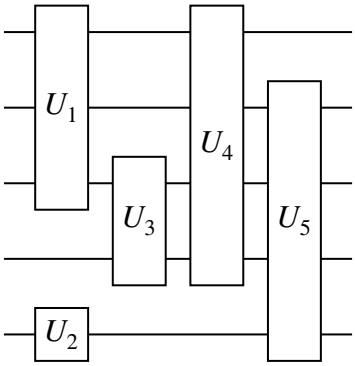


Figure 1.15: Structure of a typical quantum circuit.

A binary quantum circuit operates in the following manner. First, the qubits participating

in the circuit are initialized to some predetermined values, usually $|0\rangle$ or $|1\rangle$. Then, the gates making up the circuit each act on their corresponding qubits. The state of the qubits after all gates have acted is considered to be the output of the circuit.

Quantum circuits have a more rigid and fixed structure compared with classical digital logic circuits. Specifically, since all quantum gates are reversible, each gate must have the same number of inputs as outputs. Quantum gates do not receive input signals and then produce new output signals in the same way that digital logic gates do. Instead, the gates act on the qubits "in place", merely altering their states without changing their number. I elaborate more on this point in Section 1.3.6. A qubit is a discrete physical entity and cannot simply disappear from a circuit, although it may be unused after a certain point. Similarly, a qubit cannot split into two—in other words, no fan-out is allowed in a quantum circuit. The prohibition on fan-out is due to a well-known theorem, the no-cloning theorem, which states that it is impossible to perfectly duplicate an arbitrary, unknown quantum state [13, §1.3.5]. This is why, in a schematic diagram for a quantum circuit, the horizontal lines representing the qubits always extend all the way from the beginning to the end of the circuit and never split or join.

1.3.2 Analysis of quantum circuits using matrix multiplication

Since the action of a single gate is represented by matrix-vector multiplication, the combined action of two gates when one operates after the other is represented by the matrix product of the individual gates' matrix representations. For instance, consider a simple single-qubit quantum circuit consisting of a Hadamard gate followed by an inverter, as shown in Figure 1.16. The matrix representation of the circuit's operation is then obtained by

Figure 1.16: A single-qubit circuit containing two gates.

multiplication of the matrix representations of the inverter and Hadamard gate:

$$U_{\text{circuit}} = X \cdot H = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}. \tag{1.33}$$

In (1.33), the Hadamard gate's matrix appears second in the matrix product even though it is first to act in the circuit. This ordering of the matrices is due to the fact that a matrix acts on a vector to its right via matrix-vector multiplication, so that when a product of two matrices acts on a vector, the rightmost matrix acts first. In other words, if we explicitly write out the action of the circuit from Figure 1.16 on an arbitrary state represented as a vector,

$$U_{\text{circuit}}\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{1.34}$$

we see that placing the matrix of the Hadamard gate to the right does indeed correctly represent that gate acting first on the state $\begin{bmatrix} \alpha & \beta \end{bmatrix}^{\mathsf{T}}$.

Just as the joint state of a two-qubit system is given by the tensor product of the individual qubits' states, the matrix representation of two gates acting in parallel on the individual qubits of a two-qubit system is given by the tensor product of the individual gates' matrix representations. For instance, Figure 1.17 shows a Hadamard gate and an inverter acting on the first and second qubits, respectively, of a two-qubit system. The action of this circuit is

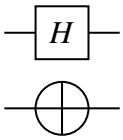


Figure 1.17: A quantum circuit containing two gates acting in parallel.

then represented by the tensor product

$$U_{\text{circuit}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$
$$= \frac{1}{\sqrt{2}}\begin{bmatrix} 1\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} & 1\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \\ 1\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} & -1\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}0 & 1 & 0 & 1\\ 1 & 0 & 1 & 0\\ 0 & 1 & 0 & -1\\ 1 & 0 & -1 & 0\end{bmatrix}. \quad (1.35)$$

The tensor-product representation of the action of two gates in parallel can also be used to derive the matrix representation of a gate when it acts on only a subset of the qubits in a quantum circuit. This is accomplished by inserting identity gates in parallel with the gate of interest, and then analyzing the parallel combination as explained above. For instance, if the Hadamard gate acts on the first qubit of a two-qubit system, as shown in Figure 1.18a, then we can insert an identity gate acting on the second qubit, as shown in Figure 1.18b, in order to treat the circuit as a combination of two gates acting in parallel. We can then use a tensor product to compute the matrix representation of the circuit:

$$U_{\text{circuit}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{bmatrix}. \quad (1.36)$$

We can now derive the matrix representation of any quantum circuit using a combination of matrix and tensor products. For instance, consider the circuit shown in Figure 1.19. The

(a) A quantum gate acting on one out of two qubits. **(b)** Insertion of an identity gate.

Figure 1.18: Analysis of a gate acting on a subset of the qubits in a circuit.

Hadamard gate in this circuit is analyzed as before—that is, as a combination of Hadamard gate acting on the first qubit with identity gate acting on the second qubit—and the matrix representation of the whole circuit is then derived by multiplication of the result with the matrix of the controlled-NOT gate:

$$U_{\text{circuit}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix}. \tag{1.37}$$

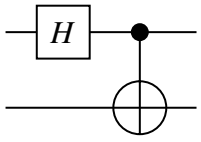


Figure 1.19: A two-qubit circuit to be analyzed using matrix multiplication and tensor products.

While the operation of any quantum circuit can in principle be analyzed using matrix representations, a binary quantum circuit containing n qubits requires each gate to be represented using a $2^n \times 2^n$ matrix. Naive attempts at analyzing quantum circuits entirely in terms of matrices therefore quickly become computationally intractable as the size of the circuit increases. Fortunately, many useful circuits can be analyzed using only Boolean algebra and without using matrix representations at all, as explained in the next subsection. In Chapters 2 through 4, I also demonstrate the analysis of certain carefully-designed circuits using symmetric functions.

1.3.3 Analysis of binary permutative quantum circuits

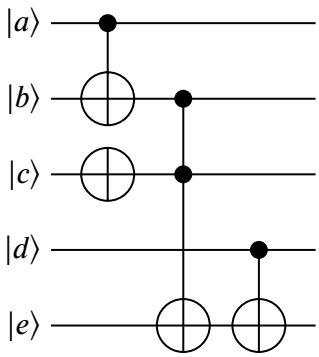
If a binary quantum circuit contains only NOT, CNOT, and (multiple-control) Toffoli gates, it can be analyzed using Boolean algebra, taking advantage of the Boolean-algebraic descriptions of those gates introduced in Sections 1.2.6 through 1.2.8. As an example, Figure 1.20 demonstrates the analysis of such a circuit. As a consequence of the ability to express the outputs entirely using Boolean algebra, if every qubit is initialized to either $|0\rangle$ or $|1\rangle$, then their final state must also consist of only $|0\rangle$ and $|1\rangle$. In other words, circuits of this type have the property that when the input is a basis state, the output will also be a basis state, meaning that their matrix representations are always permutation matrices. Accordingly, I will refer to such quantum circuits as *permutative* circuits.

1.3.4 Realization of Boolean functions using quantum circuits

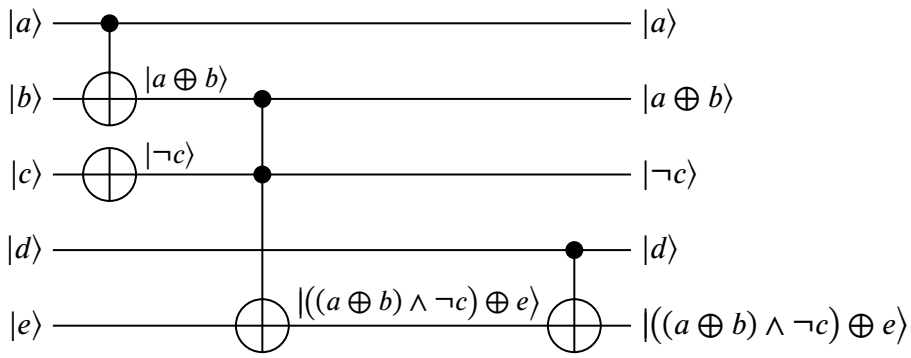
In classical digital logic design, one of the most fundamental and common tasks to be performed is to realize a desired Boolean function. *Realization* of a Boolean function means designing a digital logic circuit that will compute the output or outputs of the function from its inputs. This usually takes the form of a circuit in which the inputs are transmitted into the circuit via a collection of input terminals and the outputs are similarly received via a collection of output terminals, as shown in Figure 1.21a.

Although computing using multiple-valued logic remains mostly experimental, a function with multiple-valued rather than Boolean inputs and outputs can in theory be realized by a classical circuit in an analogous manner to a Boolean function: one needs to design a multiple-valued circuit that will accept a collection of input signals and produce the correct output signals, again as in Figure 1.21a.

In order to execute quantum algorithms such as Grover's algorithm (discussed in Sec-



(a) A quantum circuit containing only NOT, CNOT, and Toffoli gates.



(b) The same circuit with intermediate and final states labeled.

Figure 1.20: Analysis of a quantum circuit using Boolean algebra

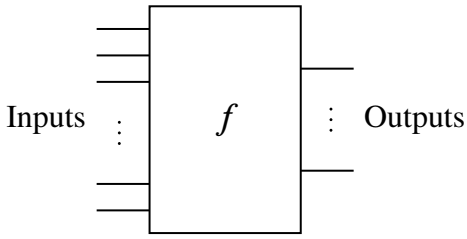
tion 1.4), one requires the capability to realize Boolean functions using quantum circuits. As previously discussed, quantum circuits are subject to restrictions that classical circuits are not: all quantum gates must be reversible, and no fan-out is allowed. Because of these restrictions, one cannot in general realize an arbitrary Boolean function using a quantum circuit in the same way that one does using a classical circuit. The type of circuit depicted in Figure 1.21a is usually not possible in a quantum setting, because the circuit will not be reversible if the numbers of inputs and outputs are unequal. Even if the numbers of inputs and outputs are equal, reversibility of the function is not guaranteed.

The problem of realizing Boolean functions therefore comprises two distinct problems

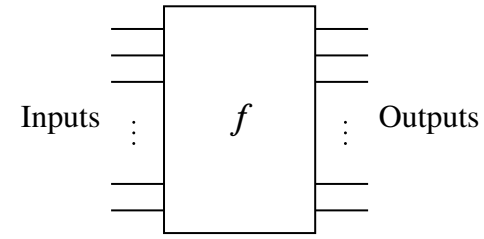
in the quantum setting: one of realizing reversible Boolean functions, and one of realizing non-reversible Boolean functions. If a Boolean function is reversible, then it can in theory be realized using a quantum circuit in a manner similar to that of Figure 1.21a, although the reversibility requirement constrains the numbers of inputs and outputs to be equal, as shown in Figure 1.21b. If a Boolean function is not reversible, it must instead be transformed into a reversible form using a construction like that depicted in Figure 1.21c. In this figure, we see that the inputs to the function are supplied through one collection of qubits while the outputs are retrieved by supplying a second collection of qubits, where the circuit inverts the state of an output qubit to indicate an output of 1 from the Boolean function. The resemblance of the notations used in Figures 1.21c and 1.11 is intentional: under the model of Boolean function realization described here, an inverter controlled by a single-output Boolean function f can be thought of as a realization of f, and indeed is a special case of Figure 1.21c where the function f has only one output. The use of multiple "control bars" (the vertical lines connecting f to the inverters) in Figure 1.21c visually indicates that the circuit can independently invert each of the output qubits depending on the value of the corresponding output of the Boolean function f that it realizes.

The particular case of realizing a single-output Boolean function, as represented by 1.11, is important enough to warrant its own discussion. Using a so-called exclusive-OR sum-of-products (ESOP), a Boolean function may be expressed in terms of AND and exclusive-OR operations, which are then readily implemented using multiple-control Toffoli gates. Algorithms that have been developed for generating ESOP expressions include EXMIN2 [15] and EXORCISM [16, 17].

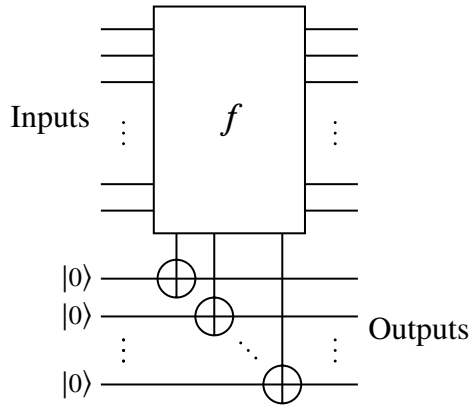
As an example of the realization of a Boolean function using an ESOP expression, consider the function of four variables given as a Karnaugh map in Figure 1.22a. This



(a) Classical realization.



(b) Quantum realization if f is reversible.



(c) Quantum realization if f is non-reversible.

Figure 1.21: Three different types of realizations for a Boolean function f.

function may be expressed in ESOP form as

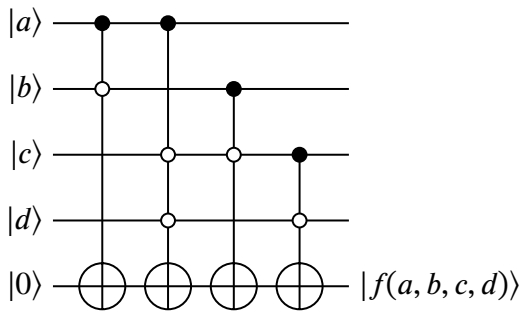
$$f(a,b,c,d) = (a \wedge \neg b) \oplus (a \wedge \neg c \wedge \neg d) \oplus (b \wedge \neg c) \oplus (c \wedge \neg d), \tag{1.38}$$

which leads to the circuit shown in Figure 1.22b, where each term consisting of the logical AND of two or more variables is realized using a multiple-control Toffoli gate.

In this dissertation I consider the realization of both reversible and non-reversible Boolean functions. Realization of non-reversible Boolean functions, especially single-output functions, is covered in Chapters 2 through 4, while realization of reversible Boolean functions is covered in Chapter 5.

ab \ cd	00	01	11	10
00	0	0	0	1
01	1	1	0	1
11	0	1	0	1
10	0	1	1	0

(a) A Karnaugh map representing a Boolean function of four variables.



(b) Realization of the function using multiple-control Toffoli gates.

Figure 1.22: Realization of a single-output Boolean function using an ESOP expression.

1.3.5 Ancillary qubits and mirror circuits

Any Boolean function is in principle realizable using the just-described procedure of expressing the function in ESOP form and translating the terms to multiple-control Toffoli gates. However, it is often not straightforward, or even outright computationally intractable, to express a function in such a form. Boolean functions of interest are often not given explicitly as truth tables, but instead as complicated equations that are not easily transformed to ESOP form. In some cases, a function of interest is only indirectly defined through a set of constraints and an equation must be derived using these constraints. If the function is of a large number of variables, it will be completely impractical to compute a complete truth table for the function since the number of entries in a truth table is given by 2^n where n is the number of variables of the function. Other realization methods must therefore be used in such cases.

When a function is given as a Boolean-algebraic expression (in any form), it may be realized by translating each term in the expression to an appropriate quantum gate. This however usually requires intermediate results to be stored somewhere while the circuit is

computing the function. Additional *ancillary* qubits (often shortened to "ancilla", and also known as *work* qubits) must therefore be added to the circuit for this purpose. For example, consider the function

$$y = f(a, b, c, d) = (a \vee b) \wedge (c \vee d). \tag{1.39}$$

This function can be realized by using Toffoli gates to compute the two logical OR terms and then using a third Toffoli gate to compute the logical AND of the two intermediate results, as shown in Figure 1.23. Two ancillary qubits are used to store the intermediate results $a \vee b$ and $c \vee d$.

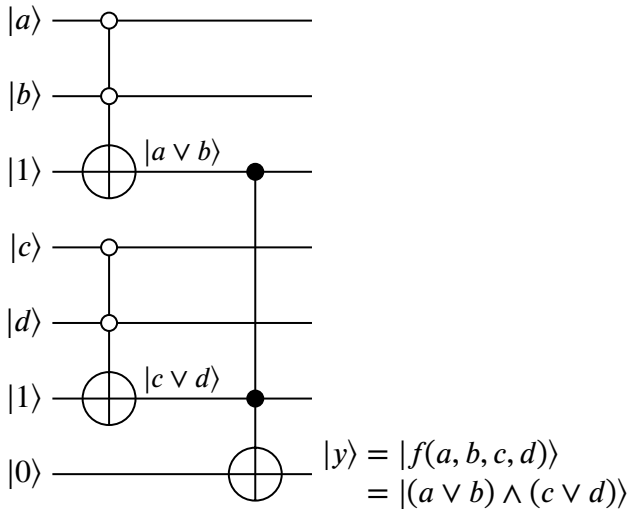
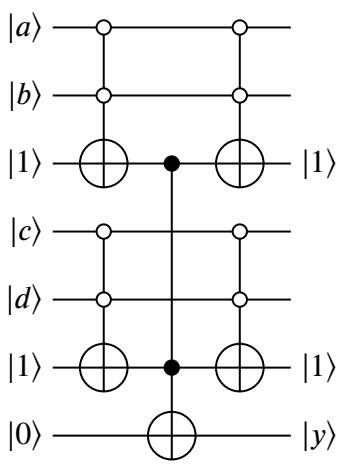


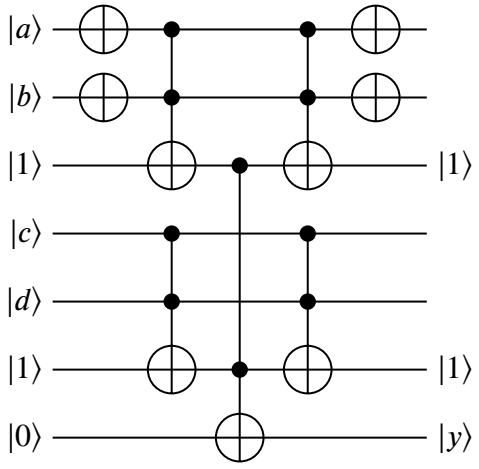
Figure 1.23: Realization of a Boolean function using ancillary or work qubits.

In many quantum circuit design scenarios, including (as will be described in Section 1.4) the realization of Boolean functions for use as oracles in Grover's algorithm, there is a requirement that all qubits in the circuit other than the final result-carrying qubit must be restored to their original values, so that they can be reused later on. This restoration can be accomplished using a *mirror circuit*. A mirror circuit consists of inverses of all the gates in the circuit that do not directly modify the state of the result qubit, applied in the reverse of

their original order. Figure 1.24a illustrates a mirror circuit for the circuit from Figure 1.23. The negative controls of the Toffoli gates in this circuit can also be replaced with sequences of inverters followed by positive-control Toffoli gates to better illustrate the reversed gate order of a mirror circuit, as in Figure 1.24a. Specifically, a Toffoli gate with negative controls can be emulated by first applying an inverter to each control qubit with a negative control polarity, and then using a Toffoli gate with all-positive control polarities. In the mirror circuit, the Toffoli gate (which is its own inverse) is applied first and then followed by the inverters, which restore the control qubits to their original states.



(a) The circuit from Figure 1.23 with a mirror circuit added.



(b) The same circuit with negative controls replaced by inverters.

Figure 1.24: Illustration of mirror circuits used to restore ancillary qubits to their original states.

One particular use of an ancillary qubit is to implement an f-controlled U gate (introduced in Section 1.2.9) for an arbitrary gate U, assuming that a realization of the function f as an f-controlled inverter is available. This is done using the circuit shown in Figure 1.25. The ancillary qubit is initialized to $|0\rangle$, and if the function f evaluates to 1, then the ancillary qubit is inverted and attains the state $|1\rangle$. A controlled-U gate with the ancillary qubit as its control qubit then applies a U gate to the target qubit. If the function f evaluates to 0, the

ancillary qubit is not inverted and retains the state $|0\rangle$, and the controlled-U has no effect on the target qubit. We therefore see that the overall effect of the circuit is to apply a U gate to the target qubit when f evaluates to 1, which is exactly the behavior expected from an f-controlled U gate. The second f-controlled inverter in Figure 1.25 acts as a mirror circuit to restore the ancillary qubit to its initial state of $|0\rangle$.

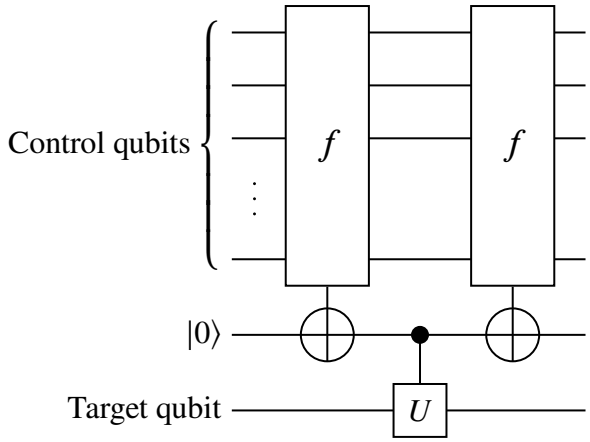


Figure 1.25: Implementation of an f-controlled U gate using an ancillary qubit.

1.3.6 Qubits as quantum memory

The use of ancillary qubits in circuits such as the ones shown in Figures 1.23 and 1.24 demonstrate an important attribute of qubits in a quantum circuit—namely, that they act as quantum *memory* rather than "wires", as the schematic diagrams for quantum circuits might suggest. While signals are conducted between gates using physical wires in a classical digital logic circuit, this is not the case for a quantum circuit. The qubits in a quantum circuit are physical components (possibly microscopic, for instance only single atoms or particles) that have the ability to maintain a quantum state, and the horizontal axis in a quantum circuit schematic diagram represents time, not space, so each horizontal line in a schematic diagram represents a qubit's progression through time and not a physical wire.

For instance, in an ion-trap quantum computer as described in [13], the qubits are trapped ions whose energy states are used to represent the computational basis states $|0\rangle$ and $|1\rangle$. In a more recent device developed at IBM, as described by Gambetta *et al.* [18], the qubits are superconducting electronic circuits with multiple energy states and these energy states are also used to represent the computational basis states. The fact that quantum states are not transmitted through wires but are instead carried by the physical qubits themselves explains why ancillary qubits are necessary to store the intermediate results of a computation: without wires, the intermediate result can only be transmitted to the next stage of a computation by allocating an additional qubit to store it.

Just as qubits in a quantum circuit are not physical wires, quantum gates are also not components of the physical apparatus of a quantum computing system, in contrast to classical digital logic gates. Since the input to a quantum gate is transmitted by a set of qubits, and the output of the gate is also carried by those same qubits, a quantum gate is in fact a manipulation performed on the qubits by the physical apparatus, not the apparatus itself. For instance, in an ion-trap quantum computer, quantum gates are implemented by using carefully-modulated laser pulses directed at the ions to induce state changes, while in a computer using superconducting qubits, the gates can be implemented by exciting the superconducting circuits with microwave energy.

Since qubits are not physical wires and gates are not physical components, a quantum computing system must necessarily also contain a classical computer that controls the quantum hardware (*i.e.*, the physical quantum computing apparatus). Therefore, a quantum circuit might in fact better be described as a quantum *program*, because it is nothing more than a sequence of instructions for the aforesaid classical computer that is controlling the quantum hardware. I will continue to use the term "quantum circuit" because it is already

standard. However, the observation that a quantum circuit is more akin to a program and that gates are akin to individual instructions of that program is of great importance, because it means that a quantum circuit can easily be reconfigured on the fly in between executions of a quantum algorithm. One can thus speak of executing a quantum circuit, as this simply means executing the instructions corresponding to the gates that make up the quantum circuit. The ability to freely modify and reconfigure quantum circuits will be exploited in the quantum algorithms described in Chapters 7 and 8 to solve optimization problems using Grover's algorithm.

1.3.7 Quantum gates versus circuits

From a mathematical point of view, there is little distinction between a quantum gates an a quantum circuit. Both quantum gates and circuits act on sets of qubits to modify those qubits' states. As discussed in Section 1.3.2, the operation of a quantum circuit as a whole, just like that of a gate, can be described by a matrix. In this dissertation, I will use one word or the other to indicate a difference in conceptual attitude towards the gate or circuit being discussed. Quantum gates are conceptualized as performing simple, easily understood operations that are meant to be used as building blocks for more complicated behavior, while a circuit is a compound entity consisting of a number of these building blocks working together. I will speak of a quantum gate being *implemented* by a circuit: this means that the behavior of the gate in question is identical to that of the circuit, which consists of other quantum gates. A quantum gate may have multiple implementations by distinct circuits, since nothing prevents distinct circuits from having the same overall behavior. Therefore, another way of phrasing the distinction between gates and circuits is that a gate is defined only by its behavior, while a circuit is defined as a specific implementation of that behavior.

1.3.8 Quantum cost

The preceding discussion should not be taken to indicate that quantum gates must necessarily correspond to individual physical operations that are performed by a quantum computer. Indeed, work on on the physical realization of a quantum computing system often aims to realize only a bare-minimum set of operations that is sufficient for universality, with most quantum gates requiring multiple such operations to implement. When a quantum gate is implemented by a circuit, the gates making up that circuit themselves need to be implemented by sequences of multiple physical operations before a quantum computing system can actually execute the circuit. An implementation of a quantum gate by a circuit is therefore only useful if all of the gates contained in that circuit themselves have known implementations in terms of underlying physical operations.

The need to ultimately decompose a quantum circuit into primitive physical operations before execution leads to the notion of *quantum cost*. Quantum cost is a metric that attempts to compare execution times of quantum circuits relative to each other. A variety of quantum cost models are possible, but one of the most popular and widely used is Maslov and Dueck's definition, which takes the quantum cost of a circuit to be the number of single-qubit and two-qubit controlled gates in the circuit [19]. If the circuit contains other gates (such as Toffoli gates), they must be decomposed into sequences of single-qubit and two-qubit controlled gates in order to calculate the quantum cost of the circuit. A gate may have multiple implementations in terms of single-qubit and two-qubit controlled gates, in which case the quantum cost of a circuit containing that gate will depend on which implementation is chosen. It might then seem that the obvious course of action is to always choose the implementation of each gate that results in the lowest cost. However, a lower-cost implementation of a given quantum gate is usually made possible by the use of a larger number of ancillary

qubits. For instance, with no ancillary qubits, the implementation of an n-qubit Toffoli gate requires $\mathcal{O}(n^2)$ two-qubit controlled gates, while with one ancillary qubit, the cost drops to $\mathcal{O}(n)$ [20]. Therefore, the quantum cost of a circuit depends on the number of ancillary qubits assumed to be available for implementing Toffoli gates (and any other gates whose implementations may use ancillary qubits) within the circuit. Since qubits can be thought of as quantum memory, as explained in Section 1.3.6, the dependence of quantum cost on number of available ancillary qubits is a trade-off between space and time complexity, analogous trade-offs being found in nearly all practical hardware and software engineering problems.

Throughout this dissertation, I will use Maslov and Dueck's quantum cost as the metric of choice to evaluate run-time complexities of quantum circuit families and quantum algorithms. This effectively entails assuming that every two-qubit controlled gate takes approximately the same time to execute, which is reasonable since every such gate can be implemented using at most two CNOT and three single-qubit gates [20, Lemma 5.1].

1.3.9 Linearity of quantum circuits

The descriptions of quantum gates such as the controlled-NOT, Toffoli, and other controlled gates in Section 1.2 implicitly assume that all qubits begin in one of the states $|0\rangle$ or $|1\rangle$, since they are stated in terms of a gate performing one action if a qubit is in the state $|0\rangle$ and another action if that qubit is in the state $|1\rangle$. In addition, the CNOT and Toffoli gates are described as performing logical exclusive-OR and AND operations, which have no meaning for states other than $|0\rangle$ and $|1\rangle$.

For qubits in states other than $|0\rangle$ and $|1\rangle$, the previously unspecified behavior of controlled gates is determined by linearity. Since all quantum gates have a matrix representation,

in which the gate acts on an input quantum state via matrix-vector multiplication, and matrix-vector multiplication is linear, all quantum gates behave in a linear manner. This means that, for an input state that is a superposition of multiple basis states, the gate's output is simply the corresponding superposition of the corresponding output states. As an example, consider the CNOT gate acting on a control qubit that begins in the state $\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$ and a target qubit that begins in the state $|1\rangle$. The joint state of the two qubits is

$$\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big) \otimes |1\rangle = \frac{1}{\sqrt{2}}\big(|01\rangle + |11\rangle\big). \tag{1.40}$$

We already know that the CNOT gate acts on the state $|01\rangle$ to produce $|01\rangle$ and on the state $|11\rangle$ to produce $|10\rangle$. Therefore, by linearity, when acting on the state given in (1.40), the CNOT gate produces

$$\frac{1}{\sqrt{2}}\Big(\text{CNOT}\big(|01\rangle\big) + \text{CNOT}\big(|11\rangle\big)\Big) = \frac{1}{\sqrt{2}}\big(|01\rangle + |10\rangle\big). \tag{1.41}$$

This is the same entangled state previously seen in Section 1.1.3, and shows that the CNOT gate is capable of entangling two qubits when they begin in an unentangled state.

As another example of linearity, suppose that we again have a CNOT gate acting on the same two-qubit system as before, except that the initial state of the target qubit is changed to $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$. Then, the joint state of the two qubits is

$$\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big) \otimes \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big) = \frac{1}{2}\big(|00\rangle - |01\rangle + |10\rangle - |11\rangle\big), \tag{1.42}$$

and the CNOT gate acting on this state produces

$$\frac{1}{2}\big(|00\rangle - |01\rangle - |10\rangle + |11\rangle\big) = \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{1.43}$$

From the right-hand side of (1.43), we see that in this case the resulting state is separable. Furthermore, there is the curious result of the target qubit's state seemingly being unaffected, but the control qubit's state having its relative phase altered. This is surprising because for basis states, the CNOT gate is defined to leave the control qubit unchanged. We therefore see that a controlled gate has the ability to alter the relative phase of the state of its control qubit or qubits, even though these qubits always remain unchanged when the gate acts on a basis state. This relative-phase-affecting ability is used in Grover's algorithm, discussed in Section 1.4, to mark a particular component of a quantum state by inverting its phase.

Linearity applies in the same way to all quantum gates, so the behavior of a Toffoli gate, controlled-V gate, or any other quantum gate acting on a superposition is similarly determined by the gate's behavior when acting on basis states. In particular, suppose that G is a quantum gate acting on n qubits, and let $|0\rangle$ through $|2^n - 1\rangle$ be a basis for the state of that n-qubit system. Then, given an initial state $|\psi\rangle$, expressed in the basis as

$$|\psi\rangle = \sum_{i=0}^{2^n-1} a_i |i\rangle, \tag{1.44}$$

the result of G acting on $|\psi\rangle$ is given by

$$G\big(|\psi\rangle\big) = \sum_{i=0}^{2^n-1} a_i G\big(|i\rangle\big). \tag{1.45}$$

Therefore, the behavior of G is completely determined by its behavior when acting on each

of the basis states $|i\rangle$ for $0 \leq i \leq 2^n - 1$.

Because of linearity, when analyzing the behavior of a quantum circuit, it is sufficient to determine how the circuit behaves when acting on basis states only. If a circuit's behavior is known for a complete set of initial basis states, then we can be assured that the circuit will also behave as expected for all other possible initial states, of which there are infinitely many. For instance, consider a quantum circuit consisting of two adjacent controlled-V gates as shown in Figure 1.26. We can easily summarize how this circuit behaves when the control qubit begins in one of the basis states: if the control qubit begins in state $|0\rangle$, no action is performed on the target qubit, and if the control qubit begins in state $|1\rangle$, two V gates are applied to the target qubit, which is equivalent to a single NOT gate as described in Section 1.2.4. Therefore, the circuit behaves as a controlled-NOT gate when acting on basis states. Linearity then guarantees that the circuit also behaves as a controlled-NOT gate for all other initial states, and therefore it is in fact equivalent to a single controlled-NOT gate.

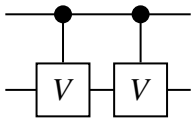


Figure 1.26: Two adjacent controlled-V gates, equivalent to a single CNOT gate.

There is one important caveat to (1.45) and the discussion of the previous paragraph: global phase factors must be taken into account when evaluating the behavior of the circuit or gate in question for basis states. Although a global phase by itself is not physically meaningful, when two or more of the $G\big(|i\rangle\big)$ terms on the right-hand side of (1.45) are summed, their global phases become relative phases, which cannot be ignored. This is the reason for introducing "phase-corrected" rotation gates in Section 1.2.5: when such a gate is used as the target gate in a controlled gate, a global phase for the uncontrolled gate in isolation becomes a relative phase. For instance, consider the $R_x(\pi/2)$ gate defined in (1.22).

This gate differs from the V gate only by a constant multiple of $(1+i)/\sqrt{2}$, and therefore is indistinguishable from the V gate in isolation. However, the controlled-$R_x(\pi/2)$ gate is not the same as the controlled-V gate: the former has matrix representation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{i}{2} \\ 0 & 0 & -\frac{i}{2} & \frac{1}{2} \end{bmatrix} \tag{1.46}$$

while the latter has representation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} \\ 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} \end{bmatrix}. \tag{1.47}$$

We can see that what was a global phase has become a relative phase, because the basis states for which the controlled gate is inactive (*i.e.*, the control qubit is in the state $|0\rangle$) do not undergo any phase shifts at all. Only the controlled-V gate has the property, described in the previous paragraph, that two adjacent controlled-V gates acting on the same control and target qubits are equivalent to a single CNOT gate. The controlled-$R_x(\pi/2)$ gate does not have this property, as can be verified by squaring the matrix (1.46). Introducing the "phase-corrected" $\tilde{R}_x(\theta)$ gates fixes this problem: the $\tilde{R}_x(\pi/2)$ gate is exactly the same as the V gate, including the global phase factor, and therefore a controlled-$\tilde{R}_x(\pi/2)$ is the same as the controlled-V gate.

Throughout this dissertation, I will rely on linearity of quantum circuits when demonstrating that presented circuits or families of circuits behave as claimed. In particular, in Chapters 2 through 4, I will, without comment, implicitly use arguments similar to the one

above showing that two adjacent controlled-V gates are equivalent to a single controlled-NOT gate. In other chapters I will similarly make use of linearity without comment and analyze the behavior of quantum circuits by only considering their behavior when acting on initial basis states. For the same reason, I will also sometimes conflate quantum states with the states of ordinary classical bits, for instance referring to or labeling a qubit as having a "value of 0" when this really means that the qubit is in the state $|0\rangle$. No confusion can arise from this use of terminology when only basis states are being considered.

1.4 Grover's search algorithm

Note: This section was previously published as Section 3.1 of the following journal article:
E. Tsai and M. A. Perkowski, "A quantum algorithm for automata encoding," *Facta Universitatis, Series: Electronics and Energetics*, vol. 33, no. 2, pp. 169–215, 2020.

Grover's algorithm [6, 7, 8] is a well-known quantum algorithm and was one of the first theoretical demonstrations of quantum supremacy—that is, Grover's algorithm runs with a lower time complexity than any possible classical algorithm for the same task. The task performed by Grover's algorithm is to solve the problem of satisfying a Boolean function: given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, find x_1 through x_n such that $f(x_1, \ldots, x_n) = 1$. This type of problem is commonly known as a satisfaction or decision problem. Classically, assuming that nothing is known about the function f, one can on average do no better than an exhaustive search of all possible inputs to f. Such a search requires time $\mathcal{O}(N)$ where $N = 2^n$, the total number of possible assignments of values $\{0,1\}$ to the variables x_1 through x_n. If a quantum computer is available, however, Grover's algorithm provides a method to solve the same problem in $\mathcal{O}\left(\sqrt{N}\right)$ time, a quadratic speedup.

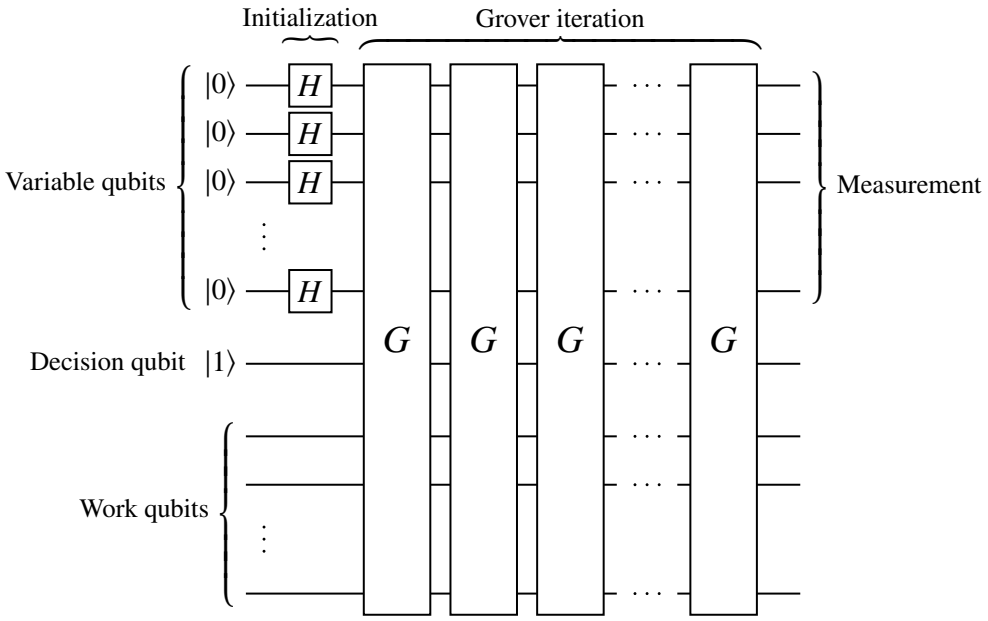


Figure 1.27: A high-level schematic for Grover's algorithm.

Figure 1.27 shows a high-level schematic of a quantum circuit implementing Grover's algorithm. We see that the algorithm consists of an initialization step involving Hadamard gates, followed by repeated applications of an operation G (the *Grover iterate*), and then measurement at the end. First, we examine the Grover iterate, which is the most important step. The Grover iterate accomplishes the task of *amplitude amplification*, the central concept underlying Grover's algorithm. Amplitude amplification is the process of selectively increasing the amplitudes of certain component states of a superposition, while decreasing the amplitudes of others. In Grover's algorithm, the component states whose amplitudes are increased correspond to variable assignments satisfying the function f. By applying the Grover iterate enough times to a superposition, one obtains a quantum state where the components corresponding to satisfactory variable assignments have large amplitudes; all other components have zero or negligibly small amplitudes. A measurement then gives (with probability near 1) a variable assignment satisfying f.

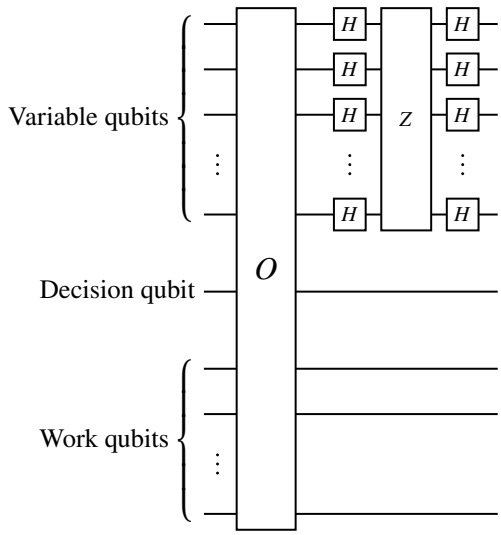


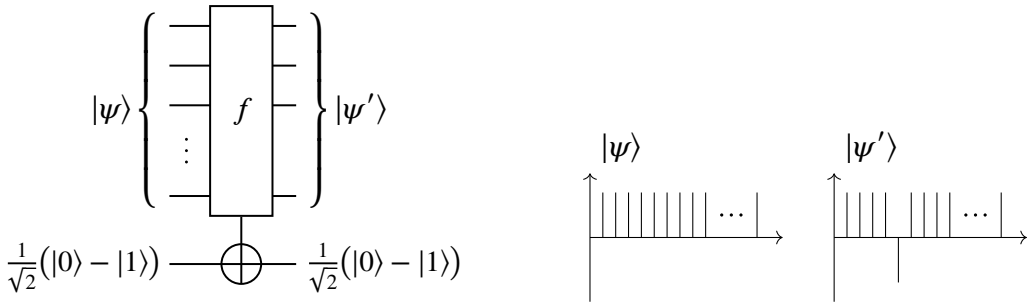
Figure 1.28: Structure of the Grover iterate G from Figure 1.27 with quantum oracle O and zero-state phase shift Z.

Figure 1.28 depicts a schematic of the Grover iterate, showing that it consists of a *quantum oracle O*, followed by the *zero-state phase shift Z* surrounded by arrays of Hadamard gates. The quantum oracle is a quantum circuit implementation of the function f to be satisfied. More specifically, the oracle is an f-controlled inverter as previously defined and illustrated in Figure 1.11. The quantum oracle essentially defines the search criteria for Grover's algorithm and is the only part of Grover's algorithm whose precise details depend on the nature of the function f. Therefore, *a quantum computer cannot execute Grover's algorithm for a given function f unless a quantum oracle for f is available*. For some if not the majority of satisfaction problems arising from practical scenarios, the function f is not given explicitly as a formula, but is instead only defined indirectly through a set of conditions or constraints. In such a case, designing a quantum oracle is far from trivial. Later, we will examine how to design quantum oracles for two different practical problems. For now, we examine the quantum oracle's role in Grover's algorithm without concern for the details of its implementation.

In Grover's algorithm, the quantum oracle performs a phase flip, essentially "marking" the components of a superposition corresponding to variable assignments that satisfy the function f (from now on, we will refer to these components as the "solution" components since they represent solutions to the satisfaction problem). Specifically, one can prove that if one supplies the state $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$ on the oracle's target qubit and supplies any state $|\psi\rangle$ on the variable qubits, then the oracle inverts the phase of the solution components of $|\psi\rangle$. Figure 1.29a illustrates the quantum circuit diagram for an oracle set up to perform the phase inversion, showing the output qubit initialized to $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$. Figure 1.29b then visualizes the initial and final states, $|\psi\rangle$ and $|\psi'\rangle$, of the oracle's variable qubits by plotting their component amplitudes. In other words, given the state $|\psi\rangle$ (or $|\psi'\rangle$) expressed as a sum of components,

$$|\psi\rangle = \sum_{i=0}^{N-1} a_i |i\rangle, \tag{1.48}$$

the graphs plot a_i against i. Comparison between $|\psi\rangle$ and $|\psi'\rangle$ shows that the phase of a single component (the solution component) has been inverted. This illustration assumes that only a single solution component exists; we discuss the case of multiple solutions later on.



(a) A quantum oracle set up to perform a phase flip. **(b)** Amplitude-graph visualization of the phase flip.

Figure 1.29: The phase-flip stage of the Grover iterate.

We observe that the phase inversion can be interpreted to mean that the oracle in a

sense evaluates the function f simultaneously for all possible inputs and finds the solution immediately. Unfortunately, the solution is encoded as a phase variation which is not directly observable. Therefore, the amplitude amplification procedure must convert this phase difference into a measurable amplitude difference. A combination of zero-state phase shift Z and Hadamard gates, as seen on the right-hand side of 1.28, accomplishes this task. The zero-state phase shift is defined to invert the phase of the $|0\rangle$ component of any quantum state on which it acts. Specifically, given a state $|\psi\rangle$ expressed as a sum of components as above, the operation of Z on this state produces the state

$$Z|\psi\rangle = -a_0|0\rangle + \sum_{i=1}^{N-1} a_i|i\rangle. \tag{1.49}$$

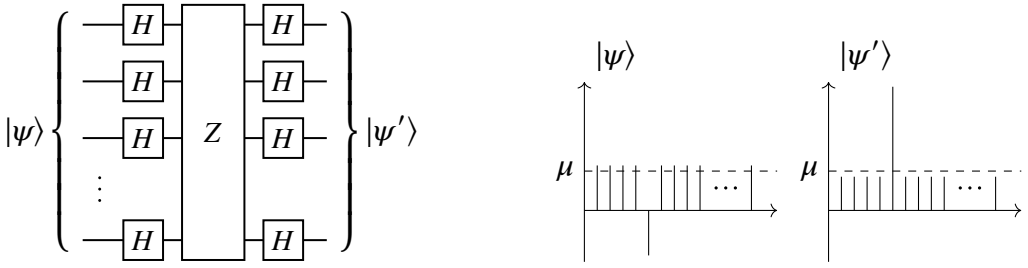
One can prove that when Hadamard gates are applied before and after a zero-state phase shift as shown in Figure 1.28, the result is an *inversion about the mean*, defined as follows: if μ denotes the mean amplitude of all the components of a quantum state, then under an inversion about the mean, each component's amplitude becomes $2\mu - a$ where a is the component's prior amplitude. This transformation is mathematically represented by the following equation:

$$HZH\left(\sum_{i=0}^{N-1} a_i|i\rangle\right) = \sum_{i=0}^{N-1}(2\mu - a_i)|i\rangle, \tag{1.50}$$

where HZH denotes the aforementioned combination of Hadamard gates and the zero-state phase shift.

Figure 1.30 depicts the effect of inversion about the mean when applied immediately following the quantum oracle. Figure 1.30a shows the portion of the Grover iterate G (from Figures 1.27 and 1.28) that performs inversion about the mean. Figure 1.30b then visualizes the initial and final quantum states in the same manner as Figure 1.29. We see

that following inversion about the mean, the amplitudes of the solution components have increased while those of all other components have decreased. Successive application of the Grover iterate similarly increases the amplitudes of the solution components further. After enough iterations, the superposition consists entirely or nearly entirely of solution components. Specifically, one can prove that the number of iterations required is on the order of $\sqrt{N}$, where N is the total number of components (*i.e.*, the number of possible assignments of values to variables). A measurement then gives (with near certainty) a variable assignment satisfying the function f.



(a) The circuit that performs the inversion. **(b)** Amplitude-graph visualization.

Figure 1.30: Inversion about the mean, the second stage of the Grover iterate.

Since we assume that nothing is known in advance about the function f, the just-described amplitude amplification procedure should begin with a superposition of all possible variable assignments, as any of them could potentially satisfy f. Therefore, before performing the amplitude amplification procedure, Grover's algorithm first initializes the qubits that will serve as inputs to the quantum oracle, x_1 through x_n, to a superposition of all possible states. Conventionally, this is accomplished by initializing each variable qubit (x_1 through x_n) to $|0\rangle$ and then applying a Hadamard gate to each variable qubit, as shown in 1.27. Similarly, the quantum oracle's output qubit is initialized to $\frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$ by applying a Hadamard gate to an initial state of $|1\rangle$, also shown in Figure 1.27. Following completion of the amplitude amplification procedure, the qubits x_1 through x_n are measured to obtain the final output of

Grover's algorithm. We therefore see that the complete Grover's algorithm consists of the following steps:

1. Begin with n variable qubits and one target qubit, initialized as described in the preceding paragraph.

2. Perform the amplitude amplification procedure by applying the Grover iterate $\sqrt{N}$ times as previously described.

3. Measure the variable qubits; the result of measurement forms the output of Grover's algorithm.

Since the initialization and measurement steps in Grover's algorithm are proportional to the number of qubits used, which is on the order of $n = \log_2 N$, the number of variables of f, the $\mathcal{O}\left(\sqrt{N}\right)$ time required for the amplitude amplification procedure forms the dominant contribution to the overall runtime of Grover's algorithm. Therefore, Grover's algorithm requires a time complexity of $\mathcal{O}\left(\sqrt{N}\right)$.

If presented with a function f that can be satisfied in multiple ways—that is, multiple solutions exist—Grover's algorithm can still find one of the solutions, although it must be slightly modified. Boyer *et al.* [21] show that if the number of solutions is k, then Grover's algorithm will work if the number of iterations is reduced to $\mathcal{O}\left(\sqrt{N/k}\right)$, with all other aspects of the algorithm remaining unchanged. Brassard *et al.* [22] present a *quantum counting* algorithm that can be used to estimate the number of solutions. Therefore, given a function f with an unknown number of solutions, one can first estimate the number of solutions and then apply Grover's algorithm with the appropriately modified number of iterations. Even though an additional quantum counting step is introduced, the number of

Grover iterations is reduced, so this procedure still results in a significant speedup compared to an uninformed classical search.

Our discussion so far has implicitly assumed that a solution to the satisfaction problem being solved by Grover's algorithm exists in the first place. If no solution exists, Grover's algorithm can detect this fact with high certainty. A quantum oracle for a satisfaction problem with no solutions will not perform any phase inversions when it is applied as in 1.29. The inversion about the mean will then have no effect on the amplitude of any component of the resulting quantum state. Therefore, at the conclusion of Grover's algorithm, one will be measuring a quantum state consisting of a superposition of all possible basis states, and will therefore obtain a completely random answer. One can check the answer returned by Grover's algorithm, using either a classical computer or the same quantum oracle with just a single basis state as input. If the returned answer is incorrect, one then deduces with high certainty that no solution exists, because Grover's algorithm is certain or near-certain to produce a solution when one exists.

1.5 Organization of this dissertation

The remainder of this dissertation is divided into three parts. In the first part, consisting of Chapters 2 through 4, I consider the realization of a special class of Boolean functions known as symmetric functions. Specifically, Chapter 2 first introduces the main concepts that will be used in all three chapters and illustrates how those concepts can be used to obtain realizations for a limited class of symmetric functions. Chapter 3 then extends this limited class to a larger class and also demonstrates one particular application of the realized symmetric functions as a quantum counter. Chapter 4 introduces a method based on the Walsh-Hadamard transform that allows all symmetric functions to be realized using the class

of functions from Chapter 3. Chapter 4 also demonstrates, via the technique of *symmetrizing* functions by repeating variables, that even non-symmetric Boolean functions can be realized using the same method used for symmetric functions.

The second part of this dissertation comprises Chapters 5 and 6, which address the realization of permutative functions with the same number of inputs and outputs. In Chapter 5, I introduce the concept of *distance gates* together with a cycle-based algorithm for realizing permutative Boolean functions using distance gates. In Chapter 6, I extend distance gates to a multiple-valued setting and show that they can also be used for cycle-based synthesis of permutative functions with multiple-valued inputs and outputs.

Chapters 7 and 8 deal with the design of quantum oracles for Grover's algorithm. In Chapter 7, I demonstrate the detailed design of a quantum oracle to be used with Grover's algorithm to solve a state-machine encoding problem. Along the way, I introduce a method by which Grover's algorithm can be used to solve an optimization problem, even though it only directly solves satisfaction problems. In Chapter 8, I demonstrate a quantum oracle design for a completely different class of problems, which I call state-space path planning problems. I show detailed oracle designs for two such problems, and based on these examples, I describe a general approach that can be used to solve other problems of this type as well. Both chapters combined demonstrate several different uses of quantum counters in oracle design.

Finally, Chapter 9 summarizes my achievements and contributions and concludes the dissertation.

Chapter 2

Direct realization of single-output Boolean symmetric functions using two-qubit gates

Universality is a requirement for any general-purpose computing system. A computing system is said to be universal if it is capable of computing the value of any function that can in principle be computed. Universality in hypothetical quantum computing systems has, for the most part, been achieved using Toffoli gates, which are known to be universal. In particular, most of the existing literature has built on the work of Barenco *et al.* [20] and Maslov and Dueck [19], who demonstrated implementations of n-qubit Toffoli gates using two-qubit controlled gates.

Although Toffoli gates are universal, they do not necessarily provide the optimal realization for many functions of interest, where "optimal" means using as few primitive operations as possible. In other words, the possibility of realizing arbitrary Boolean functions (thereby achieving universality) without using n-qubit Toffoli gates as intermediates has received little attention. This is unfortunate because circuit optimizations that are not apparent when using high-level gates like Toffoli gates can appear when those gates are decomposed to their constituent primitives [23].

Totally symmetric Boolean functions are a special type of Boolean functions with the property that they are unchanged by any permutation of their inputs [24, 25, 26]. Symmetric functions have attracted a wide range of attention, including theoretical complexity analysis

[27, 28, 29] and synthesis methods for reversible logic [30, 31, 32, 33, 34, 35, 36, 37].

I first introduced the concept of "TISC gates" in [38]. These gates were implemented using a scalable circuit structure that realized a specific family of symmetric functions. In this chapter, I demonstrate a far-reaching generalization of the TISC gate concept, which allows for the realization of a class of functions that I call *zero-offset dyadic indicator-periodic symmetric functions* using only two-qubit controlled gates. I introduce a recursively-defined circuit structure for this purpose, in which symmetric functions are arranged in a hierarchy beginning with the exclusive-OR function, and a function at any point in the hierarchy is realized with the help of functions lower in the hierarchy. This circuit structure is naturally scalable to any number of qubits, allowing it to realize infinite families of symmetric functions consisting of what are essentially different versions of the same function for different numbers of inputs.

2.1 Preliminary definitions and notations

A *symmetric function* is a function with the property that its output remains unchanged under any permutation of its inputs.[1] For instance, the function of two real numbers $f(x, y) = x + y$ is symmetric because addition of real numbers is commutative, so $f(x, y) = x + y = y + x = f(y, x)$. The function $g(x, y) = x + xy$, on the other hand, is not symmetric because $g(y, x) = y + yx$, which is not equal to $g(x, y)$ when $x \neq y$.

In this and the next chapter, I will consider Boolean symmetric functions in particular; that is, symmetric functions with the output and all inputs being Boolean variables that only

[1] The term "symmetric function" is commonly used to denote functions satisfying only a weaker property: that the function's output remains unchanged under permutations of some subset or subsets of its input variables. For instance, a function of three variables a, b, and c could be described as "symmetric" (with respect to the subset $\{a, b\}$) if it remains unchanged under exchange of a and b but not a and c. Functions that remain unchanged under any permutation of inputs are then called "totally symmetric". Here, I consider only totally symmetric functions, and therefore I will refer to such functions as simply "symmetric" for simplicity.

take on the values 0 and 1. Therefore, from now on, "symmetric function" without further qualification will refer to a Boolean symmetric function unless otherwise specified. The standard NOT, AND, and OR operations of Boolean algebra are all symmetric functions: $NOT(x) = \neg x$ is trivially symmetric since it takes only a single input, while $AND(x, y) = x \wedge y = y \wedge x$ and $OR(x, y) = x \vee y = y \vee x$ are symmetric by commutativity.

The requirement that a symmetric function's output remain unchanged under *any* permutation of its inputs implies that the output of a Boolean symmetric function can only depend on the number of 1s (or equivalently, the number of 0s) present among its input variables. Following Maslov's terminology [33], I will refer to any collection of specific input values to a symmetric function as an "input pattern", and the number of 1s in an input pattern as its *weight*. I will write $\sum_{i=1}^{n} x_i$ for the weight of the input pattern $x_1 \ldots x_n$, where x_1 through x_n are all Boolean variables. This notation derives from treating the possible values for a Boolean variable, 0 and 1, as integers as opposed to abstract labels (which can be synonymous with other labels such as "true/false" or "on/off"). The sum $\sum_{i=1}^{n} x_i$ then represents an ordinary arithmetic sum of integers, as opposed to the logical OR operation in Boolean algebra, which is sometimes also written as a sum. I will sometimes drop the limits from this sum and simply write $\sum x_i$, with the understanding that the sum is always over all x_i.

The following definition formalizes the concept of a symmetric function according to the preceding discussion and also introduces some additional terminology:

Definition 1. A function $f\colon \{0,1\}^n \to \{0,1\}$ that takes n Boolean variables and returns another Boolean value is a *symmetric function* if it satisfies the following condition: letting $\mathbb{N}_{\leq n}$ denote the set of natural numbers not greater than n, there exists another function

$I : \mathbb{N}_{\leq n} \to \{0, 1\}$ for which

$$f(x_1, \ldots, x_n) = I\left(\sum_{i=1}^{n} x_i\right) \tag{2.1}$$

for all $x_1, \ldots, x_n \in \{0, 1\}$. The function I will then be called an *indicator function* for the symmetric function f. The set of natural numbers $S_{\text{ON}} = \left\{w \in \mathbb{N}_{\leq n} \mid I(w) = 1\right\}$ will be called an *ON-set* for f.

Essentially, an indicator function is a function that expresses a symmetric function in terms of the weight of its input pattern rather than the inputs themselves. The name "indicator function" comes from the fact that it can be thought of as the set-theoretic indicator function[2] of a certain set, an ON-set, which contains those input pattern weights for which the symmetric function outputs 1. For a fixed number of inputs, an indicator function or ON-set uniquely determines a symmetric function via Definition 1. One can see that the converse is also true: if one has an n-input symmetric function f, it is easy to produce an input pattern having any weight from 0 through n.[3] Then, the value of the indicator function for any w is found by evaluating f on an input pattern of weight w, and the ON-set contains w if and only if f evaluates to 1. This observation is restated as the following proposition for convenience.

Proposition 2. *For a fixed number of inputs n, there is a one-to-one correspondence between n-input symmetric functions, indicator functions, and ON-sets. In other words, given a function $I : \mathbb{N}_{\leq n} \to \{0, 1\}$ or a set $S_{\text{ON}} \subset \mathbb{N}_{\leq n}$, there exists exactly one n-input symmetric function with I as an indicator function or S_{ON} as an ON-set, and vice versa.*

From now on, I will therefore refer to *the* indicator function or ON-set of a symmetric

[2] In set theory, the indicator function of a set S is a function which outputs 1 if its input is an element of S and 0 otherwise. A set is completely determined by its indicator function and vice versa.

[3] For instance, to get a weight of w, take $x_i = 1$ for $i < w$ and $x_i = 0$ otherwise.

function.

The concept of an ON-set allows for the following convenient notation for symmetric functions: $\mathrm{S}_n^{S_{\mathrm{ON}}}$ will denote the n-input symmetric function with ON-set S_{ON}. For instance, $\mathrm{S}_3^{\{1,2\}}$ denotes a symmetric function of three variables that returns 1 when its input pattern has a weight of 1 or 2; that is, when one or two of its inputs are 1. Because of the one-to-one correspondence between ON-sets and symmetric functions, this notation is unambiguous and can represent all possible symmetric functions. I will drop the n subscript when the number of inputs is clear from the context: $\mathrm{S}_3^{\{1,2\}}(a, b, c)$ becomes $\mathrm{S}^{\{1,2\}}(a, b, c)$. I will also drop the braces from the ON-set when it is presented as a list of its elements, so $\mathrm{S}^{\{1,2\}}(a, b, c)$ further simplifies to $\mathrm{S}^{1,2}(a, b, c)$. Combining this notation with Definition 1 leads to the following equivalence:

$$\mathrm{S}^{w_1,w_2,\ldots,w_m}(x_1, \ldots, x_n) = 1 \text{ if and only if } \sum_{i=1}^{n} x_i = w_j \text{ for some } j, \tag{2.2}$$

where w_1 through w_m are natural numbers with $w_j \leq n$ for all j.

In Proposition 2, the condition that a number of inputs n be fixed is important. If this condition is removed, it is possible for a single ON-set to correspond to more than one symmetric function, because one can have a whole family of symmetric functions, each with differing number of inputs, that all share the same ON-set. For instance, given the ON-set $\{1, 2\}$, the symmetric functions $\mathrm{S}_3^{1,2}$, $\mathrm{S}_4^{1,2}$, and indeed $\mathrm{S}_n^{1,2}$ for any $n \geq 2$ all share the same ON-set, $\{1, 2\}$.

A related concept to a family of symmetric functions sharing the same ON-set is a family of symmetric functions that would share the same ON-set, if not for some members of the family not having enough inputs to cover the whole ON-set. For instance, given the set $\{3, 4, 5, 6\}$, the corresponding family of symmetric functions includes S_3^3, $\mathrm{S}_4^{3,4}$, $\mathrm{S}_5^{3,4,5}$, and

$S_6^{3,4,5,6}$, where the ON-sets of the first three members have been "truncated" due to them not having enough inputs. It is even possible for such a family to be defined by an infinite ON-set; in such a case, the ON-set is truncated to a finite set for every member of the family. An indicator function can also be defined for the family as a whole, rather than for its individual members, in which case the indicator function's domain consists of all natural numbers with no upper limit. Such families will feature prominently in this chapter, to the point where I essentially only consider such families as a whole rather than considering individual symmetric functions. This is because the circuit structures that I introduce are naturally scalable to any number of qubits and so realize such families all at once. In a slight abuse of terminology, I will sometimes use phrases like "the symmetric function $S_n^{2,3,4,\ldots}$" when I am really referring to the whole infinite family of functions for all n, with the understanding that the statement applies to all members of the family. Similarly, I will speak of the indicator function or the ON-set of such a "function" when I am really referring to the shared indicator function or ON-set of the whole family. There is little opportunity for confusion because, as already stated, this and the following chapter will be concerned almost exclusively with such families of symmetric functions rather than with their individual members.

Using the concepts and notations introduced above, we can express some standard operations of Boolean algebra in terms of symmetric functions. The logical AND of n Boolean variables, x_1 through x_n, is given by $\bigwedge_{i=1}^{n} x_i = S^n(x_1, \ldots, x_n)$ since the result of a logical AND is 1 only when all n inputs are 1. The corresponding indicator function is defined by $I_\wedge(n) = 1$ and $I_\wedge(w) = 0$ for all other w. Similarly, the logical OR of the same n Boolean variables is given by $\bigvee_{i=1}^{n} x_i = S^{1,2,\ldots,n}(x_1, \ldots, x_n)$ since the result of a logical OR is 1 when at least one of the inputs is 1. The corresponding indicator function is then defined by $I_\vee(0) = 0$ and $I_\vee(w) = 1$ for all other w. Of particular interest is the exclusive

OR: the exclusive OR of x_1 through x_n is 1 whenever an odd number of x_1 through x_n are 1, so it has the symmetric-function representation $\bigoplus_{i=1}^{n} x_i = \mathrm{S}^{1,3,5,\cdots}(x_1, \ldots, x_n)$, where the ON-set $\{1, 3, 5, \ldots\}$ contains the odd natural numbers up to n. The corresponding indicator function is

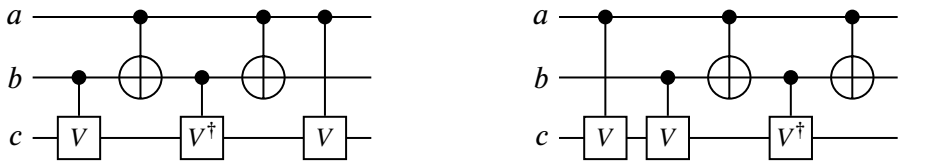
$$I_{\oplus}(w) = w \bmod 2 = \begin{cases} 0 & w \text{ even}, \\ 1 & w \text{ odd}. \end{cases} \tag{2.3}$$

Equation (2.3) will play an important role in the realization method for symmetric functions described in this chapter.

2.2 Realization of symmetric functions with controlled-V and $V^\dagger$ gates

2.2.1 Extension of circuit structure for realizing the Toffoli gate

It is a well-known result [20] that a Toffoli gate may be implemented using controlled-NOT and controlled-V gates, using the circuit shown in Figure 2.1a. In the same work, Barenco *et al.* also demonstrated that n-bit Toffoli gates for any n could be implemented by extended variants of this circuit. Here, I will demonstrate a different way in which this circuit may be extended so as to realize symmetric functions other than the logical AND of a collection of variables, which is the function that is realized by an n-bit Toffoli gate.



(a) Original implementation by Barenco *et al.* **(b)** Rearrangement of gates in Figure 2.1a.

Figure 2.1: Two implementations of a Toffoli gate.

Beginning from the original circuit in Figure 2.1a, rearranging the gates yields the circuit shown in Figure 2.1b. It is not immediately clear that this rearrangement does not alter the

circuit's overall operation, but the analysis later in this section will show that it does not. The circuit in Figure 2.1b can be divided into two stages: the first stage, consisting of the first two gates, is a sequence of controlled-V gates in which the last qubit is always the target qubit and all other qubits are used in turn as control qubits. The second stage, consisting of the remainder of the circuit, is a controlled-$V^\dagger$ gate whose control input is the exclusive OR of the first two qubits, which is computed using a CNOT gate. A second CNOT gate is used to restore qubit b to its initial state.

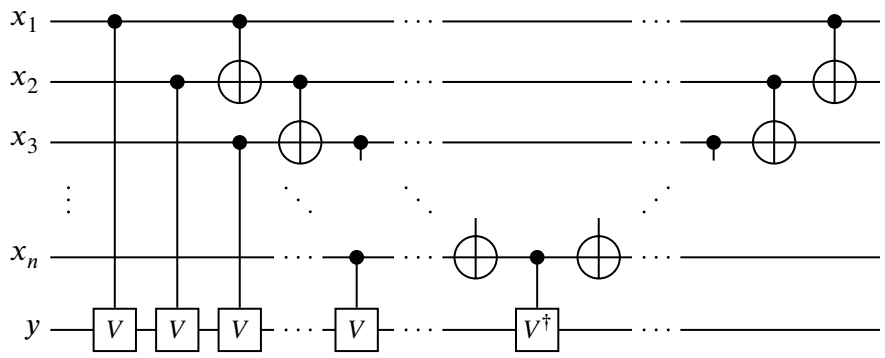


Figure 2.2: Extension of Figure 2.1b to an arbitrary number of qubits.

Based on the division of the circuit from Figure 2.1b into two stages, I introduce the circuit structure shown in Figure 2.2, which follows the same two-stage pattern. The first stage consists of a sequence of controlled-V gates in which qubit y is always the target qubit and qubits x_1 through x_n are each used once as a control qubit. The second stage is then the remainder of the circuit, which consists of a sequence of CNOT gates that compute the exclusive OR of x_1 through x_n, followed by a controlled-$V^\dagger$ gate that uses this exclusive OR as its control input and targets qubit y, followed by another sequence of CNOT gates that acts as a mirror circuit to restore qubits x_1 through x_n to their original states. I refer to the contents of Figure 2.2 as a "circuit structure" rather than just a circuit because it actually represents an infinite family of circuits. Specifically, the number of qubits in Figure 2.2 is

variable and has no upper limit, so the figure represents one circuit for each positive integer n, with the number of qubits in the circuit being $n+1$. I will call any such circuit an *instance* of the circuit structure. In other words, an instance of the circuit structure from Figure 2.2 is any particular circuit with a definite number of qubits that follows this structure.

2.2.2 Analysis of circuit structure by counting 1s

Consider an instance of the circuit structure from Figure 2.2 with $n+1$ qubits. We may think of this circuit as realizing a function f of the variables x_1 through x_n, where f is unknown and to be characterized. The qubit y is not an input to f and is instead used to receive its output. The only gates targeting qubit y are controlled V and $V^\dagger$ gates, meaning that the effect of the circuit will be to apply some sequence of V and $V^\dagger$ gates to y. Qubits x_1 through x_n are unaffected by the circuit due to the presence of mirror gates (the CNOT gates at the end of the circuit) that restore these qubits to their initial state at the end of the circuit. Therefore, analyzing the circuit's operation amounts to determining how the sequence of V and $V^\dagger$ gates applied to y depends on the values of x_1 through x_n.

As discussed in Section 1.2.4, we can visualize the effect of the V and $V^\dagger$ gates as rotations of the Bloch sphere around its x-axis, where a V gate acts by rotating 90° clockwise and a $V^\dagger$ gate acts by rotating 90° counterclockwise. A rotation of 90° counterclockwise can also be thought of as a rotation of $-90°$ clockwise, and consecutive rotations around the same axis simply combine their angles additively. Therefore, given any sequence of V and $V^\dagger$ gates, the effect of this sequence does not depend on the order in which the individual gates are applied, but only on the sum of the rotation angles contributed by those gates. We may define a quantity, the total effective number of V gates, as the number of V gates minus the number of $V^\dagger$ gates in the sequence. The sum of the angles of all rotations performed

by the sequence is then equal to this total effective number times 90°. For instance, in a sequence of three V gates and two $V^\dagger$ gates, the total effective number of V gates is $3-2=1$ and the sum of the rotation angles is 90°. Since a single V gate also performs a 90° rotation, the entire sequence is equivalent to a single V gate, and this is true regardless of the order of gates in the sequence. Because $V^4 = I$, the total effective number of V gates can be reduced modulo 4, so that it is always one of 0, 1, 2, or 3.

Table 2.1 now shows an analysis that applies to any instance of the circuit structure from Figure 2.2. The first column of this table lists the possible input weights $\sum x_i$, which are simply all the natural numbers. The second and third columns show, for each input weight, the number of V gates applied by the first and second stages of the circuit, respectively. As before, "first stage" refers to the sequence of controlled-V gates at the start of the circuit, and "second stage" refers to the remainder of the circuit, which consists of the controlled-$V^\dagger$ gate whose control input is the exclusive-OR of x_1 through x_n. The entries in the second and third columns of Table 2.1 are determined as follows. In the first stage, since each of x_1 through x_n acts as a control input to exactly one of the controlled-V gates, the number of V gates applied in this stage is simply equal to the input weight, $\sum x_i$. In the second stage, the exclusive-OR of x_1 through x_n will be equal to 0 if the input weight is even and 1 if the input weight is odd, as expressed by (2.3). A $V^\dagger$ gate is therefore applied if the input weight is odd. As discussed previously, in a sequence of V and $V^\dagger$ gates, the total effective number of V gates is the number of V gates minus the number of $V^\dagger$ gates, so a single $V^\dagger$ gate is considered to contribute -1 to this total effective number. The third column of Table 2.1 accordingly contains a 0 for even input weights and a -1 for odd input weights.

The fourth column of Table 2.1 shows the total effective number of V gates resulting from each input weight, which is obtained by adding the entries in the second and third columns.

Table 2.1: Calculation of total effective number of V gates applied by instances of the circuit structure from Figure 2.2.

$\sum_{i=1}^{n} x_i$	Applied V gates		Total effective number of V gates	Effective number of V gates modulo 4	Operation applied to qubit y
	Stage 1	Stage 2			
0	0	0	0	0	I
1	1	−1	0	0	I
2	2	0	2	2	X
3	3	−1	2	2	X
4	4	0	4	0	I
5	5	−1	4	0	I
6	6	0	6	2	X
7	7	−1	6	2	X
8	8	0	8	0	I
⋮	⋮	⋮	⋮	⋮	⋮

As per the earlier discussion about this total effective number, the entries of this column may be reduced modulo 4, giving the fifth column. The sixth and final column then shows the equivalent single gate for each total effective number of V gates. Specifically, since the V gate is a square-root-of-NOT gate, a total effective number of 2 V gates is equivalent to a single NOT gate (denoted X). A total effective number of 0 is of course equivalent to an identity gate by definition.

To summarize, the result of the preceding discussion is that the last column of Table 2.1 shows the equivalent gate applied to qubit y for each possible input weight. We can see that qubit y is inverted when the input weight is an element of the set $\{2, 3, 6, 7, 10, \ldots\}$, which consists of the natural numbers that are congruent to either 2 or 3 modulo 4. If the input weight is not in this set, then the circuit applies the equivalent of an identity gate to qubit y; *i.e.*, no change is made to y. Therefore, the operation of the circuit is described by the

equation

$$y' = \begin{cases} y & \text{if } \sum_{i=0}^{n} x_i \bmod 4 = 2 \text{ or } 3 \\ \neg y & \text{otherwise} \end{cases} \tag{2.4}$$

$$= \begin{cases} y & \text{if } \mathrm{S}^{2,3,6,\cdots}(x_1, \ldots, x_n) = 1 \\ \neg y & \text{otherwise} \end{cases} \tag{2.5}$$

$$= y \oplus \mathrm{S}^{2,3,6,\cdots}(x_1, \ldots, x_n), \tag{2.6}$$

where y' denotes the new state of qubit y at the end of the circuit. The equivalence (2.2) is used to establish the equality of (2.4) and (2.5). We see that the circuit is equivalent to an inverter controlled by the function $\mathrm{S}^{2,3,6,\cdots}(x_1, \ldots, x_n)$, as shown in Figure 2.3, and therefore it realizes this function.

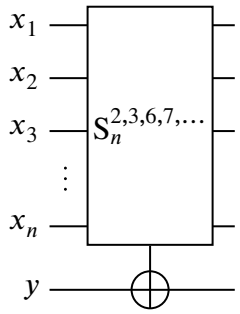


Figure 2.3: Equivalent controlled-gate representation of Figure 2.2, based on Table 2.1.

It is important to observe that the preceding analysis applies to *any* instance of the circuit structure from Figure 2.2, regardless of the number of inputs n. This means that all such circuits realize symmetric functions of the same general form, $\mathrm{S}_n^{2,3,6,7,\cdots}$, differing only in the number of inputs to the function.

2.2.3 Representation of circuit using an operational equation

To investigate further modifications and extensions to the circuit structure from Figure 2.2, it is helpful to represent the information in Table 2.1 as an equation. Let N_V denote the number of V gates that the first stage of the circuit applies to qubit y, $N_{V^\dagger}$ denote the number of $V^\dagger$ gates that the second stage applies, and N_{eff} denote the resulting total effective number of V gates. Then we have

$$N_V - N_{V^\dagger} = N_{\text{eff}}. \tag{2.7}$$

As previously observed, the number of V gates applied in the first stage is equal to the input weight $\sum x_i$, since one V gate is applied for every x_i that begins in the state $|1\rangle$. Therefore,

$$N_V = \sum_{i=1}^{n} x_i. \tag{2.8}$$

The $V^\dagger$ gate in the second stage is controlled by the exclusive-OR of x_1 through x_n. Since an exclusive OR can be represented as a symmetric function, $\mathrm{S}^{1,3,5,7,\cdots}(x_1, \ldots, x_n)$, we can then write

$$N_{V^\dagger} = \mathrm{S}^{1,3,5,7,\cdots}(x_1, \ldots, x_n). \tag{2.9}$$

Finally, we can represent the total effective number of V gates applied by the circuit using another symmetric function:

$$N_{\text{eff}} \equiv 2\mathrm{S}^{2,3,6,7,\cdots}(x_1, \ldots, x_n) \pmod 4 \tag{2.10}$$

where the coefficient of 2 indicates that two effective V gates are applied when the symmetric function $\mathrm{S}^{2,3,6,7,\cdots}(x_1, \ldots, x_n)$ evaluates to 1. In this notation, the output of the symmetric function is treated as an integer to which all standard arithmetic operations can be applied,

and not just as a Boolean variable.

Substituting (2.8), (2.9), and (2.10) into (2.7) yields

$$\sum_{i=1}^{n} x_i - \mathrm{S}^{1,3,5,\cdots}(x_1, \ldots, x_n) \equiv 2\mathrm{S}^{2,3,6,\cdots}(x_1, \ldots, x_n) \pmod 4, \tag{2.11}$$

which compactly summarizes the information presented in Table 2.1 in equation form. As in (2.10), the output of the symmetric function $\mathrm{S}^{1,3,5,\cdots}(x_1, \ldots, x_n)$ is treated as an integer, representing the number of $V^\dagger$ gates applied by the circuit, even though it can only take on the values 0 and 1. The minus sign on the left-hand side of (2.11) thus denotes an ordinary arithmetic subtraction of integers (and not an operation of Boolean algebra), just as it does in (2.7).

We may obtain a simpler form of (2.11) by rewriting it in terms of an input pattern weight and indicator functions. Recall from Definition 1 that the indicator function of a symmetric function expresses the symmetric function in terms of the weight of its input pattern. In (2.11), the first term on the left-hand side is simply the input pattern weight itself and the other terms are symmetric functions, which can be replaced with indicator functions of the input pattern weight. Making these replacements gives

$$w - I_{\oplus}(w) \equiv 2I^{2,3,6,\cdots}(w) \pmod 4, \tag{2.12}$$

where w is the input pattern weight and may therefore be any natural number, $I_{\oplus}$ is the indicator function of the exclusive-OR function (first introduced in 2.1), and $I^{2,3,6,\cdots}$ is the indicator function of the symmetric function $\mathrm{S}_n^{2,3,6,\cdots}$. This last indicator function can be

explicitly written as

$$I^{2,3,6,\cdots}(w) = \begin{cases} 1 & \text{if } w \equiv 2 \text{ or } w \equiv 3 \pmod 4, \\ 0 & \text{otherwise .} \end{cases} \tag{2.13}$$

I refer to (2.11) as an *operational equation* for the circuit structure shown in Figure 2.2 because it expresses the overall operation performed by an instance of this circuit structure as a combination of the operations performed in each stage, where these operations are specified as effective number of V gates.

2.3 Recursive realization of symmetric functions using multi-stage circuits

2.3.1 Replacement of controlled-V and controlled-$V^\dagger$ gates with other root-of-NOT gates

The analysis of the circuit structure from Figure 2.2, as shown in Table 2.1 and compactly summarized by equations (2.11), (2.12), and (2.24), can still be used even if the target gates applied to the output qubit y are replaced with other gates. In Figure 2.2, these target gates are controlled-V and controlled-$V^\dagger$ gates. However, Table 2.1 simply counts the total effective number of gates applied to qubit y in Figure 2.2 and does not depend on what those gates actually are. In other words, if V and $V^\dagger$ are replaced with some other gate U and its inverse $U^\dagger$, then the observation made at the start of Section 2.2.2 remains valid—namely, the behavior of any resulting circuit only depends on the total effective number of U gates applied to qubit y, where this total effective number is defined as the number of applied U gates minus the number of applied $U^\dagger$ gates. The only difference from the analysis conducted in Section 2.2.2 is that the total effective number can no longer be reduced modulo 4, since that ability comes from the status of the V gate as a square-root-of-NOT, which is lost when

V is replaced by a different gate.

To see this principle in action, suppose that we replace V and $V^\dagger$ in Figure 2.2 with X_2 and X_2^{-1} gates, recalling from Section 1.2.5 that X_2 is a fourth-root-of-NOT gate. I have chosen to denote the inverse of X_2 as X_2^{-1} rather than ${X_2}^\dagger$ to emphasize that a X_2^{-1} contributes -1 to the total effective number of X_2 gates. The two notations are equivalent since all quantum gates are unitary. The resulting circuit structure is shown in Figure 2.4, and its analysis via calculation of the total effective number of X_2 gates is shown in Table 2.2.

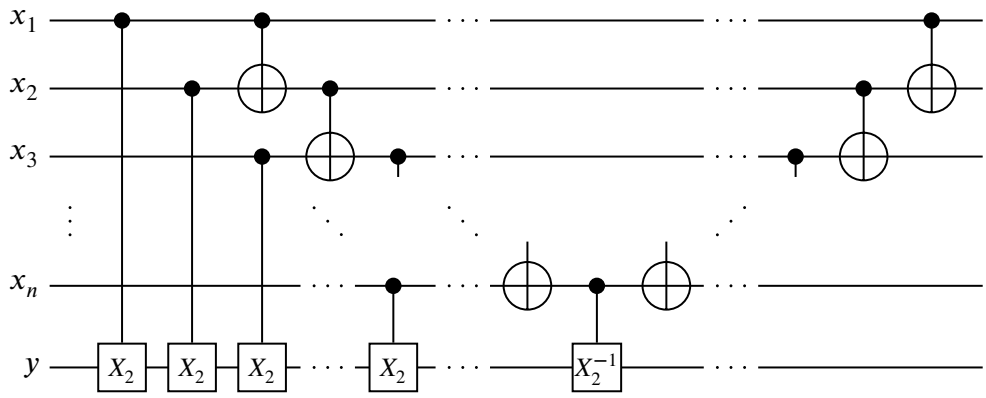


Figure 2.4: Replacement of V and $V^\dagger$ in Figure 2.2 with X_2 and X_2^{-1} gates.

In accordance with the discussion at the start of this section, the first two columns of Table 2.2 are identical to the first and fourth[4] columns of Table 2.1, since these columns show the total effective number of gates as a function of input weight. However, the total effective number of X_2 gates in Table 2.2 cannot be reduced modulo 4. Instead, since X_2 is a fourth-root-of-NOT and therefore an eighth-root-of-identity (*i.e.*, $X_2^8 = I$), the total effective number can be reduced modulo 8, giving the third column of Table 2.2. The last column of Table 2.2 then shows the equivalent gates that the total effective numbers translate into.

Table 2.2 shows that the circuit structure from Figure 2.4 does not produce permutative

[4]Table 2.2 omits the breakdown of applied gates by stage, since there would be no change from the second and third columns of Table 2.1.

Table 2.2: Calculation of total effective number of X_2 gates applied by instances of the circuit structure from Figure 2.4.

$\sum_{i=1}^{n} x_i$	Total effective number of X_2 gates	Effective number of X_2 gates modulo 8	Operation applied to qubit y
0	0	0	$X_2^0 = I$
1	0	0	$X_2^0 = I$
2	2	2	$X_2^2 = V$
3	2	2	$X_2^2 = V$
4	4	4	$X_2^4 = X$
5	4	4	$X_2^4 = X$
6	6	6	$X_2^6 = V^\dagger$
7	6	6	$X_2^6 = V^\dagger$
8	8	0	$X_2^0 = I$
⋮	⋮	⋮	⋮

circuits, unlike the one from Figure 2.2. For example, when presented with an input weight of 2, any instance of the circuit structure from Figure 2.4 applies a total effective number of 2 X_2 gate, which is equivalent to a single V gate, to qubit y. Similarly, when presented with an input weight of 6, instances of Figure 2.4 apply an total effective number of 6 X_2 gates, which is equivalent to a single $V^\dagger$ gate. Since neither the V nor $V^\dagger$ gates are permutative, any instance of Figure 2.4 will perform a non-permutative operation on qubit y when presented with input weights of 2, 3, 6, 7, etc. Therefore, the circuit structure from Figure 2.4 cannot be used on its own to realize Boolean functions.

2.3.2 Three-stage circuit structure

Although the circuit structure from Figure 2.4 on its own does not directly produce permutative circuits, it can do so with a slight modification. This modification takes the form of a third stage in which a $X_2^{-2} = V^\dagger$ gate is controlled by the function $\mathrm{S}^{2,3,6,7,\cdots}(x_1, \ldots, x_n)$.

The function $\mathrm{S}^{2,3,6,7,\cdots}(x_1, \ldots, x_n)$ is realized using the circuit structure from Figure 2.2. It is made to act as a control function to a X_2^{-2} gate using the method illustrated in Figure 1.25, which requires one ancillary qubit. Figure 2.5 shows the resulting circuit structure. In this figure, the blocks labeled with a circled plus sign ($\oplus$) are abbreviations for cascades of CNOT gates used to compute the exclusive-OR of x_1 through x_n, previously shown in full in Figures 2.2 and 2.4. Note that this structure is in a sense recursive, because it extends Figure 2.2 while using copies of Figure 2.2 itself as subcircuits. For reference, Figure 2.6 shows the same circuit structure as Figure 2.5, but with all subcircuits expanded in full using two-qubit controlled gates.

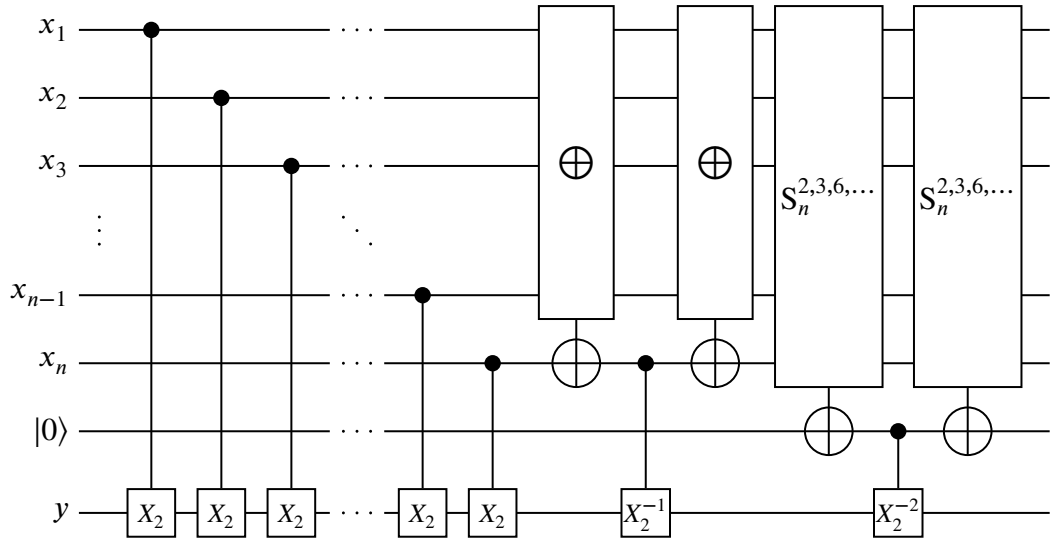


Figure 2.5: The circuit structure from Figure 2.4, with an additional stage added to restore permutativity.

Table 2.3 shows the analysis of the circuit structure from Figure 2.5. This table is analogous to Table 2.1 and tracks the number of X_2 gates applied by the stages of instances of Figure 2.5. The columns "Stage 1" and "Stage 2" are identical to those from Table 2.1, while the column "Stage 3" shows the contribution that the new third stage shown in Figure 2.5 makes to the total effective number of X_2 gates. Since this third stage is controlled by a

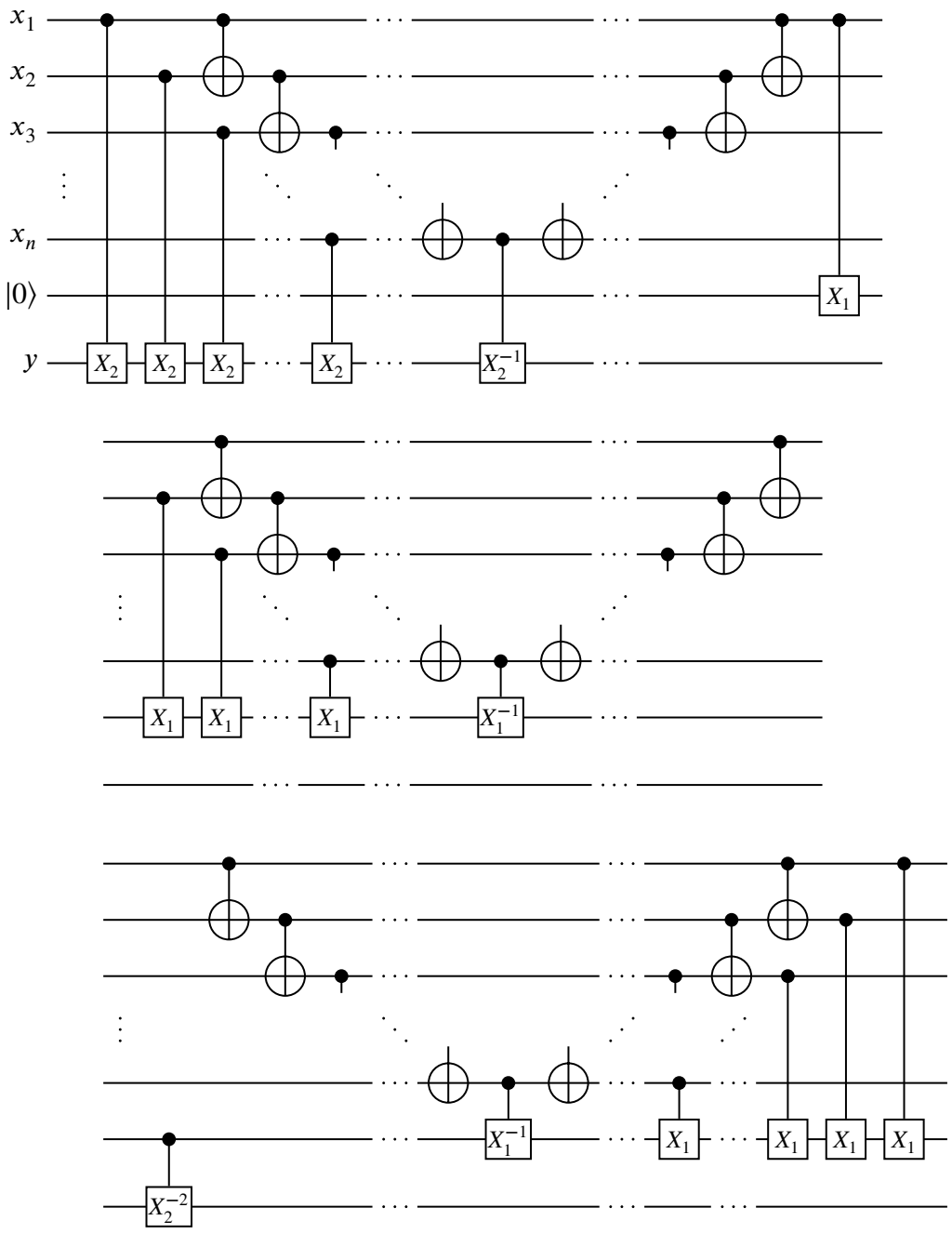


Figure 2.6: The circuit structure from Figure 2.5 fully expanded in terms of two-qubit controlled gates.

function of the form $\mathrm{S}_n^{2,3,6,7,\cdots}$, it is only active when the input weight is congruent to either 2 or 3 modulo 4. In addition, since its target gate is a X_2^{-2} gate, its contribution is -2 when active, instead of -1 as in Stage 2. From the second-to-last column of Table 2.3, we can see that the total effective number of X_2 gates is always a multiple of 4, so when reduced modulo 8, as shown in the last column, the result is always either 0 or 4. Since $X_2^0 = I$ and $X_2^4 = X$, we conclude that instances of the circuit structure from Figure 2.5 therefore performs an operation equivalent to a single NOT gate on qubit y when the input weight is 4, 5, 6, 7, 12, 13, etc., these weights being those natural numbers that are congruent to 4, 5, 6, or 7 modulo 8. Therefore, an instance of Figure 2.5 with n input qubits (not counting the ancillary qubit and qubit y) realizes the symmetric function $\mathrm{S}^{4,5,6,7,12,\cdots}(x_1, \ldots, x_n)$.

2.3.3 Operational equation for the three-stage circuit structure

Analogously to Section 2.2.3, we may condense the information presented in Table 2.3 down to a single operational equation that describes the operation of all instances of the circuit structure from Figure 2.5. We write

$$N_1 + N_2 + N_3 = N_{\text{eff}}, \tag{2.14}$$

where N_1, N_2, and N_3 are respectively the contributions that the first, second, and third stages in Figure 2.5 make to the total effective number of X_2 gates applied to qubit y. As in (2.7), N_{eff} denotes the total effective number itself. N_1, N_2, N_3, and N_{eff} therefore correspond to the second through fifth columns in Table 2.3. Equation (2.14) looks slightly different from equation (2.7)—specifically, the terms on the left-hand side are added rather than subtracted—because N_2 and N_3 now represent negative contributions to the total effective number of X_2 gates rather than the (positive) number of X_2^{-1} gates applied. Since N_1

Table 2.3: Calculation of total effective number of X_2 gates applied by instances of the circuit structure from Figure 2.5.

$\sum_{i=1}^{n} x_i$	Applied W gates			Tot. eff. num. of X_2 gates	Eff. num. of X_2 gates modulo 8	Operation applied to qubit y
	Stage 1	Stage 2	Stage 3			
0	0	0	0	0	0	$X_2^0 = I$
1	1	−1	0	0	0	$X_2^0 = I$
2	2	0	−2	0	0	$X_2^0 = I$
3	3	−1	−2	0	0	$X_2^0 = I$
4	4	0	0	4	4	$X_2^4 = X$
5	5	−1	0	4	4	$X_2^4 = X$
6	6	0	−2	4	4	$X_2^4 = X$
7	7	−1	−2	4	4	$X_2^4 = X$
8	8	0	0	8	0	$X_2^0 = I$
9	9	−1	0	8	0	$X_2^0 = I$
10	10	0	−2	8	0	$X_2^0 = I$
11	11	−1	−2	8	0	$X_2^0 = I$
12	12	0	0	12	4	$X_2^4 = X$
13	13	−1	0	12	4	$X_2^4 = X$
⋮	⋮	⋮	⋮	⋮	⋮	⋮

corresponds to the first column of Table 2.3, it is simply the input weight, $N_1 = w = \sum x_i$. N_2 and N_3 may be expressed in terms of indicator functions,

$$N_2 = -I_{\oplus}(w) \qquad \text{and} \qquad N_3 = -2I^{2,3,6,7,\cdots}(w), \tag{2.15}$$

where $I_{\oplus}(w)$ and $I^{2,3,6,7,\cdots}(w)$ are as defined before. Similarly, referring to the fifth and sixth columns of Table 2.3, we see that N_{eff} may also be expressed in terms of an indicator function when reduced modulo 8:

$$N_{\text{eff}} \equiv 4I^{4,5,6,7,12,13,\cdots}(w) \pmod 8, \tag{2.16}$$

where $I^{4,5,6,7,12,13,\cdots}$ is the indicator function corresponding to the family of symmetric functions $\mathrm{S}_n^{4,5,6,7,12,13,\cdots}$. Substituting (2.15) and (2.16) into (2.14) gives

$$w - I_{\oplus}(w) - 2I^{2,3,6,7,\cdots}(w) \equiv 4I^{4,5,6,7,12,13,\cdots}(w) \pmod 8, \tag{2.17}$$

which is analogous to (2.12).

2.3.4 Rewriting of operational equations in terms of bit-level operations on input weight

At this point, it is convenient to introduce some additional notation to concisely represent the indicator functions seen in (2.12) and (2.17). Observe that the indicator function of the exclusive-OR function, $I_{\oplus}(w) = w \bmod 2$, is periodic with period 2. Since the function alternates between 0 and 1 with each successive w, it takes on the value 0 for half of each period and takes on 1 for the other half. Next, the indicator function $I^{2,3,6,7,\cdots}(w)$ is also

periodic but with period 4. It also takes on the value 0 for half of its period and 1 for the other half: it takes on 0 for $w = 0$ and $w = 1$, takes on 1 for $w = 2$ and $w = 3$, takes on 0 for $w = 4$ and $w = 5$, and so on. Informally, this can be described as a "2-off, 2-on" pattern. Finally, $I^{4,5,6,7,12,13,\cdots}(w)$ has a period of 8 and takes on 0 for half that period and 1 for the other half, therefore following a "4-off, 4-on" pattern.

These "m-off, m-on" patterns, where m is a power of 2, have a very simple description when the input weight is represented in base two. Specifically, an indicator function following a "2^k-off, 2^k-on" pattern can be obtained as the k-th bit in the base-two representation of the input weight w, where the least significant bit is considered the zeroth bit, the next least significant bit is considered the first bit, and so on. From now on, I will denote the k-th bit of the base-two representation of w by $\mathrm{b}_k(w)$, and I will refer to b_k as the k-th *bit-extraction* function. Then each of the indicator functions appearing in (2.17) is identical to a bit-extraction function: $I_{\oplus}(w) = \mathrm{b}_0(w)$, $I^{2,3,6,7,\cdots}(w) = \mathrm{b}_1(w)$, and $I^{4,5,6,7,12,13,\cdots}(w) = \mathrm{b}_2(w)$.

Using the $\mathrm{b}_k(w)$ notation, the third, fourth, and sixth columns of Table 2.3, as well as the second and fourth columns of Table 2.1, can be represented, which is to be expected since these columns are just integer multiples of indicator functions and correspond to the terms of (2.12) and (2.17). For instance, the fourth column of Table 2.3 is given by $-2I^{2,3,6,7,\cdots}(w) = -2\mathrm{b}_1(w)$. The fifth column of Table 2.3 and the fourth column of Table 2.1, which show the total effective number of X_2 (resp. V) gates prior to reduction modulo 8 (resp. 4), cannot be easily represented with the $\mathrm{b}_k(w)$ notation, but they are still related to the base-two representation of the input weight w. The following definition introduces a notation that helps make this relationship clear.

Definition 3. For natural numbers N and k, we define $\mathrm{tr}_k(N)$, the k-th *right truncation* of

Table 2.4: Values of $\mathrm{tr}_k(N)$ and $\mathrm{b}_k(N)$ for $k = 0, 1, 2, 3$ and $0 \leq N \leq 10$.

N	$\mathrm{tr}_0(N)$	$\mathrm{b}_0(N)$	$\mathrm{tr}_1(N)$	$\mathrm{b}_1(N)$	$\mathrm{tr}_2(N)$	$\mathrm{b}_2(N)$	$\mathrm{tr}_3(N)$	$\mathrm{b}_3(N)$
0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
2	2	0	2	1	0	0	0	0
3	3	1	2	1	0	0	0	0
4	4	0	4	0	4	1	0	0
5	5	1	4	0	4	1	0	0
6	6	0	6	1	4	1	0	0
7	7	1	6	1	4	1	0	0
8	8	0	8	0	8	0	8	1
9	9	1	8	0	8	0	8	1
10	10	0	10	1	8	0	8	1

N, to be the natural number obtained by truncating to zero the last k bits in the base-two representation of N. In equation form, we have

$$\mathrm{tr}_k(N) = \sum_{i=k}^{\infty} \mathrm{b}_i(N) 2^i, \tag{2.18}$$

where all but finitely many terms of the sum on the right-hand side must be zero because the base-two representation of N can only contain finitely many nonzero bits.

The total effective numbers of X_2 and V gates shown in Tables 2.3 and 2.1 can be expressed using right truncations: in Table 2.3, the total effective number of X_2 gates is $\mathrm{tr}_2(w)$, while in Table 2.1, the total effective number of V gates is $\mathrm{tr}_1(w)$. Intuitively, $\mathrm{tr}_k(N)$ may be thought of as rounding the natural number N down to the nearest multiple of 2^k. For illustration, Table 2.4 displays the values of $\mathrm{tr}_0(N)$, $\mathrm{tr}_1(N)$, $\mathrm{tr}_2(N)$, and $\mathrm{tr}_3(N)$ for N from 1 to 10. The values of $\mathrm{b}_k(N)$ for $k = 1, 2, 3$ are also shown for comparison. We observe that $\mathrm{tr}_0(N) = N$, and that $\mathrm{tr}_k(N)$ is always a multiple of 2^k. For instance, $\mathrm{tr}_1(N)$ is always even, and $\mathrm{tr}_2(N)$ is always a multiple of 4. These properties follow directly from Definition 3,

because a natural number whose base-two representation ends in k zeros is divisible by 2^k.

We can also see from Table 2.4 that

$$\mathrm{tr}_k(N) \bmod 2^{k+1} = 2^k \mathrm{b}_k(N) \tag{2.19}$$

for k from 0 to 3. The following reasoning shows that this is true for all k. Reducing a natural number modulo 2^{k+1} is equivalent to taking only its rightmost $k+1$ bits and discarding the rest. If N is a natural number with an n-bit base-two representation $N_n N_{n-1} \ldots N_2 N_1 N_0$, then $\mathrm{tr}_k(N)$ keeps only N_k and the bits to its left, while $\mathrm{tr}_k(N) \bmod 2^{k+1}$ additionally discards all bits to the left of N_k, therefore leaving only N_k itself. In other words, the base-two representation of $\mathrm{tr}_k(N) \bmod 2^{k+1}$ is $N_k 00 \ldots$, with k zeroes, which is equal to $2^k \mathrm{b}_k(N)$.

Finally, $\mathrm{tr}_k(N)$ can also be expressed as $\mathrm{tr}_k(N) = N - (N \bmod 2^k)$. To see this, we observe that $N \bmod 2^k$ consists of the rightmost k bits of N, and so subtracting $N \bmod 2^k$ from N simply sets those bits to zero, which is exactly the definition of $\mathrm{tr}_k(N)$. Then, we can derive the identity

$$\begin{aligned} \mathrm{tr}_k(N) + 2^k &= N - (N \bmod 2^k) + 2^k \\ &= (N + 2^k) - \big((N + 2^k) \bmod 2^k\big) = \mathrm{tr}_k(N + 2^k). \end{aligned} \tag{2.20}$$

The following proposition summarizes the observations made in the preceding discussion:

Proposition 4. *For all natural numbers N and k, the following two properties hold:*

1. $\mathrm{tr}_k(N)$ *is divisible by* 2^k.
2. $\mathrm{tr}_k(N) \bmod 2^{k+1} = 2^k \mathrm{b}_k(N)$.

3. $\mathrm{tr}_k(N) + 2^k = \mathrm{tr}_k(N + 2^k)$.

We can now rewrite (2.17) using bit-extraction functions:

$$w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) \equiv 4\mathrm{b}_2(w) \pmod 8. \tag{2.21}$$

Additionally, using a right-truncation function, we can also write a variant of this equation using an exact equality rather than a congruence modulo 8:

$$w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) = \mathrm{tr}_2(w). \tag{2.22}$$

We are able to state (2.22) as an exact equality because its right-hand side corresponds to the fifth column of Table 2.3, which shows the total effective number of X_2 gates prior to reduction modulo 8. In contrast, the right-hand side of (2.21) corresponds to the sixth column of Table 2.3, which has been reduced modulo 8, so (2.21) can only be stated as a congruence. In fact, equation (2.21) can be immediately derived from (2.22) using (2.19). Equation (2.12) can be rewritten in the same way as well, giving

$$w - \mathrm{b}_0(w) \equiv 2\mathrm{b}_1(w) \pmod 4, \tag{2.23}$$

and the corresponding variant equation using an exact equality is

$$w - \mathrm{b}_0(w) = \mathrm{tr}_1(w). \tag{2.24}$$

2.3.5 Four-stage circuit using controlled-X_3 gates

It may seem that the rewriting of equations in Section 2.3.4 has not accomplished anything useful. However, expressing the equations in terms of bit-extraction and right-truncation operations makes it easier to see how they can be generalized. In addition, while the operational equations for the circuit structures shown in Figures 2.2 and 2.5 were derived after the behavior of those circuit structures had already been analyzed, it is also possible to go the other way around; *i.e.*, starting from an operational equation, we can generate a corresponding circuit structure and use the operational equation itself to analyze that circuit structure's behavior.

As an example of generating and analyzing a new circuit structure starting from an operational equation, consider the following equality, which has the same general form as (2.22) and (2.22):

$$w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) - 4\mathrm{b}_2(w) = \mathrm{tr}_3(w). \tag{2.25}$$

It may not be obvious that this equality even holds at all, but it fits into a pattern that I will prove later on to always hold, so we may simply assume its validity for now. From (2.25), we would like to generate a circuit structure that operates by applying some sequence of gates to a single target qubit, just as the circuit structures from Figures 2.2 and 2.5 operate by applying sequences of $V = X_1$ and X_2 gates and their inverses to a target qubit y. The gates applied to the target qubit will all be powers of U, which is a placeholder for a gate whose identity will be determined later. Each term on the left-hand side of (2.25) will correspond to a stage in the circuit structure to be generated, and in particular each term will represent the contribution of its corresponding stage to the total effective number of U gates applied

by the circuit, where a U^p gate makes a contribution of p.

The first term of (2.25), which is just the input weight w itself, corresponds to a cascade of controlled-U gates analogous to the initial cascades of controlled-V and controlled-X_2 gate in Figures 2.2 and 2.5, respectively. The second term on the left-hand side of (2.25), $-\mathrm{b}_0(w)$, corresponds to a U^{-1} gate that is controlled by a symmetric function with indicator function $\mathrm{b}_0(w)$, since this will contribute -1 to the total effective number of U gates when $\mathrm{b}_0(w) = 1$ and contribute 0 otherwise. But the n-input symmetric function with indicator function $\mathrm{b}_0(w)$ is just the exclusive-OR of n variables, so the second stage is just a U^{-1} gate controlled by the exclusive-OR of all inputs. Using similar reasoning, the third and fourth terms on the left-hand side of (2.25) correspond to U^{-2} and U^{-4} gates controlled by symmetric functions with indicator functions $\mathrm{b}_1(w)$ and $\mathrm{b}_2(w)$, respectively. These symmetric functions are also familiar: they are $\mathrm{S}_n^{2,3,6,7,\cdots}$ and $\mathrm{S}_n^{4,5,6,7,12,13,\cdots}$, whose realizations were demonstrated in Sections 2.2 and 2.3.2, respectively. The circuit structure that results from combining all four stages is shown in Figure 2.7. Like Figure 2.5, this circuit structure is recursive because it uses the circuit structures from Figures 2.2 and 2.5 as subcircuits but is itself a larger variant of those structures.

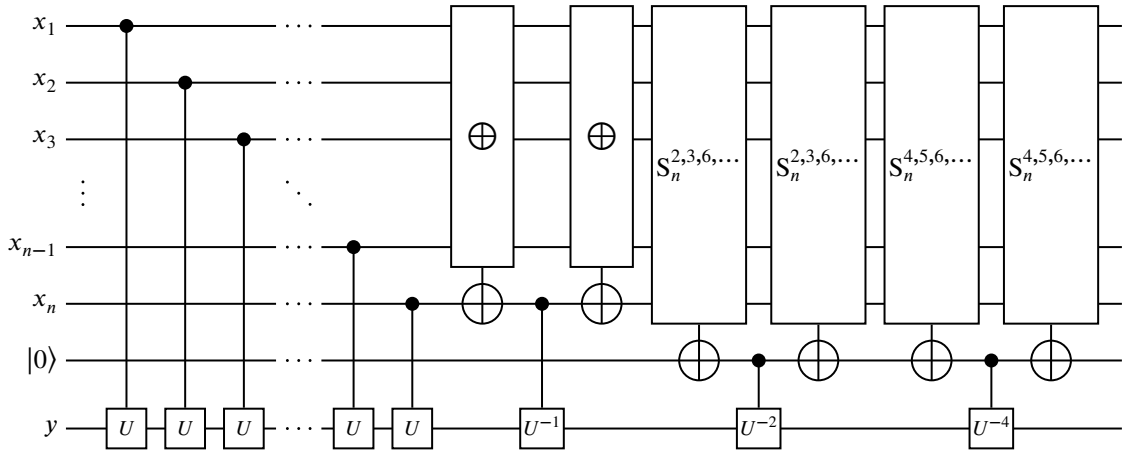


Figure 2.7: Four-stage circuit structure generated from eq. (2.25).

What we have done is essentially the reverse of what was done in Sections 2.2.3 and 2.3.3: instead of deriving an operational equation based on the analysis of a circuit structure, we have created a circuit structure based on an operational equation. The right-hand side of the operational equation (2.25) then immediately tells us the behavior of the newly-created circuit structure; namely, the total effective number of U gates applied to qubit y will be $\mathrm{tr}_3(w)$. From Proposition 4 we know that $\mathrm{tr}_3(w)$ is always a multiple of 8. Therefore, to create permutative instances of Figure 2.7, we should choose $U = X_3$, an eighth-root-of-NOT gate. With this choice, we then have $U^{16} = I$, so the total effective number of U gates can be reduced modulo 16. Using (2.19) again, we see that $\mathrm{tr}_3(w) \bmod 16 = 8\mathrm{b}_3(w)$ and substituting into (2.25) gives

$$w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) - 4\mathrm{b}_2(w) \equiv 8\mathrm{b}_3(w) \pmod{16}. \tag{2.26}$$

Equation (2.26) allows us to deduce the form of the symmetric functions realized by instances of Figure 2.7 with $U = X_3$. When $\mathrm{b}_3(w) = 1$, the total effective number of U gates applied to y is congruent to 8 modulo 16, so the operation performed on y is equivalent to a NOT gate (since X_3 is an eighth-root-of-NOT gate). When $\mathrm{b}_3(w) = 0$, the total effective number is congruent to 0 modulo 16, so no overall change is made to y. Hence, the symmetric functions realized have indicator function $\mathrm{b}_3(w)$ and are therefore of the form $\mathrm{S}_n^{8,9,10,\ldots,15,24,25,\ldots}$. We can arrive at the same conclusion by calculating the total effective number of U gates using a table analogous to Tables 2.1 and 2.3. Such a table is unnecessary at this point, because all the information contained therein is already represented by (2.25) and (2.26). Nevertheless, it is included here as Table 2.5 for completeness.

Table 2.5: Calculation of total effective number of U gates applied by instances of the circuit structure from Figure 2.7. When $U = X_3$, then an effective number of 8 U gates is equivalent to applying a single NOT gate to qubit y.

$\sum_{i=1}^{n} x_i$	Applied U gates				Tot. eff. num. of U gates	Eff. num. of U gates modulo 16
	Stage 1	Stage 2	Stage 3	Stage 4		
0	0	0	0	0	0	0
1	1	−1	0	0	0	0
2	2	0	−2	0	0	0
3	3	−1	−2	0	0	0
4	4	0	0	−4	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
7	7	−1	−2	−4	0	0
8	8	0	0	0	8	8
9	9	−1	0	0	8	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	−1	−2	−4	8	8
16	16	0	0	0	16	0
17	17	−1	0	0	16	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
23	23	−1	−2	−4	16	0
24	24	0	0	0	24	8
25	25	−1	0	0	24	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮

2.4 General circuit structure with arbitrary number of stages

2.4.1 General form of realized symmetric functions and operational equations

All of the symmetric functions realized so far have indicator functions that are identical to a bit-extraction function. Specifically, $S_n^{2,3,6,7,\cdots}$ and $S_n^{4,5,6,7,12,13,\cdots}$ have indicator functions $b_1(w)$ and $b_2(w)$, respectively, as observed in Section 2.3.4, and the functions $S_n^{8,9,10,\ldots,15,24,25,\cdots}$ realized in Section 2.3.5 have the indicator function $b_3(w)$. The following definition introduces a name and systematic notation for these symmetric functions

Definition 5. Given natural numbers k and n, the n-input *dyadic indicator-periodic symmetric function* (DIPS[5]) with *order* k and zero offset, which we denote by $S_n^{\mathrm{DIP}(k,0)}$, is the n-input symmetric function with indicator function

$$I^{\mathrm{DIP}(k,0)}(w) = b_k(w). \tag{2.27}$$

The name "dyadic indicator-periodic symmetric function" describes the "m-off, m-on" nature of these functions, as previously mentioned in Section 2.3.4: "indicator-periodic" means that the indicator function of a DIPS is always periodic, while "dyadic" means that the period is always a power of 2, with m being exactly half of the period. The meaning of "zero offset" and the 0 in DIP$(k, 0)$ will be discussed in Chapter 3; before then, I will omit the phrase "zero offset" entirely when describing DIPS. The symmetric functions realized in Sections 2.2, 2.3.2, and 2.3.5 are zero-offset DIPS with orders 1, 2, and 3, respectively, which in the notation of 5 are $S_n^{\mathrm{DIP}(1,0)}$, $S_n^{\mathrm{DIP}(2,0)}$, and $S_n^{\mathrm{DIP}(3,0)}$. The order-0 DIPS $S_n^{\mathrm{DIP}(0,0)}$ has indicator function $b_0(w)$ and is therefore just the exclusive-OR function of n variables.

[5]The acronym "DIPS" is both singular and plural, as the "S" can stand for "symmetric function" or "symmetric functions". I will thus use both phrases such as "the DIPS" and "collection of DIPS".

The operational equations (2.24), (2.22), and (2.25) also form a series that follows a clear pattern. Specifically, it appears that

$$w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) - \cdots - 2^{k-1}\mathrm{b}_{k-1}(w) = \mathrm{tr}_k(w) \tag{2.28}$$

for any nonnegative integer k. If true, then a whole infinite series of circuit structures can be generated from (2.28), in the same way that the circuit structure from Figure 2.7 was generated from the operational equation (2.25).

One straightforward way to see that (2.28) is always true is to observe that $2^i\mathrm{b}_i(w)$ simply discards all bits of w other than the i-th one, while keeping the i-th bit in its original position with trailing zeroes. In other words, if w has a $j+1$-bit base-two representation $w_j w_{j-1} \ldots w_2 w_1 w_0$, then $2^i\mathrm{b}_i(w)$ is given by $w_i 0 \ldots 00$, with i trailing zeroes. Then, the left-hand side of (2.28) simply subtracts away the rightmost k bits of w, setting them to zero and leaving only bits from the k-th bit and leftwards, which is exactly what $\mathrm{tr}_k(w)$ does. Hence, (2.28) does indeed hold for all natural numbers k.

While the foregoing proof of (2.28) is simple and straightforward, it does not showcase an important connection between (2.28) and the recursive nature of the circuit structures from Figures 2.5 and 2.7. This deeper connection will be used in Chapter 3. To illustrate it, I now consider the left-hand side of (2.28) one term at a time, which corresponds to building up a circuit structure such as the ones from Figures 2.5 and 2.7 one stage at a time.

2.4.2 Representation of adding one stage in the operational equation

The circuit structures from Figures 2.2, 2.5, and 2.7 form a series with each one being an extension of the one before it. Specifically, the circuit structure shown in Figure 2.5 was obtained by extending the one from Figure 2.2 with an additional stage. The circuit structure

from Figure 2.7 was not derived this way, instead being generated directly from an operational equation, but comparing Figure 2.7 with Figure 2.5 makes it clear that Figure 2.7 can also be thought of as an extension of Figure 2.5. The corresponding operational equations likewise form such a series, as do Tables 2.1, 2.3, and 2.5. The first two terms on the left-hand side of (2.22) are the same as the left-hand side of (2.24), and the first three terms on the left-hand side of (2.25) are the same as the left-hand side of (2.22). Similarly, the columns under the "Stage 1" and "Stage 2" headings in Table 2.3 are identical to the corresponding columns of Table 2.1, and the columns under the "Stage 1", "Stage 2", and "Stage 3" headings in Table 2.5 are identical to the corresponding columns of Table 2.3.

The process of extending an existing circuit with a new stage can itself be represented by an operational equation. For instance, consider the process of adding an additional stage to the circuit structure in Figure 2.5 to create the one in Figure 2.7. In Table 2.5, we can replace the columns for the first three stages with a single column representing the combined contribution of these three stages, because they are the same as in Figure 2.5 (other than altered target gates on qubit y) and their behavior is therefore already known from Table 2.3. This gives Table 2.6, where the second column represents the combined contribution of the first three stages in Figure 2.7 and is taken from the second-to-last column of Table 2.3.

Table 2.6 clearly shows how the behavior of the circuit structure from Figure 2.5, represented by the second column, is modified by the addition of a new stage, represented by the third column, to create a new behavior, represented by the last column. Whereas the original circuit structure from Figure 2.5 applies a number of gates to the target qubit y that is always a multiple of 4, the addition of the new stage causes the effective number of applied gates to instead always be a multiple of 8. Table 2.6 has an analogue as an operational equation,

Table 2.6: Calculation of total effective number of U gates applied by instances of the circuit structure from Figure 2.7, treating Figure 2.7 as an extension of Figure 2.5 and explicitly using the already-known behavior of the latter.

$\sum_{i=1}^{n} x_i$	Applied U gates		Total effective number of U gates
	Stages 1–3	Stage 4	
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	4	−4	0
⋮	⋮	⋮	⋮
7	4	−4	0
8	8	0	8
⋮	⋮	⋮	⋮
11	8	0	8
12	12	−4	8
⋮	⋮	⋮	⋮
15	12	−4	8
16	16	0	16
⋮	⋮	⋮	⋮

which can be derived by substituting (2.22) into (2.25):

$$\mathrm{tr}_2(w) - 4\mathrm{b}_2(w) = \mathrm{tr}_3(w). \tag{2.29}$$

Equation (2.29) clearly shows that the existing behavior of the circuit structure from Figure 2.5—as expressed by the $\mathrm{tr}_2(w)$ term, which gives the effective number of applied gates as a function of input weight—is modified by the new stage—which contributes $-4\mathrm{b}_2(w)$ to the total effective number of applied gates—to create a new behavior expressed by the $\mathrm{tr}_3(w)$ term.

The preceding discussion can be generalized in the following way. Suppose that we have an existing circuit structure with the following properties:

1. It operates on a variable number n of input qubits x_1 through x_n together with a result qubit y. Any number of ancilla qubits may also be used.

2. All qubits other than y are restored to their original states at the end of the circuit.

3. All gates in the structure that can affect the state of qubit y are controlled gates whose target gates are powers of a single gate U. Therefore, in any circuit derived from this structure, the effective operation performed on qubit y can always be expressed as a number of U gates.

4. The effective number of U gates applied to qubit y depends only on the input weight, $w = \sum x_i$, and is given by $\mathrm{tr}_k(w)$ for some fixed k.

Properties 1 and 2 are just another way of stating that any circuit derived from the structure satisfying these properties acts as a controlled gate whose target qubit is y and whose control function takes x_1 through x_n as inputs. All of the circuit structures shown in this chapter

for realizing DIPS satisfy properties 3 and 4 as well. Conversely, given a circuit structure satisfying the above conditions, any instance of the structure with $U = X_k$ will realize an order-k DIPS. If $U = X_k$, then $U^{2^{k+1}} = I$, meaning that the total effective number of U gates applied to qubit y can be reduced modulo 2^{k+1}. From property 4 above, and using (2.19), we see that this effective number is just $\mathrm{tr}_k(w) \bmod 2^{k+1} = 2^k \mathrm{b}_k(w)$. Since U is a 2^k-th-root-of-NOT gate, $U^{2^k} = X$, so the overall operation performed on y is equivalent to a NOT gate when $\mathrm{b}_k(w) = 1$. It then follows from Definition 5 that the circuit realizes an order-k DIPS.

Now, given a circuit structure satisfying all of the properties above, I claim that it is always possible to extend the structure, appending more gates at its end, to obtain a new structure that also satisfies all of the properties above but with k increased by one. Specifically, the existing circuit structure should be extended as follows:

1. Add an additional ancilla qubit, initialized to 0.

2. Use another copy of the existing circuit structure with $U = X_k$ to realize an order-k DIPS, as described above. This copy targets the new ancilla qubit so that the output of the realized DIPS is stored on that qubit.

3. Add a controlled-U^{-2^k} gate targeting y, with the new ancilla qubit as its control qubit.

4. Repeat step 2 to generate a mirror circuit that restores the ancilla qubit to its original value of 0.

The new circuit structure resulting from the above procedure trivially satisfies property 1, and it also satisfies property 2 since the additional ancilla bit is restored to its original value using a mirror circuit. The controlled gate added in step 3 is the only one that targets qubit y, and its target gate is indeed a power of U, so the new circuit structure also satisfies property

Table 2.7: Generalization of Table 2.6 to the scenario of adding a new stage to an existing circuit structure, where the existing structure applies a total effective number of $\mathrm{tr}_k(w)$ U gates.

$w = \sum_{i=1}^{n} x_i$	Applied U gates		Total effective number of U gates
	Existing circuit $\mathrm{tr}_k(w)$	New stage $-2^k \mathrm{b}_k(w)$	
0	0	0	0
1	0	0	0
⋮	⋮	⋮	⋮
$2^k - 1$	0	0	0
2^k	2^k	-2^k	0
⋮	⋮	⋮	⋮
$2^{k+1} - 1$	2^k	-2^k	0
2^{k+1}	2^{k+1}	0	2^{k+1}
⋮	⋮	⋮	⋮
$3 \cdot 2^k - 1$	2^{k+1}	0	2^{k+1}
$3 \cdot 2^k$	$3 \cdot 2^k$	-2^k	2^{k+1}
⋮	⋮	⋮	⋮

3. To see that the new structure satisfies property 4, we consult Table 2.7 and observe that the second column of this table is just $\mathrm{tr}_k(w)$, so it represents the effective number of U gates applied to qubit y by any circuit derived from the given, preexisting structure. The third column shows the effective number of U gates applied by the controlled gate from step 3 of the above procedure. Since this controlled gate is controlled by a DIPS of order k and has target gate U^{-2^k}, the effective number of U gates is $-2^k \mathrm{b}_k(w)$. Adding the effective number of U gates contributed by the existing circuit structure and the new stage gives the last column of Table 2.7, which we can see is just $\mathrm{tr}_{k+1}(w)$. Therefore, the new circuit structure satisfies property 4 with k replaced by $k + 1$.

In equation form, the information presented in Table 2.7 can be represented as

$$\mathrm{tr}_k(w) - 2^k \mathrm{b}_k(w) = \mathrm{tr}_{k+1}(w). \tag{2.30}$$

The following proposition formalizes (2.30) as a result with an explicit proof.

Proposition 6. *For all natural numbers N and k, the following equality holds:*

$$\mathrm{tr}_k(N) - 2^k \mathrm{b}_k(N) = \mathrm{tr}_{k+1}(N), \tag{2.31}$$

where, as before, $\mathrm{b}_k(N)$ *denotes the k-th bit of N,* i.e.*, the bit with a positional value of* 2^k.

Proof. If N has the $n+1$-bit base-two representation $b_n b_{n-1} \ldots b_2 b_1 b_0$, then $\mathrm{tr}_k(N)$ has the base-two representation

$$\mathrm{tr}_k(N) = b_n \ldots b_{k+1} b_k 00 \ldots 0. \tag{2.32}$$

(If $n < k$, then N's base-two representation can simply be padded on the left with zeros until $n = k$, in which case (2.32) simply gives $\mathrm{tr}_k(N) = 0$.) Meanwhile, $\mathrm{b}_k(N) = b_k$ by definition, so $2^k \mathrm{b}_k(N)$ has the base-two representation $b_k 00 \ldots 0$, with k zeros. Performing subtraction in base-two arithmetic, we see that $\mathrm{tr}_k(N) - 2^k \mathrm{b}_k(N)$ has the base-two representation $b_n \ldots b_{k+1} 000 \ldots 0$ and is therefore equal to $\mathrm{tr}_{k+1}(N)$. □

Using Proposition 6 we can easily prove (2.28) using induction: the case $k = 0$ reduces to $w = \mathrm{tr}_0(w)$, which was already observed in Section 2.3.4, while if (2.28) is true for some k, then we can add $\mathrm{b}_k(w)$ to both sides to obtain

$$\begin{aligned} w - \mathrm{b}_0(w) - 2\mathrm{b}_1(w) - \cdots - 2^{k-1}\mathrm{b}_{k-1}(w) - 2^k \mathrm{b}_k(w) \\ = \mathrm{tr}_k(w) - 2^k \mathrm{b}_k(w) = \mathrm{tr}_{k+1}(w) \end{aligned} \tag{2.33}$$

where Proposition 6 is used for the second equality. Eq. (2.28) is therefore true for $k+1$ as well, completing the induction step.

2.4.3 General circuit structure for realizing DIPS

Figure 2.8 shows how (2.30) translates to circuit form. This figure illustrates the earlier described procedure of adding an additional stage to an existing circuit structure. The first gate on the left of figure represents the existing circuit structure, which is assumed to apply an effective number of $\mathrm{tr}_k(w)$ U gates to qubit y. I have introduced the new notation of a double line connecting the control function $\mathrm{tr}_k(w)$ to its target gate U to indicate that U may be effectively applied multiple times to y. This can be thought of as a generalization of a controlled gate: instead of applying the target gate U either zero or one time to the target qubit y based on the output of a Boolean-valued control function, the target gate is effectively applied a number of times determined by an integer-valued control function, which in this case is $\mathrm{tr}_k(w)$.

The remaining part of Figure 2.8 is the new stage being added, which corresponds to the $-2^k \mathrm{b}_k(w)$ term in (2.30). This stage consists of a U^{-2^k} gate controlled by the function $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$, which has indicator function $\mathrm{b}_k(w)$. In accordance with (2.30), the whole circuit structure shown in Figure 2.8 then applies a total effective number of U gates equal to $\mathrm{tr}_{k+1}(w)$.

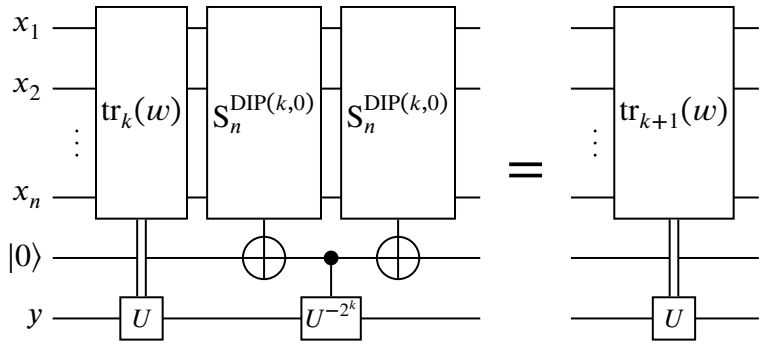


Figure 2.8: Recursion relation used to realize a DIPS of order $k+1$ from one of order k.

Figure 2.9 shows a circuit structure generated from (2.28) in the same way that the circuit

structure from Figure 2.7 was generated from the operational equation (2.25). This figure uses the same double-lined control notation from Figure 2.8 as an abbreviation for a cascade of controlled-U gates like the one at the start of the circuit structure shown in Figure 2.7. The control function is labeled as "$\sum$" because the number of U gates applied by such a cascade is equal to the input weight, $\sum x_i$. The proof of (2.28) from (2.30) by induction, given at the end of Section 2.4.2, can also be translated into circuit terms to show that, for all k, the circuit structure from Figure 2.9 applies a total effective number of $\mathrm{tr}_k(w)$ U gates to qubit y. Specifically, the base case $k = 0$ corresponds to a single cascade of controlled-U gates in which the number of U gates applied to y is $w = \mathrm{tr}_0(w)$. The induction step then corresponds to using the identity represented by Figure 2.8 to show that, if all instances of Figure 2.9 for some value of k operate as claimed, then all instances with k increased by one also operate as claimed.

For clarity and future reference, the following theorem states the final result of the preceding reasoning.

Theorem 7. *For any positive integer k, an instance of the circuit structure from Figure 2.9 applies to qubit y a total effective number of* $\mathrm{tr}_k(w)$ *U gates, where w is the input weight. In other words, the overall operation performed on qubit y will be $U^{\mathrm{tr}_k(w)}$.*

As previously discussed in Section 2.4.2, if we take $U = X_k$ in Figure 2.9, then the total effective number of U gates can be reduced modulo 2^{k+1}, and from (2.19) we then see that this total effective number can be expressed as $2^k \mathrm{b}_k(w)$. Since U is a 2^k-th-root-of-NOT, any instance of the resulting circuit structure will therefore invert qubit y when $\mathrm{b}_k(w) = 0$, meaning that such an instance realizes the symmetric function $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$. We therefore see that instances of Figure 2.9 are able to realize all functions of the form $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$ for any positive n and k. In fact, k can even be zero: in this case, $\mathrm{S}_n^{\mathrm{DIP}(k,0)} = \mathrm{S}_n^{\mathrm{DIP}(0,0)}$ is just the

exclusive-OR of n variables, and Figure 2.9 reduces to just its first stage consisting of a cascade of controlled-U gates. Since we take $U = X_k$, U will in fact just be an inverter for $k = 0$, so that the resulting instance of Figure 2.9 is a cascade of CNOT gates, which does indeed realize the exclusive-OR of x_1 through x_n.

Corollary 8. *For any nonnegative integer k and positive integer n, the symmetric function* $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$ *can be realized by an instance of Figure 2.9 with the given k and n, and with* $U = X_k$.

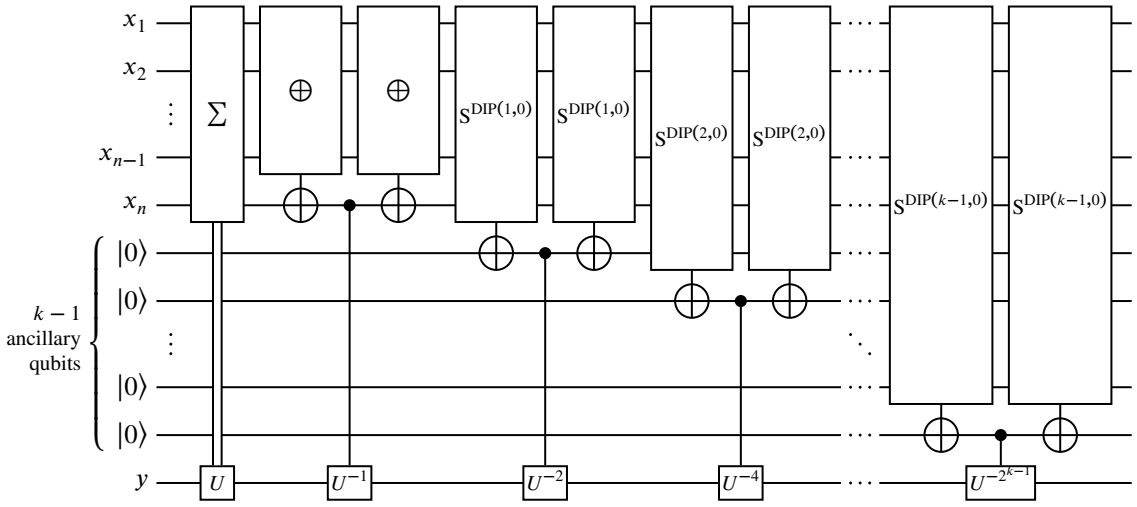


Figure 2.9: General circuit structure for an order-k DIPS.

2.5 Conclusion

In this chapter, I introduced the following:

- The general idea of constructing a circuit with a single output qubit y in which only one type of gate, plus its inverse and their powers, act on y. These types of circuits can be analyzed by simply counting the effective number of gates applied to y.

- The idea of considering such circuits to be a generalization of controlled gates. Like a controlled gate, such circuits may or may not apply a target gate to the target qubit y depending on the value of a control function. Unlike a controlled gate, the target gate can be applied multiple times or even an effective negative number of times (corresponding to applying the gate's inverse). The control function determines the effective number of times the target gate is applied, so its output may be any integer instead of just a Boolean value as in a controlled gate. In particular, I introduced circuits with a control function dependent on the base-two representation of the input weight. Chapter 3 will make further use of such circuits.

- The new concept of *dyadic indicator-periodic symmetric functions* (DIPS), which are denoted $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$.

- A recursive method for realizing DIPS for any number of inputs, which makes use of circuits of the above-described type.

As a complete example of the method for realizing DIPS, Figure 2.10 demonstrates the realization of the nine-variable symmetric function $\mathrm{S}_9^{8,9} = \mathrm{S}_9^{\mathrm{DIP}(3,0)}$. This circuit is a fully-expanded instance, of the circuit structure from Figure 2.7 with $n = 9$, Figure 2.7 itself being a particular case of Figure 2.9 with $k = 3$ and $U = X_3$. To improve the legibility of what would otherwise be very small labels in Figure 2.10, each of the target gates on qubit y is labeled with just a single integer, which represents a power of X_3. In other words, a target gate labeled p represents a X_3^p gate, so for instance the label -2 represents a $X_3^{-2} = X_2^{-1}$ gate.

From this figure, it is apparent that the recursive expansion causes the final circuit to contain a large number of gates. However, it can also be seen that there are many opportunities for CNOT gates to be canceled, which could dramatically reduce the number

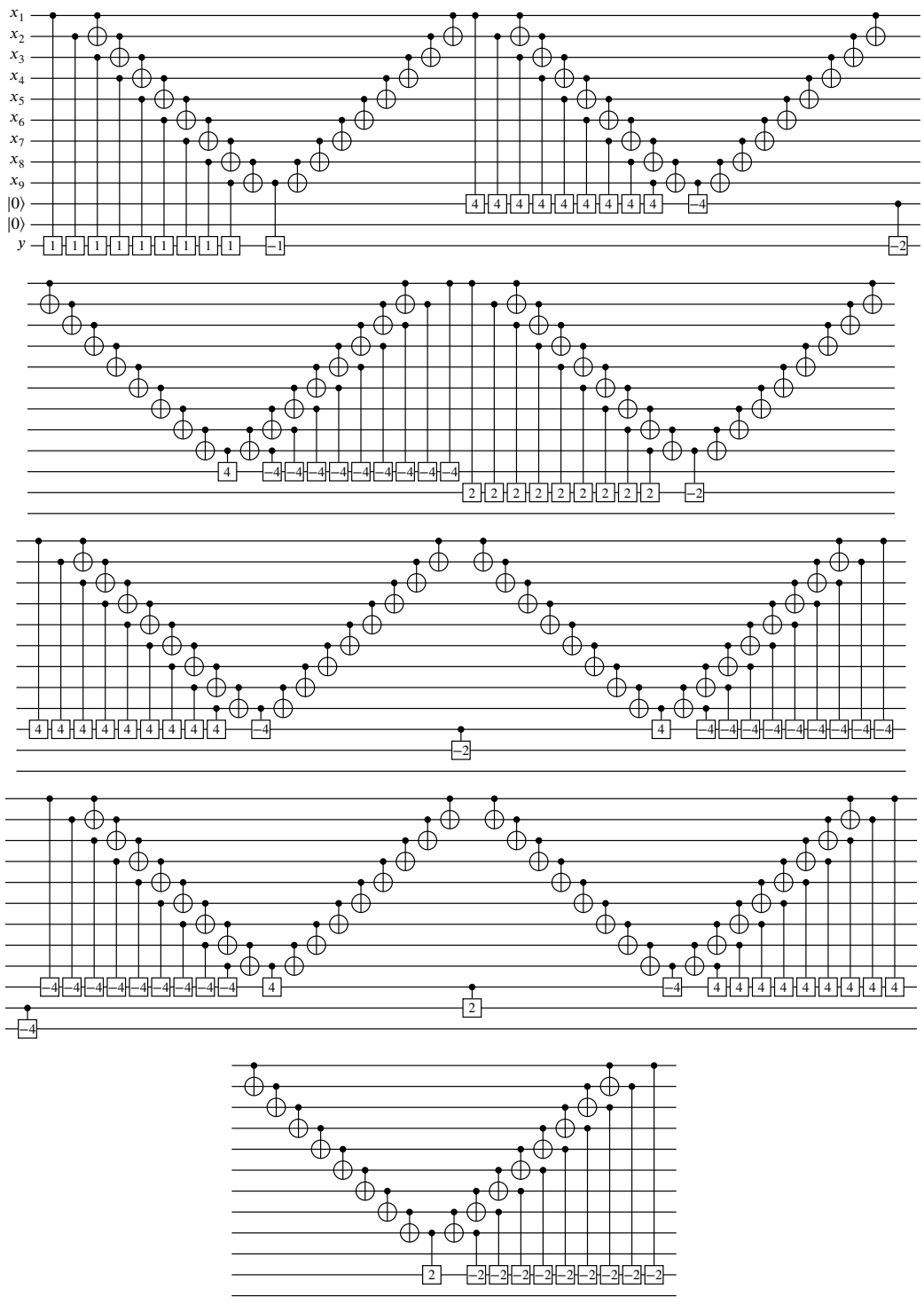


Figure 2.10: Realization of the symmetric function $S_9^{8,9}$ using the method presented in this chapter.

of gates. Nevertheless, it is unclear how this cancellation can be performed optimally in the general case for any circuit of the form shown in Figure 2.9, due to the multiple levels of recursion. In the next chapter, I show that circuits such as the one in Figure 2.10 may indeed be drastically simplified to ones with much fewer gates. I also use the concepts introduced in this chapter to create realizations for a larger class of symmetric functions.

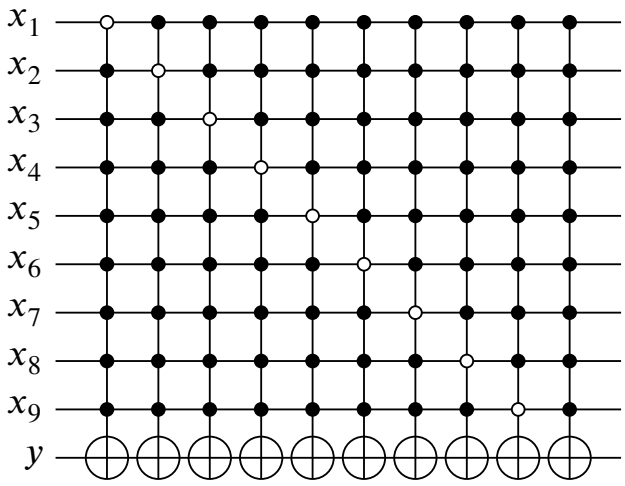


Figure 2.11: A straightforward, naive realization of the symmetric function $S_9^{8,9}$ using multiple-control Toffoli gates.

By way of comparison, Figure 2.11 shows how the same symmetric function, $S_9^{8,9}$, may be realized using multiple-control Toffoli gates in the most straightforward and naive possible manner. This realization can be obtained by simply enumerating all possible input patterns with a weight of 8 or 9: there are nine possible weight-8 input patterns with eight 1s and one 0, and one weight-9 input pattern which simply consists of all inputs set to 1. In equation form, these input patterns allow the symmetric function to be expressed as

$$S^{8,9}(x_1, \ldots, x_9) = \bigoplus_{1 \le i \le 9} \left(\neg x_i \wedge \bigwedge_{\substack{1 \le j \le 9 \\ j \ne i}} x_j \right) \oplus \bigwedge_{1 \le i \le 9} x_i, \tag{2.34}$$

the exclusive-OR of ten logical-AND terms that are realized by the ten 9-control Toffoli gates of Figure 2.11. Using Maslov and Dueck's quantum cost values for multiple-control Toffoli gates [19], if we assume that no more than two ancillary qubits (the amount used in Figure 2.10) are allowed, then each such 9-control Toffoli gate has a quantum cost of 154, meaning that the 9-control Toffoli gate must be implemented using a sequence of 154 two-qubit controlled gates. Therefore, the whole circuit in Figure 2.11 requires 1540 two-qubit controlled gates. In contrast, the circuit in Figure 2.10 requires only 236 two-qubit controlled gates, even without making any of the obvious simplifications such as cancellation of neighboring CNOT gates.

Chapter 3

Realization of dyadic indicator-periodic symmetric functions with arbitrary offset

The main result from Chapter 2 was a method for realizing an infinite family of symmetric functions, the zero-offset dyadic indicator-periodic symmetric functions. However, these functions comprise only an infinitesimally small fraction of all possible symmetric functions. Therefore, the immediate practical utility of this method is limited, even if one is considering only symmetric functions, because an arbitrarily chosen symmetric function is very unlikely to be a member of this family.

The objective of this chapter is to demonstrate that the method of Chapter 2 can, with some minor modifications, be extended to realize a significantly larger family of symmetric functions. As with the symmetric functions whose realizations were demonstrated in Chapter 2, these new functions have the property that their indicator functions are periodic with the period being a power of two. However, unlike the symmetric functions discussed in Chapter 2, the indicator functions may now start at any point within their period, which I call the offset of the symmetric function.

I will show that any symmetric function with the properties described above can be realized using circuit structures very similar to those demonstrated in Chapter 2, with the only difference being that some of controlled rotation gates in the structure may be replaced with their inverses. Furthermore, the pattern of rotation directions can be determined from

the base-two expansion of a symmetric function's offset.

3.1 Motivating examples

3.1.1 Modification of circuit structure with altered rotation signs

The realization method presented in Chapter 2 began with the circuit structure shown in Figure 2.2. Here, I consider a modification to this circuit structure that was not considered in Chapter 2: replacement of the controlled-$V^\dagger$ gate with a controlled-V gate, which produces the circuit structure shown in Figure 3.1.

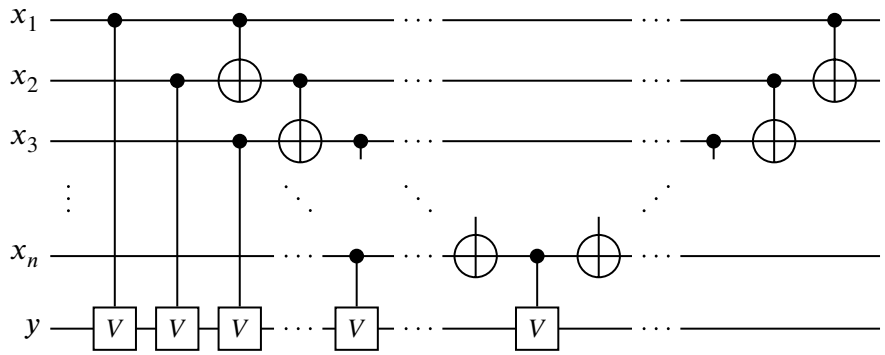


Figure 3.1: Result of replacing the controlled-$V^\dagger$ gate in Figure 2.2 with a controlled-V gate.

As in Chapter 2, we may analyze this circuit structure by first counting the number of V gates applied in each stage of the circuit and then adding the contributions from both stages to obtain a total effective number of V gates for each possible input pattern weight. Table 3.1 shows the result. From this table, we can see that all instances of the circuit structure in Figure 3.1 are permutative, because the total effective number of V gates is always a multiple of 2. In fact, the total effective number of V gates applied to qubit y is given by $N_{\text{eff}} = \text{tr}_1(w + 1)$. In comparison, for the original circuit structure introduced in Chapter 2, shown in Figure 2.2, the total effective number of V gates was given by $N_{\text{eff}} = \text{tr}_1(w)$.

Table 3.1: Calculation of total effective number of V gates applied by instances of the circuit structure from Figure 3.1.

$\sum_{i=1}^{n} x_i$	Applied V gates		Tot. eff. num. of V gates	Eff. num. of V gates modulo 4	Operation applied to qubit y
	Stage 1	Stage 2			
0	0	0	0	0	I
1	1	1	2	2	$V^2 = X$
2	2	0	2	2	$V^2 = X$
3	3	1	4	0	I
4	4	0	4	0	I
5	5	1	6	2	$V^2 = X$
6	6	0	6	2	$V^2 = X$
⋮	⋮	⋮	⋮	⋮	⋮

Therefore, the effect of replacing the last controlled-$V^\dagger$ gate in Figure 2.2 with a V gate is to reduce the input weight required to achieve any given N_{eff} by 1.

The information contained in Table 3.1 can be represented as an operational equation expressing the relationship between the number of V gates applied in each stage, exactly as in (2.24):

$$w + \mathrm{b}_0(w) = \mathrm{tr}_1(w+1). \tag{3.1}$$

Since $V^4 = I$, we can reduce $N_{\text{eff}} = \mathrm{tr}_1(w+1)$ modulo 4, giving $\mathrm{tr}_1(w+1) \bmod 4 = 2\mathrm{b}_1(w+1)$. Therefore, an instance of Figure 3.1 applies the equivalent of two V gates to qubit y when $\mathrm{b}_1(w+1) = 1$. This occurs when w is congruent to either 1 or 2 modulo 4, and the first few values of w satisfying that condition are 1, 2, 5, 6, etc. The family of symmetric function realized by instances of Figure 3.1 is hence $\mathrm{S}_n^{1,2,5,6,\cdots}$. Since the indicator function, $\mathrm{b}_1(w+1)$, of this family of symmetric functions is just a shifted $\mathrm{b}_1(w)$ function, it still exhibits the same "2-on, 2-off" property previously noted in Chapter 2.

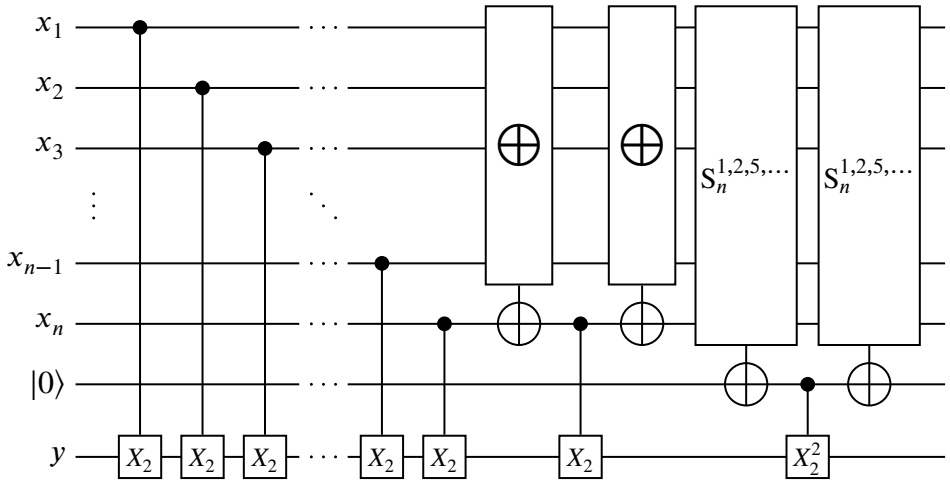


Figure 3.2: Extended variant of Figure 3.1 with V replaced by X_2, analogous to Figure 2.5.

3.1.2 Use of various combinations of rotation signs in circuits with three and four stages

The family of symmetric functions realized by instances of the circuit structure from Figure 3.1, $S_n^{1,2,5,6,\cdots}$, is not really new because it can be obtained as the exclusive-OR of $S_n^{2,3,6,7,\cdots}$ with $S_n^{1,3,5,7,\cdots}$, where $S_n^{1,3,5,7,\cdots}$ is itself an exclusive-OR of all its input variables. However, we can extend the circuit structure of Figure 3.1 by analogy with the procedure used in Section 2.3.2, replacing the V gates of Figure 3.1 with X_2 gates and adding a third stage controlled by a symmetric function with a known realization. In Section 2.3.2, specifically as shown in Figure 2.5, the symmetric function controlling the third stage was $S_n^{2,3,6,7,\cdots}$, but here we will use $S_n^{1,2,5,6,\cdots}$ instead, for a reason that will be discussed shortly. This function has a known realization, that being just demonstrated in Figure 3.1. The target gate of the third stage was a $X_2^{-2} = V^\dagger$ gate in Figure 2.5, but here we replace it with X_2^2 by analogy with the change from Figure 2.2 to 3.1, where the $V^\dagger$ gate was replaced with V.

Figure 3.2 shows the result of the changes described in the previous paragraph, and Table 3.2 shows the corresponding calculation of the total effective number of X_2 gates

Table 3.2: Calculation of total effective number of X_2 gates applied by instances of the circuit structure from Figure 3.2.

$\sum_{i=1}^{n} x_i$	Applied X_2 gates			Tot. eff. num. of X_2 gates	Eff. num. of X_2 gates modulo 8	Operation applied to qubit y
	Stage 1	Stage 2	Stage 3			
0	0	0	0	0	0	I
1	1	1	2	4	4	$X_2^4 = X$
2	2	0	2	4	4	$X_2^4 = X$
3	3	1	0	4	4	$X_2^4 = X$
4	4	0	0	4	4	$X_2^4 = X$
5	5	1	2	8	0	I
6	6	0	2	8	0	I
7	7	1	0	8	0	I
8	8	0	0	8	0	I
9	9	1	2	12	4	$X_2^4 = X$
10	10	0	2	12	4	$X_2^4 = X$
11	11	1	0	12	4	$X_2^4 = X$
⋮	⋮	⋮	⋮	⋮	⋮	⋮

applied to y. Again following the same reasoning used in Chapter 2, we reduce the total effective number of X_2 gates modulo 8 because $X_2^8 = I$. From the last column of Table 3.2, we see that an instance of Figure 3.2 inverts y when the input weight is in the list 1, 2, 3, 4, 9, 10, 11, etc., so it realizes the symmetric function $\mathrm{S}_n^{1,2,3,4,9,10,11,\ldots}$.

A operational equation similar to (3.1), and also analogous to the operational equations (2.24), (2.22), and (2.25) introduced in Chapter 2, can now be written for Figure 3.2:

$$w + \mathrm{b}_0(w) + 2\mathrm{b}_1(w+1) = \mathrm{tr}_2(w+3), \tag{3.2}$$

where the terms in this equation correspond to the second through fifth columns of Table 3.2. Reducing the right-hand side of (3.2) modulo 8 gives $\mathrm{tr}_2(w+3) \bmod 8 = 4\mathrm{b}_2(w+3)$, which corresponds to the sixth (second-to-last) column of Table 3.2. Therefore, the symmetric

Table 3.3: Comparison of total effective number of X_2 gates applied by instances of the circuit structure from Figure 3.2 for third-stage control functions $S_n^{2,3,6,7,\cdots}$ vs. $S_n^{1,2,5,6,\cdots}$

$\sum_{i=1}^{n} x_i$	Applied gates from first two stages	With Stage 3 control func. $S_n^{2,3,6,7,\cdots}$		With Stage 3 control func. $S_n^{1,2,5,6,\cdots}$	
		Stage 3 applied gates	Tot. eff. num. of gates	Stage 3 applied gates	Tot. eff. num. of gates
0	0	0	0	0	0
1	2	0	2	2	4
2	2	2	4	2	4
3	4	2	6	0	4
4	4	0	4	0	4
5	6	0	6	2	8
6	6	2	8	2	8
⋮	⋮	⋮	⋮	⋮	⋮

functions $S_n^{1,2,3,4,9,10,11,\cdots}$ realized by instances of Figure 3.2 have indicator function $b_2(w+3)$. This indicator function follows a "4-on, 4-off" pattern. Note that these symmetric functions, unlike the functions $S_n^{1,2,5,6,\cdots}$ realized in Section 3.1.1, are "completely new" in the sense that (for sufficiently large n) one cannot obtain them using exclusive-OR operations starting from the symmetric functions whose realizations were presented in Chapter 2.

Why use $S_n^{1,2,5,6,\cdots}$ instead of $S_n^{2,3,6,7,\cdots}$ in the third stage of Figure 3.2? If we were to use $S_n^{2,3,6,7,\cdots}$, then the resulting total effective number of X_2 gates applied to y would be as shown in Table 3.3. From this table, it is apparent that only $S_n^{1,2,5,6,\cdots}$ produces permutative circuits when used as the control function for the third stage of Figure 3.2: with $S_n^{2,3,6,7,\cdots}$, an instance of the resulting circuit structure might apply an effective number of 2 or 6 X_2 gates to y, which produce non-permutative operations.

We can extend the circuit structure in Figure 3.2 even further with additional stages, as was done in Section 2.3.5 of Chapter 2. Also as in Section 2.3.5, this can be done by starting with an operational equation and thence generating the circuit structure, rather than

obtaining the operational equation only as a result of analysis of an already-known circuit structure generated by other means. For instance, consider the equation

$$w + \mathrm{b}_0(w) + 2\mathrm{b}_1(w+1) + 4\mathrm{b}_2(w+3) = \mathrm{tr}_3(w+7). \tag{3.3}$$

The validity of this equation for all w will be proved later as a case of Theorem 11, but it can easily be verified for some small values of w. From this operational equation we obtain the circuit structure shown in Figure 3.3, where the four stages of this circuit structure correspond to the four terms on the left-hand side of (3.3). In particular, the third stage of Figure 3.3 is derived from the $2\mathrm{b}_1(w+1)$ term in (3.3) and therefore must apply a U^2 gate (equivalent to 2 U gates) when $\mathrm{b}_1(w+1) = 1$, which it does by using a control function with indicator function $\mathrm{b}_1(w+1)$, namely $\mathrm{S}_n^{1,2,5,6,\cdots}$. Similarly, the fourth stage of Figure 3.3 is derived from the $4\mathrm{b}_2(w+3)$ in (3.3) and therefore applies a U^4 gate with the control function $\mathrm{S}_n^{1,2,3,4,9,10,11,\cdots}$.

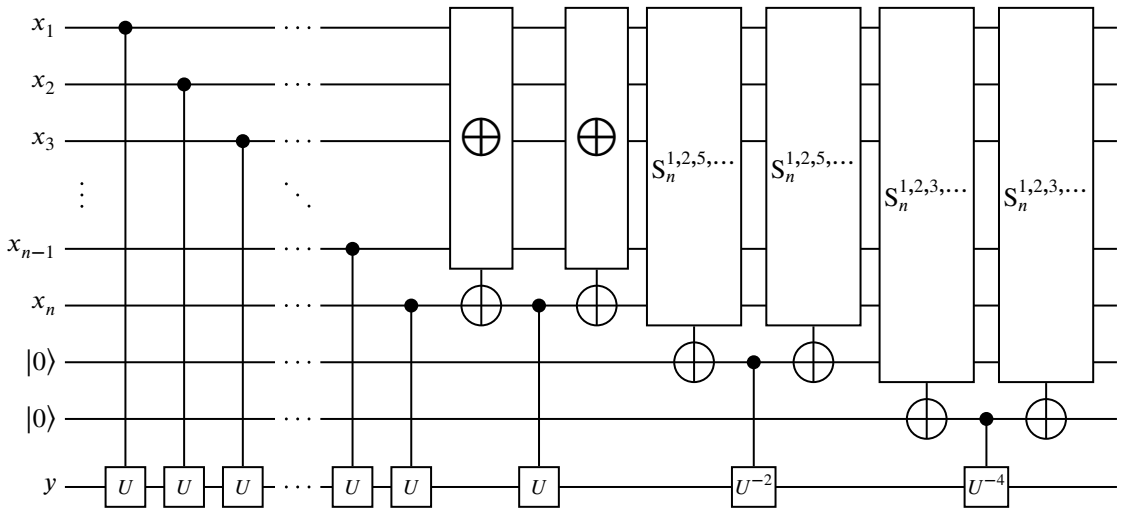


Figure 3.3: Circuit structure derived from eq. (3.3).

Eq.(3.3) tells us that instances of Figure 3.3 apply a total effective number of $\mathrm{tr}_3(w+7)$

U gates to qubit y. If we take $U = X_3$, then $U^8 = X$ and $U^{16} = I$, so this total effective number can be reduced modulo 16, giving $\mathrm{tr}_3(w + 7) \bmod 16 = 8\mathrm{b}_3(w + 7)$. Hence, an instance of Figure 3.3 with $U = X_3$ inverts y when $\mathrm{b}_3(w + 7) = 1$ and therefore realizes the symmetric function $\mathrm{S}_n^{1,2,\ldots,8,17,18,\ldots,24,33,34,\ldots}$, where the indicator function follows an "8-on, 8-off" pattern. The ON-set of this symmetric function consists of positive integers of the form $16m + p$, where m is any integer and $1 \leq 8 \leq p$.

The operational equations (3.1), (3.2), and (3.3) all follow the pattern

$$w + \mathrm{b}_0(w) + 2\mathrm{b}_1(w) + \cdots + 2^{k-1}\mathrm{b}_{k-1}(w) = \mathrm{tr}_k(w + 2^k - 1). \tag{3.4}$$

However, the next example breaks this pattern by allowing an arbitrary sign for the rotation in each stage. For instance, Figure 3.4 shows another circuit structure that differs from Figure 3.2 only in the target gate used for the third stage, which is now X_2^{-2}. This circuit structure has the operational equation

$$w + \mathrm{b}_0(w) - 2\mathrm{b}_1(w + 1) = \mathrm{tr}_2(w + 1), \tag{3.5}$$

which shows that instances of this structure realize symmetric functions with indicator function $\mathrm{tr}_2(w + 1) \bmod 8 = \mathrm{b}_2(w + 1)$ and therefore of the form $\mathrm{S}_n^{3,4,5,6,11,12,13\ldots}$.

3.1.3 DIPS with nonzero offset

At this point, we can see that the symmetric functions generated by allowing a different rotation sign in each stage all appear to have indicator functions of the form $\mathrm{b}_k(w + s)$. In other words, these indicator functions have an additional shift s compared with the indicator functions $\mathrm{b}_k(w)$ of the DIPS realized in Chapter 2. The following definition extends the

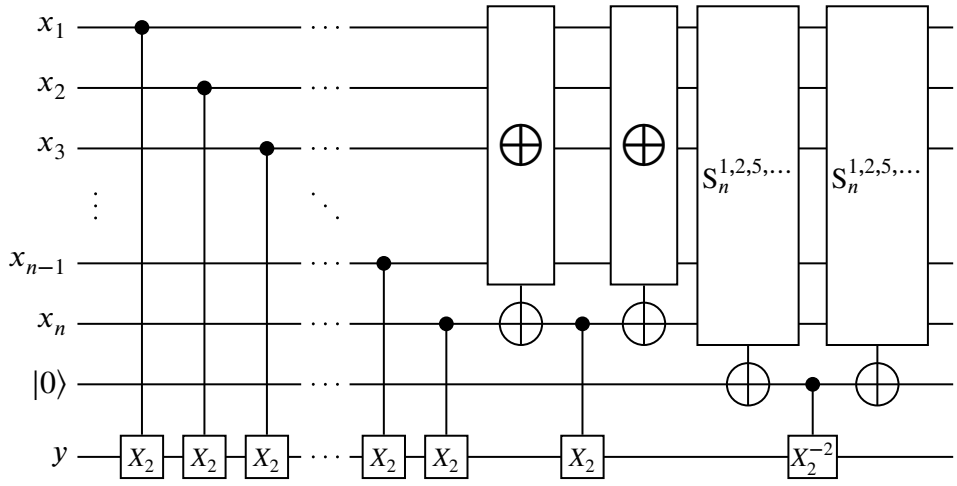


Figure 3.4: The circuit structure from Figure 3.2 with the last controlled-X_2^2 gate replaced by a controlled-X_2^{-2}.

$S^{DIP(k,0)}$ notation from Chapter 2 to incorporate the new shift s, giving the notation $S^{DIP(k,s)}$:

Definition 9. Given natural numbers k, n, and s, the n-input *dyadic indicator-periodic symmetric function* (DIPS) with *order* k and *offset* s, denoted $S_n^{DIP(k,s)}$, is the n-input symmetric function with indicator function

$$I^{DIP(k,s)}(w) = b_k(w + s). \tag{3.6}$$

Therefore, an order-k DIPS with nonzero offset has a periodic indicator function that follows a "2^k-on, 2^k-off" pattern, like the zero-offset DIPS from Chapter 2, but the point at which the indicator function switches from 0 to 1 or vice versa is different than for a zero-offset DIPS. The indicator function $b_k(w)$ of a zero-offset DIPS first attains the value 1 as $w = 2^k$, but that of an order-k offset-s DIPS, $b_k(w + s)$, first attains the value 1 earlier, at $w = 2^k - s$, assuming that $s \leq 2^k$.

To give some examples of this new notation, the symmetric functions realized by instances of Figure 3.2, $S_n^{1,2,5,6,\cdots}$, can be denoted $S_n^{DIP(1,1)}$. In comparison with $S_n^{DIP(1,0)} =$

$\mathrm{S}_n^{2,3,6,7,\ldots}$, we can see that the members of the ON-set have been shifted or offset by 1, illustrating the reason for the name "offset". Similarly, the functions $\mathrm{S}_n^{1,2,3,4,9,10,11,\ldots}$ and $\mathrm{S}_n^{3,4,5,6,11,12,13\ldots}$ can be denoted $\mathrm{S}_n^{\mathrm{DIP}(2,3)}$ and $\mathrm{S}_n^{\mathrm{DIP}(2,1)}$, respectively, and comparison with $\mathrm{S}_n^{\mathrm{DIP}(2,0)} = \mathrm{S}_n^{4,5,6,7,12,13,14,\ldots}$ shows that the members of ON-sets of the first two functions are indeed offset by 3 and 1, respectively. The functions $\mathrm{S}_n^{1,2,\ldots,8,17,18,\ldots,24,33,34,\ldots}$, realized by instances of Figure 3.3, are order-3 DIPS $\mathrm{S}_n^{\mathrm{DIP}(3,7)}$.

Any order-k DIPS with an offset greater than 2^k is either identical to or an inversion of an order-k DIPS with an offset less than 2^k. Specifically,

$$\mathrm{S}_n^{\mathrm{DIP}(k,s)} = \mathrm{S}_n^{\mathrm{DIP}(k,s \bmod 2^{k+1})} \tag{3.7}$$

for any natural numbers k and s. This can be seen from (3.6) and some basic properties of base-two arithmetic: adding 2^{k+1}, which has base-two representation $100\ldots00$ with $k+1$ zeroes, to any natural number N cannot possibly affect the last $k+1$ bits of N, so $\mathrm{b}_k(w+s+2^{k+1}) = \mathrm{b}_k(w+s)$ for all s, from which it follows that $\mathrm{b}_k(w+s) = \mathrm{b}_k(w+(s \bmod 2^{k+1}))$. Similarly, 2^k has base-two representation $100\ldots00$ with k zeroes, putting the 1 in the k-th bit position, so adding 2^k to a natural number N always inverts the k-th bit of N. Therefore,

$$\mathrm{b}_k(w+s+2^k) = \neg\mathrm{b}_k(w+s), \tag{3.8}$$

which implies via Definition 9 that

$$\mathrm{S}_n^{\mathrm{DIP}(k,s+2^k)} = \neg\mathrm{S}_n^{\mathrm{DIP}(k,s)} \tag{3.9}$$

for all n, k and s. Combining (3.7) and (3.9), it is easy to see that an order-k DIPS with

arbitrary offset can always be expressed in terms of an order-k DIPS with offset less than 2^k using at most one logical NOT and no other operations. For instance, $\mathrm{S}_n^{\mathrm{DIP}(1,5)}$ is the same as $\mathrm{S}_n^{\mathrm{DIP}(1,1)} = \mathrm{S}_n^{1,2,5,6,\cdots}$, and $\mathrm{S}_n^{\mathrm{DIP}(1,3)}$ is its negation.

3.2 General circuit structure with arbitrary rotation signs

3.2.1 Mathematical analysis of adding one stage with arbitrary rotation sign

In Section 3.1.2, it was observed that the operational equations for the considered circuit structures all followed the pattern (3.4), until this pattern was broken by (3.5). However, the following more general pattern includes both (3.4) and (3.5) as cases:

$$w \pm \mathrm{b}_0(w) \pm 2\mathrm{b}_1(w + s_1) \pm 4\mathrm{b}_2(w + s_2) \\ \pm \cdots \pm 2^{k-1}\mathrm{b}_{k-1}(w + s_{k-1}) = \mathrm{tr}_k(w + s_k), \tag{3.10}$$

where s_1 through s_k are arbitrary natural numbers and each of the terms on the left-hand side after the first may independently have a positive or negative sign. Eq. (3.10) corresponds to and can be used to generate the circuit structure shown in Figure 3.5. This circuit structure is similar to Figure 2.9 from Chapter 2 but has the following changes: the control function in each stage from the third onwards now incorporates an offset s_i, and the target gates include a plus-or-minus sign in their exponent, corresponding to the plus-or-minus signs in (3.10) and representing the possibility of a positive or negative rotation when U itself is a rotation.

Unfortunately, we do not yet know how the parameters s_i and the signs of the terms on the left-hand side of (3.10) must relate to each other in order for the equation to be true. It is also not yet clear whether all possible values of s_k on the right-hand side of (3.10) can be obtained through appropriate combinations of the other parameters. If all possible values

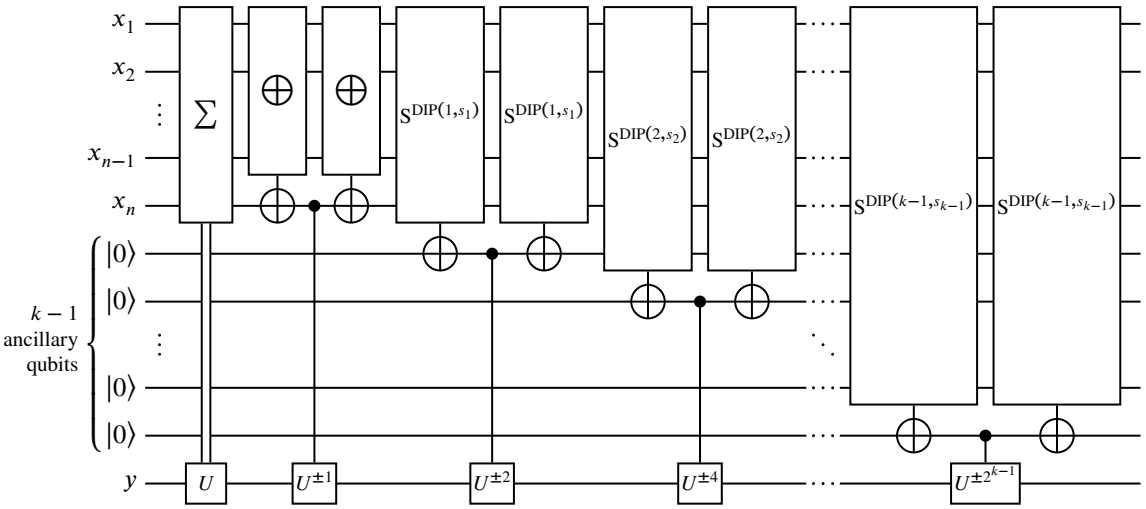


Figure 3.5: Circuit structure corresponding to (3.10).

of s_k can be obtained, then (3.10) can be used to generate circuit structures that realize order-k DIPS with any offset, by instantiating Figure 3.5 with the appropriate parameters and $U = X_k$. In this section I will show that this is indeed the case.

Following the lead of Section 2.4, it is useful to examine the effect of adding a single new stage to an existing circuit structure. Suppose that we have an existing circuit structure satisfying the following properties, which are identical to the ones introduced in Section 2.4 except for one small change:

1. It operates on any number n of input qubits x_1 through x_n together with a result qubit y. Any number of ancillary qubits may also be used.

2. All qubits other than y are restored to their original states at the end of the circuit.

3. All gates in the structure that can affect the state of qubit y are controlled gates whose target gates are powers of a single gate U. (For this purpose, $U^\dagger$ and powers of it are considered negative powers of U.) Therefore, in any instance of the structure, the

effective operation performed on qubit y can always be expressed as a number of U gates.

4. The effective number of U gates applied to qubit y depends only on the input weight, $w = \sum x_i$, and is given by $\mathrm{tr}_k(w + s)$ for some fixed k and $s < 2^k$.

The change from Section 2.4 is that property 4 now allows the total effective number of U gates to be $\mathrm{tr}_k(w + s)$, with some possibly nonzero offset s, instead of just $\mathrm{tr}_k(w)$. Consider the process of adding one additional stage to this existing structure, where the new stage consists of either a controlled-U^{2^k} or controlled-U^{-2^k} gate with a symmetric function as its control function. Tables 3.4 and 3.5 demonstrate the two possibilities for the target gate of the new stage. In both cases, if the new circuit structure is to also satisfy properties 1 through 4 above, with k replaced by $k + 1$, then there is only one possibility for the control function of the new stage: it must be $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$, because any other control function would result in the total effective number of U gates (as seen in the last columns of Tables 3.4 and 3.5) taking on values other than multiples of 2^{k+1}, which would violate property 4 above.

From Tables 3.4 and 3.5 we can make the following observations. When the added stage has a negative rotation U^{-2^k} as its target gate, then the resulting total effective number of gates jumps from 0 to 2^{k+1} at an input weight of $w = 2^{k+1} - s$, so this total effective number appears to be given by $\mathrm{tr}_{k+1}(w + s)$. On the other hand, when the added stage has a positive rotation U^{2^k} as its target gate, then the resulting total effective number of gates jumps from 0 to 2^{k+1} at an input weight of $w = 2^k - s$, so it appears to be given by $\mathrm{tr}_{k+1}(w + s + 2^k)$. In sum, the offset s appears to increase by 2^k with a positive rotation but remain the same for a negative rotation. This phenomenon can be concisely formulated in mathematical terms in the following way. Let $d = 0$ if the new stage's target gate is a positive rotation and $d = 1$ if it is a negative rotation, so that $(-1)^d$ represents the sign of the target gate's rotation. Then,

Table 3.4: Total effective number of U gates resulting from addition of a new stage with negative target rotation to an existing circuit structure.

$\sum_{i=1}^{n} x_i$	Applied U gates		Tot. eff. num. of U gates
	Existing circuit	New stage	
0	0	0	0
⋮	⋮	⋮	⋮
$2^k - s - 1$	0	0	0
$2^k - s$	2^k	-2^k	0
⋮	⋮	⋮	⋮
$2^{k+1} - s - 2$	2^k	-2^k	0
$2^{k+1} - s - 1$	2^k	-2^k	0
$2^{k+1} - s$	2^{k+1}	0	2^{k+1}
$2^{k+1} - s + 1$	2^{k+1}	0	2^{k+1}
⋮	⋮	⋮	⋮

Table 3.5: Total effective number of U gates resulting from addition of a new stage with positive target rotation to an existing circuit structure.

$\sum_{i=1}^{n} x_i$	Applied U gates		Tot. eff. num. of U gates
	Existing circuit	New stage	
0	0	0	0
⋮	⋮	⋮	⋮
$2^k - s - 1$	0	0	0
$2^k - s$	2^k	2^k	2^{k+1}
⋮	⋮	⋮	⋮
$2^{k+1} - s - 2$	2^k	2^k	2^{k+1}
$2^{k+1} - s - 1$	2^k	2^k	2^{k+1}
$2^{k+1} - s$	2^{k+1}	0	2^{k+1}
$2^{k+1} - s + 1$	2^{k+1}	0	2^{k+1}
⋮	⋮	⋮	⋮

when the target gate is active, the amount it contributes to the total effective number of U gates is $-(-1)^d 2^k$. Since the new stage's control function is $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$, which has indicator function $\mathrm{b}_k(w+s)$, that stage therefore contributes the amount of $-(-1)^d 2^k \mathrm{b}_k(w+s)$ to the total effective number of U gates. The offset s increases by $d \cdot 2^k$ when the new stage is added. The whole process of adding an additional stage to an existing circuit structure is therefore mathematically described by the following proposition.

Proposition 10. *For all natural numbers w, s, and k, and $d \in \{0,1\}$, the following equality holds:*

$$\mathrm{tr}_k(w+s) - (-1)^d 2^k \mathrm{b}_k(w+s) = \mathrm{tr}_{k+1}(w+s+d\cdot 2^k). \tag{3.11}$$

Proof. First, suppose that $d = 0$. Then, substituting $N = w + s$ into (2.31) from Chapter 2, Proposition 6, we have

$$\mathrm{tr}_k(w+s) - 2^k \mathrm{b}_k(w+s) = \mathrm{tr}_{k+1}(w+s), \tag{3.12}$$

which is the form of (3.11) when $d = 0$.

Now consider the case $d = 1$. In this case, we have

$$\begin{aligned}
\mathrm{tr}_k(w+s) + 2^k \mathrm{b}_k(w+s) &= \mathrm{tr}_k(w+s) + 2^k\big(1 - \neg \mathrm{b}_k(w+s)\big) && (3.13)\\
&= \mathrm{tr}_k(w+s) + 2^k - 2^k \mathrm{b}_k(w+s+2^k) && (3.14)\\
&= \mathrm{tr}_k(w+s+2^k) - 2^k \mathrm{b}_k(w+s+2^k) && (3.15)\\
&= \mathrm{tr}_{k+1}(w+s+2^k), && (3.16)
\end{aligned}$$

where (3.13) uses the fact that the logical NOT of a bit $b \in \{0,1\}$ can also be expressed arithmetically as $1 - b$, and uses (3.8) to replace $\neg \mathrm{b}_k(w+s)$ with $\mathrm{b}_k(w+s+2^k)$; (3.14)

follows from (2.20) of Chapter 2; and (3.16) is the same as (3.12) with $w + s$ replaced by $w + s + 2^k$. □

Proposition 10 can be represented in circuit form as shown in Figure 3.6, which is analogous to Figure 2.8 from Chapter 2. Figure 3.6 shows that given an existing circuit structure that applies a total effective number of $\mathrm{tr}_k(w + s)$ U gates to qubit y, adding $U^{\pm 2^k}$ gate controlled by an order-k offset-s DIPS creates a new structure that applies a total effective number of either $\mathrm{tr}_k(w + s)$ or $\mathrm{tr}_k(w + s)$ U gates to y, depending on the rotation sign of the target gate.

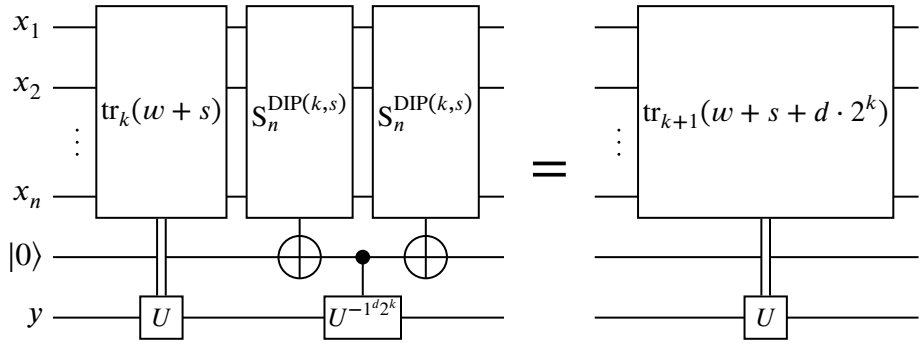


Figure 3.6: Recursion relation used to realize a DIPS of order $k + 1$ from one of order k, showing relationship between rotation sign and offsets.

3.2.2 General circuit structure for realizing DIPS of any order and offset

The following theorem, which characterizes the combinations of signs and offsets that will satisfy (3.10), can now be proved using Proposition 10 as an induction step.

Theorem 11. *Let k and s be natural numbers with $s < 2^k$. Then the following equality holds:*

$$w - \sum_{i=0}^{k-1} (-1)^{\mathrm{b}_i(s)} 2^i \mathrm{b}_i\big(w + (s \bmod 2^i)\big) = \mathrm{tr}_k(w + s) \tag{3.17}$$

for all natural numbers w.

Proof. We proceed by induction on k. First, for $k = 0$, the sum in (3.17) is empty, so the left-hand side reduces to just w. To satisfy $s < 2^k$, s must be zero, and we know that $\mathrm{tr}_0(w) = w$. Therefore, the right-hand side of (3.17) also reduces to w.

For the induction step, suppose that (3.17) holds for a particular k and all $s < 2^k$. We wish to show that (3.17) also holds for $k + 1$ and all $s < 2^{k+1}$. To avoid confusion, we will relabel this "new" s as s'. Now let $s = s' \bmod 2^k$; *i.e.*, s is obtained by discarding the most significant bit, $\mathrm{b}_{k+1}(s')$, of s'. Then $s < 2^k$, so by the induction hypothesis, (3.17) holds. Add the new term $-(-1)^{\mathrm{b}_{k+1}(s')}2^k\mathrm{b}_i(w+s)$ to both sides of (3.17) to obtain

$$w - \left[\sum_{i=0}^{k-1}(-1)^{\mathrm{b}_i(s)}2^i\mathrm{b}_i\big(w + (s \bmod 2^k)\big)\right] - (-1)^{\mathrm{b}_k(s')}2^k\mathrm{b}_k(w+s) \\ = \mathrm{tr}_k(w+s) - (-1)^{\mathrm{b}_k(s')}2^k\mathrm{b}_k(w+s). \quad (3.18)$$

Since s was obtained by discarding the most significant bit from s', the base-two representations of s and s' agree in all bits up to the $(k-1)$-th one. In other words, $\mathrm{b}_i(s) = \mathrm{b}_i(s')$ for $i < k$. Additionally, since $s < 2^k$, $s \bmod 2^k = s$, so the new term on the left-hand side of (3.18) can be absorbed into the sum, giving

$$\begin{aligned} &- \left[\sum_{i=0}^{k-1}(-1)^{\mathrm{b}_i(s)}2^i\mathrm{b}_i\big(w + (s \bmod 2^i)\big)\right] - (-1)^{\mathrm{b}_k(s')}2^k\mathrm{b}_k(w+s) \\ &\quad = -\left[\sum_{i=0}^{k-1}(-1)^{\mathrm{b}_i(s')}2^i\mathrm{b}_i\big(w + (s \bmod 2^i)\big)\right] - (-1)^{\mathrm{b}_k(s')}2^k\mathrm{b}_k\big(w + (s \bmod 2^k)\big) \\ &\quad = -\sum_{i=0}^{k}(-1)^{\mathrm{b}_i(s')}2^i\mathrm{b}_i\big(w + (s \bmod 2^i)\big) \\ &\quad = -\sum_{i=0}^{k}(-1)^{\mathrm{b}_i(s')}2^i\mathrm{b}_i\big(w + (s' \bmod 2^i)\big). \end{aligned} \quad (3.19)$$

In the last line of (3.19), we have furthermore used the fact that, since s and s' agree in all

bits up to the $(k-1)$-th, $s \bmod 2^i = s' \bmod 2^i$ for all $i \leq k$.

Now observe that, again since s was obtained by discarding the most significant bit from s', s' can be obtained by adding this bit with its appropriate place value back to s, *i.e.*, $s' = s + \mathrm{b}_k(s') \cdot 2^k$. Use this fact and apply Proposition 10 to the right-hand side of (3.18) to get

$$\begin{aligned} &\mathrm{tr}_k(w+s) - (-1)^{\mathrm{b}_k(s')} 2^k \mathrm{b}_k(w+s) \\ &\qquad = \mathrm{tr}_{k+1}\big(w + s + \mathrm{b}_k(s') \cdot 2^k\big) = \mathrm{tr}_{k+1}(w+s'). \end{aligned} \tag{3.20}$$

Substituting (3.19) and (3.20) into (3.18) gives

$$w - \sum_{i=0}^{k} (-1)^{\mathrm{b}_i(s')} 2^i \mathrm{b}_i\big(w + (s' \bmod 2^i)\big) = \mathrm{tr}_{k+1}(w+s'), \tag{3.21}$$

which completes the induction step. □

Theorem 11 is a generalization of Theorem 7 that shows how (3.10) can be satisfied for any s_k. Specifically, to get a desired s_k on the right-hand side of (3.10), take $s_i = s_k \bmod 2^i$ for $1 \leq i \leq k-1$[1] on the left-hand side and let the sign of the b_i term be positive if and only if the i-th bit of s is 1. Substituting these parameters into Figure 3.5 then allows DIPS of any order k and offset $s < 2^k$ to be realized. As observed at the end of Section 3.1, eqs. (3.7) and (3.9) then allow DIPS of all other offsets to be obtained, since they are either identical to a DIPS with offset $s < 2^k$ or can be obtained by inverting such a DIPS.

Corollary 12. *Given natural numbers n, k, and s with $s < 2^k$, the quantum circuit obtained as an instance of Figure 3.5 with the following parameters realizes the symmetric function*

[1] Note that there is no s_0 in (3.10) because it would always be 0, which is consistent with Theorem 11: $s_k \bmod 2^0 = s_k \bmod 1 = 0$ for all s_k.

Table 3.6: Parameter values used to reproduce the circuit structures from Section 3.1 using Corollary 12.

Circuit structure from figure	Order k	Offset s	Values of s_i	Rotation signs
Figure 3.1	1	1	$s_0 = 0$	Positive
Figure 3.2	2	3	$s_0 = 0$ $s_1 = 1$	Positive Positive
Figure 3.3	3	7	$s_0 = 0$ $s_1 = 1$ $s_2 = 3$	Positive Positive Positive
Figure 3.4	2	1	$s_0 = 0$ $s_1 = 1$	Positive Negative

$\mathrm{S}_n^{\mathrm{DIP}(k,s)}$:

- *n is as given;*
- *$U = X_k$;*
- *$s_i = s \bmod 2^i$ for $1 \leq i \leq k-1$; and*
- *The target gate of the stage controlled by $\mathrm{S}_n^{\mathrm{DIP}(i,s_i)}$ is $U^{2^i} = X_k^{2^i} = X_{k-i}$ if $\mathrm{b}_i(s) = 1$ and is $U^{-2^i} = X_k^{-2^i} = X_{k-i}^{-1}$ otherwise.*

Table 3.6 shows parameter values that can be used in Corollary 12 to reproduce each of the examples from Section 3.1. For each circuit structure examined in Section 3.1, Table 3.6 gives the order and offset of the realized DIPS, the value of each s_i,[2] and the rotation signs of the circuit stages corresponding to those s_i's.

[2] An $s_0 = 0$ entry is included for each example in order to specify the rotation direction for the corresponding stage, which is the one controlled by the exclusive-OR of all inputs.

3.3 Optimized circuit structure for realization of arbitrary DIPS

3.3.1 Derivation of the optimized structure

Corollary 12 provides a method to realize a DIPS with any order and offset. However, the circuit structure used to accomplish this, shown in Figure 3.5, still suffers from the same problem pointed out at the end of Chapter 2. Specifically, when fully expanded, the structure from Figure 3.5 results in a very large number of two-qubit controlled gates for even moderately large k, as demonstrated by Figure 2.10. While some of the CNOT gates in Figure 2.10 can clearly be canceled, it is not clear to what extent this cancellation is possible in general for large k, because of the many recursive steps needed to fully expand Figure 3.5.

Corollary 12 is based on (3.10) and Theorem 11, which show how an appropriate operational equation can be obtained to realize any desired DIPS. Figure 3.5 is only one way of mapping this operational equation to a circuit. Here I show that a different method of mapping (3.10) to a circuit provides a much more efficient realization of DIPS compared with recursively expanding Figure 3.5 fully.

To see how such a mapping might work, suppose that we wish to realize a DIPS of order k and offset s with $s < 2^k$. Theorem 11 then tells us that we first require a DIPS of order i and offset $s \bmod 2^i$ for every i from 0 to $k-1$. Assume that these DIPS have already been realized; we will for the moment be unconcerned with the details of how this prior realization takes place. Then the desired DIPS of order k and offset s can be realized using an instance of the circuit structure shown in Figure 3.7 with appropriate rotation signs, where the prerequisite DIPS of lower orders are assumed to be available on separate qubits.

If we now consider how the prerequisite DIPS of order $k-1$, $\mathrm{S}_n^{\mathrm{DIP}(k-1,s \bmod 2^{k-1})}$, can be realized, it becomes apparent that all of its own prerequisite DIPS are prerequisites of

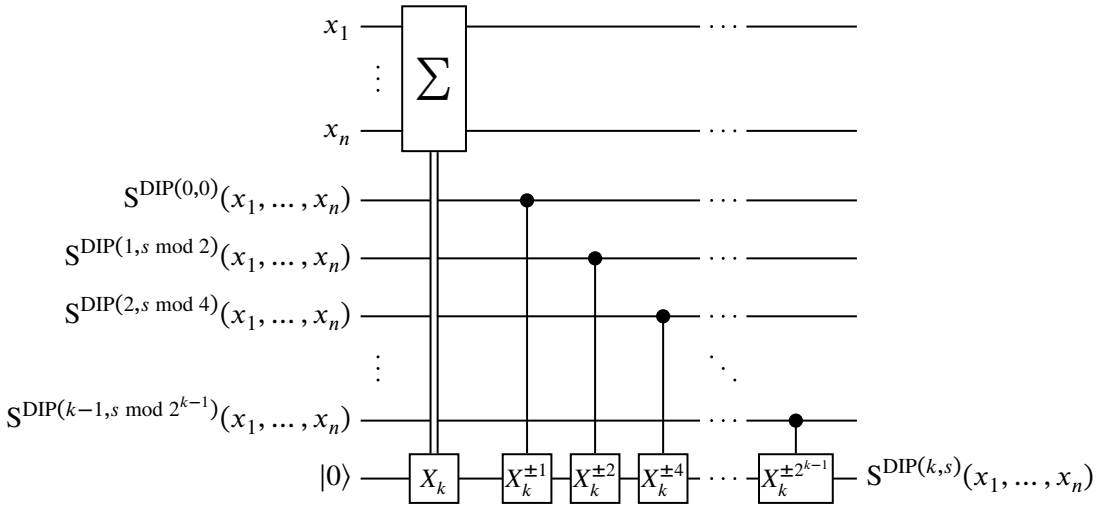


Figure 3.7: Realization of an order-k, offset-s DIPS assuming that appropriate DIPS of lower orders have already been realized.

$\mathrm{S}_n^{\mathrm{DIP}(k,s)}$ as well and so are already present in Figure 3.7. Specifically, to realize a DIPS of order $k-1$ and offset $s \bmod 2^{k-1}$, Theorem 11 tells us that we first require a DIPS of order i and offset $(s \bmod 2^{k-1}) \bmod 2^i$ for every i from 0 to $k-2$. But since 2^{k-1} is an integer multiple of 2^i for $i \leq k-2$, we have $(s \bmod 2^{k-1}) \bmod 2^i = s \bmod 2^i$, so we require a DIPS of order i and offset $s \bmod 2^i$ for i from 0 to $k-2$, which are already assumed to be available in Figure 3.7. $\mathrm{S}_n^{\mathrm{DIP}(k-1,s \bmod 2^{k-1})}$ can therefore be realized in the same way that $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$ is realized in Figure 3.7. Furthermore, this realization can be inserted back into Figure 3.7 itself, so that we do not have to assume $\mathrm{S}_n^{\mathrm{DIP}(k-1,s \bmod 2^{k-1})}$ is already available from the start. The result is shown in Figure 3.8. In Figure 3.8, it is only assumed that the prerequisite DIPS of orders 0 through $k-2$ have already been realized (as opposed to 0 through $k-1$ in Figure 3.7), and the circuit structure realizes the DIPS of both orders $k-1$ and $k-2$.

Continuing in this manner, the next step would be to realize the DIPS of order $k-2$, $\mathrm{S}_n^{\mathrm{DIP}(k-1,s \bmod 2^{k-2})}$. Theorem 11 tells us that we first require DIPS of order i and offset $(s \bmod 2^{k-2}) \bmod 2^i$ for $0 \leq i \leq k-3$, and like before we have $(s \bmod 2^{k-2}) \bmod 2^i =$

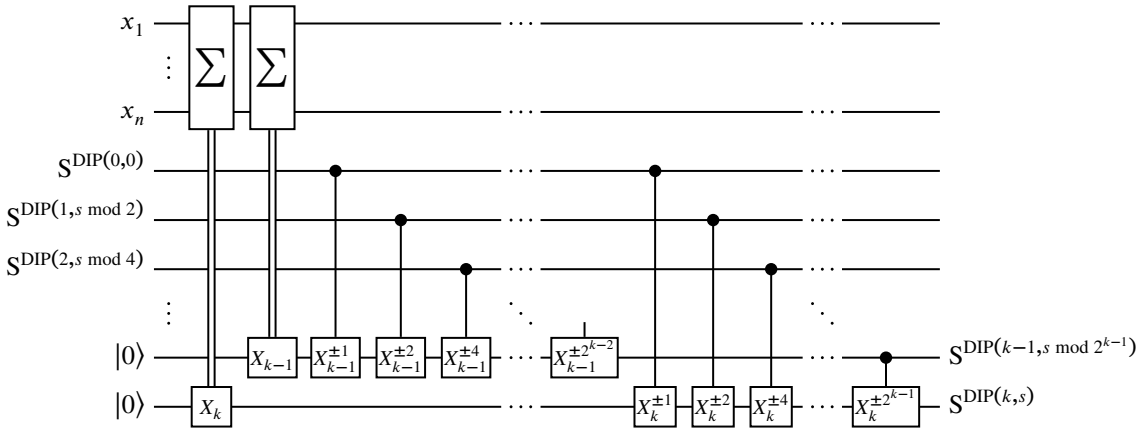


Figure 3.8: Realization of an order-k, offset-s and order-$(k-1)$, offset-$(s \bmod 2^{k-1})$ DIPS, assuming that appropriate DIPS of lower orders have already been realized.

$s \bmod 2^i$, so the prerequisite DIPS are again the ones that are already present in Figures 3.7 and 3.8. After that, the DIPS of order $k-3$ can be realized, and so on until the only DIPS needed as a prerequisite is $S_n^{DIP(0,0)}$, which is just the exclusive-OR of x_1 through x_n and so is easily realized. The final resulting circuit structure is shown in Figure 3.9.

The circuit structure of Figure 3.9 not only realizes the originally desired function $S_n^{DIP(k,s)}$, but also simultaneously realizes a whole sequence of DIPS of the form $S_n^{DIP(i,s \bmod 2^i)}$ for $0 \leq i \leq k$. This ability to realize a sequence of DIPS with orders from 0 to k all at once is very useful—particular uses will be demonstrated in 3.4 and in Chapter 4. However, if only $S_n^{DIP(k,s)}$ is wanted, then the other qubits can be returned to their original states using mirror gates.

A further optimization can be made to Figure 3.9 by noting that $S_n^{DIP(0,0)}$, which is the exclusive-OR of x_1 through x_n, does not require its own ancillary qubit but rather can be directly realized on the the qubit x_n using the same cascade of CNOT gates seen in every circuit structure from Chapter 2 and in this chapter prior to this section. Making this modification produces the circuit structure shown in Figure 3.10, which uses one less

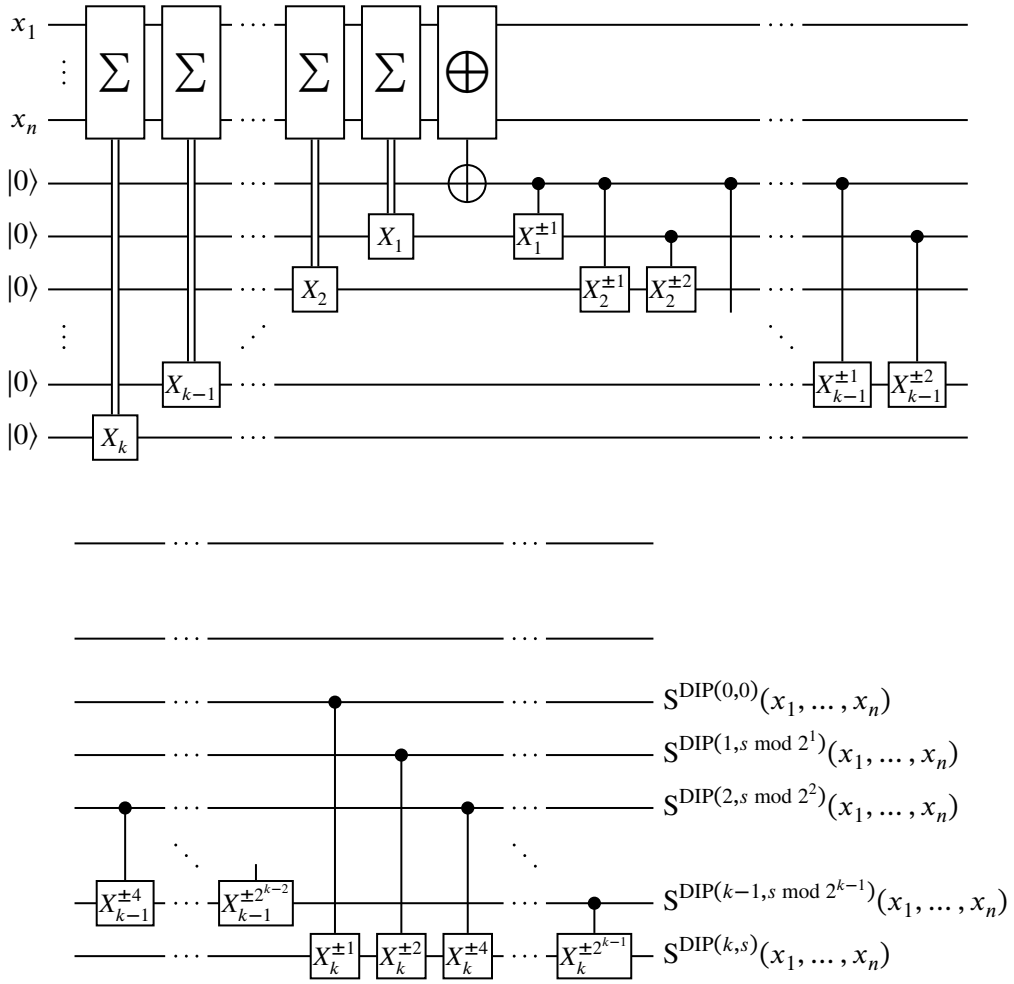


Figure 3.9: End result of continuing the realization process started in Figures 3.7 and 3.8.

ancillary qubit than the one from Figure 3.9.

As an example to demonstrate the efficiency of the circuit structure shown in Figure 3.10 in realizing DIPS, consider the DIPS $S_9^{DIP(3,0)}$ whose realization was demonstrated at the end of Chapter 2. Using Figure 3.10 to realize this DIPS produces the circuit shown in Figure 3.11. As in Figure 2.10, to improve legibility of the already-small labels, the target gates are labeled with a single integer representing a power of X_3. Comparing this circuit with the one from Figure 2.10 makes the improvement obvious—the circuit of Figure 3.11

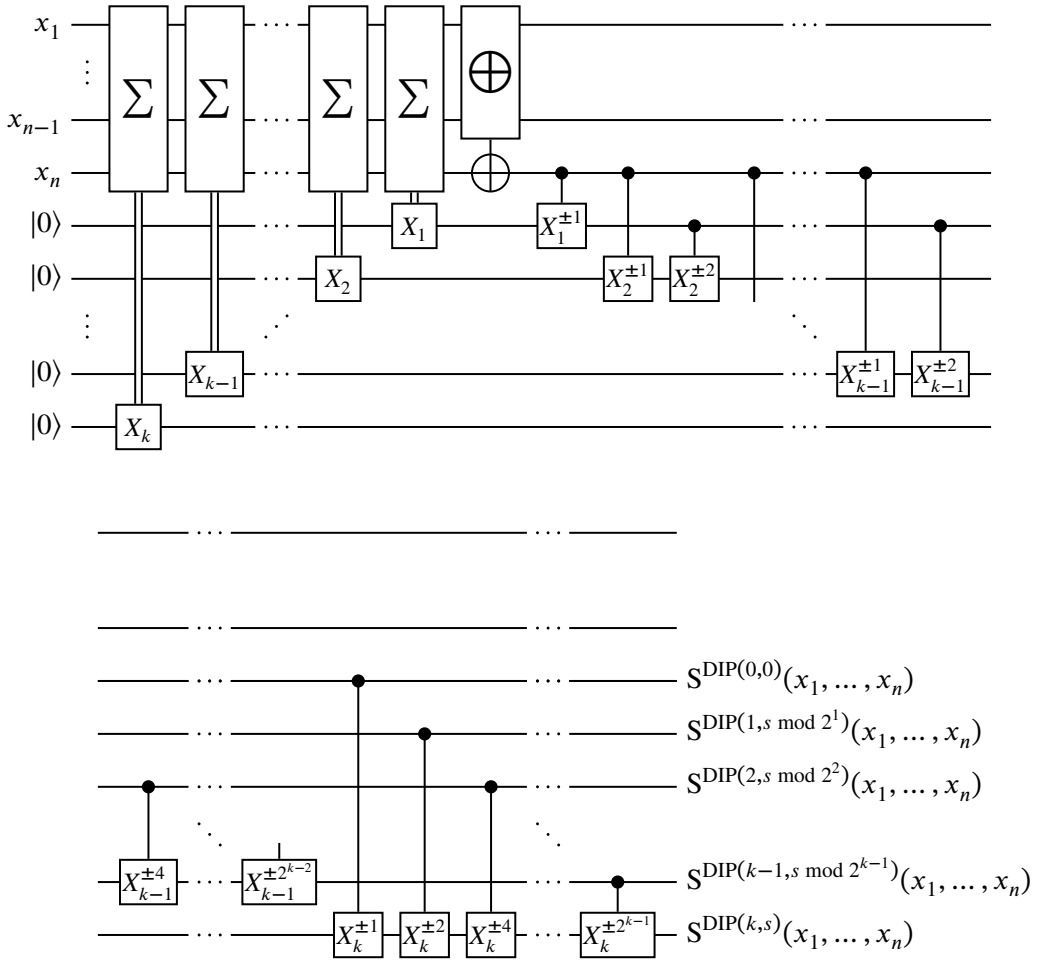


Figure 3.10: Elimination of one ancillary qubit from Figure 3.9 by realizing $S^{DIP(0,0)}(x_1, \ldots, x_n) = \bigoplus_{i=1}^{n} x_i$ directly on qubit x_n instead of on a separate ancillary qubit.

uses 41 two-qubit gates while the one from Figure 2.10 uses 236. In addition, the circuit of Figure 3.11 also produces the DIPS of lower orders $S_9^{DIP(0,0)}$, $S_9^{DIP(1,0)}$, and $S_9^{DIP(2,0)}$ as outputs, which the circuit from Figure 2.10 does not. If these additional functions are not wanted, then the quantum cost of Figure 3.11 actually increases because all gates other than those targeting y must be mirrored. Nevertheless, even in that case, the circuit from Figure 3.11 with mirror gates would only use 70 gates, which is still a drastic improvement over the 236 gates used in Figure 2.10.

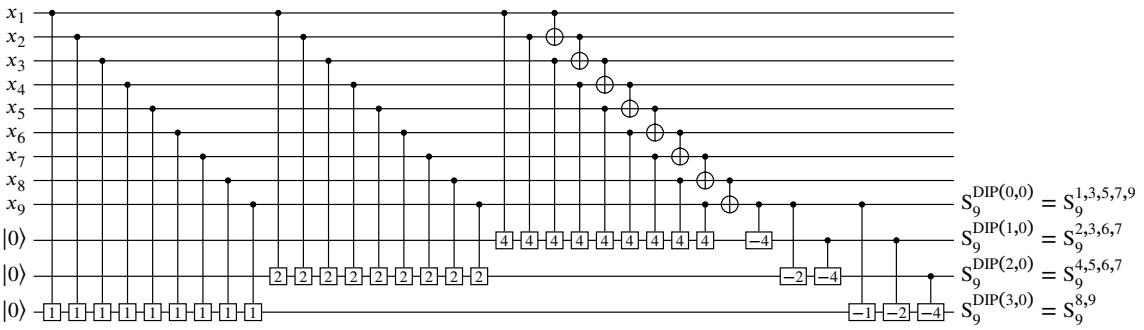


Figure 3.11: Realization of $S^{DIP(3,0)}(x_1, \ldots, x_9) = S^{8,9}(x_1, \ldots, x_9)$ using the circuit structure of Figure 3.10.

A second example that demonstrates the realization of a DIPS with nonzero offset is given in Figure 3.12, where the function $S_9^{DIP(3,5)} = S_9^{3,4,5,6,7,8,9}$ is realized. Since the offset 5 has base-two representation 101, applying Theorem 11 gives the operational equation

$$w + b_0(w) - 2b_1(w+1) + 4b_2(w+1) = tr_3(w+5), \tag{3.22}$$

so the lower-order DIPS realized are $S_9^{DIP(0,0)} = S_9^{1,3,5,7,9}$, $S_9^{DIP(1,1)} = S_9^{1,2,5,6,9}$, and $S_9^{DIP(2,1)} = S_9^{3,4,5,6}$.

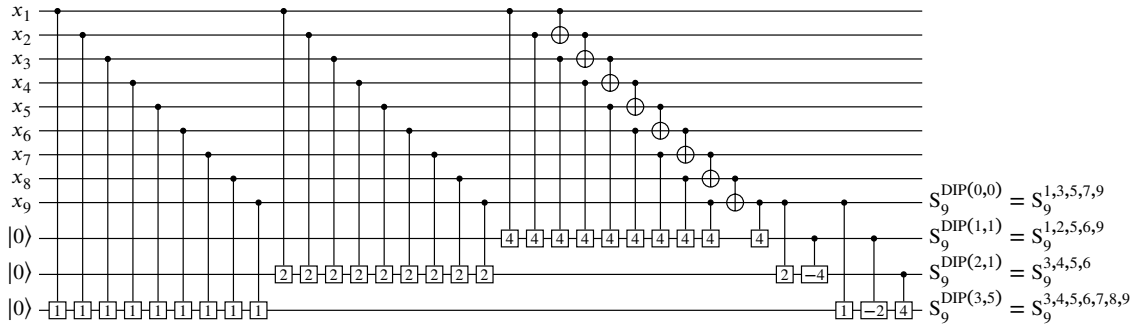


Figure 3.12: Realization of $S^{DIP(3,5)}(x_1, \ldots, x_9) = S^{3,4,5,6,7,8,9}(x_1, \ldots, x_9)$ using the circuit structure of Figure 3.10, showing mixed rotation signs in the last part of the circuit.

3.3.2 Analysis of quantum cost

Since the circuit structure of Figure 3.10 is already expressed entirely in terms of controlled-root-of-NOT gates (keeping in mind that the blocks labeled "$\sum$" and "$\bigoplus$" are just abbreviations for cascades of controlled gates), its quantum cost complexity can be determined by simply counting the total number of gates. Each of cascade of X_i gates (with i running from 1 to k) at the start consists of n gates, and there are k such cascades, so they incur a combined quantum cost of kn. The following cascade of CNOT gates contains $n - 1$ gates, bringing the total cost up to $kn + n - 1 = (k + 1)n - 1$. Finally, in the last part of the circuit, we can see that there is a single gate targeting the first ancillary qubit, a two gates targeting the second ancillary qubit, and so on up to k gates targeting the final output qubit for $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$, so the total number of gates in this part of the circuit is a triangular number $1 + 2 + \cdots + k = \sum_{i=1}^{k} i = k(k + 1)/2$. The total quantum cost of the whole circuit is therefore

$$(k + 1)n - 1 + \frac{k(k + 1)}{2} = (k + 1)\left(n + \frac{k}{2}\right) - 1. \tag{3.23}$$

Figure 3.10 was optimized from Figure 3.9 by observing that the exclusive-OR of x_1 through x_n could be realized on the same qubit where the input x_n is provided. For some purposes, like the counter circuit described in Section 3.4, we may need all output DIPS to be produced on separate qubits from the inputs x_1 through x_n, which necessitates the use of Figure 3.9. The cascade of CNOT gates in Figure 3.9 contains one more gate than the analogous cascade in Figure 3.10, and there is otherwise a one-to-one correspondence between the gates of the two circuit structures, so the quantum cost of an instance of Figure 3.9 is one more than (3.23), *i.e.*,

$$(k + 1)\left(n + \frac{k}{2}\right). \tag{3.24}$$

If a mirror circuit is used with Figure 3.10 to restore all qubits except the last to their original states, so that only the output $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$ is produced, then all gates not targeting the last qubit must be mirrored. These mirror gates consist of $k-1$ cascades of controlled-X_i gates, where each cascade contains n gates, plus a cascade of $n-1$ CNOT gates, plus $1+2+\cdots+(k-1) = k(k-1)/2$ additional controlled gates, giving a total quantum cost of $(k-1)n + (n-1) + k(k-1)/2 = kn + k(k-1)/2 - 1$ for the mirror circuit. Adding this cost onto (3.23) gives a total cost of

$$(2k+1)n + k^2 - 2 \tag{3.25}$$

for a circuit that realizes only $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$ and restores all other qubits to their original states.

Figure 3.10 is additionally seen to require k ancillary qubits. The circuit complexity for realizing a DIPS using Figure 3.10 can therefore be summarized as follows: quantum cost is linear with respect to the number of inputs and quadratic with respect to the order of the DIPS being realized, and the number of ancillary qubits needed is constant with respect to the number of inputs and linear with respect to the order of the DIPS.

3.3.3 Algorithm to generate quantum circuits for DIPS

The circuit structure shown in Figure 3.10 is highly regular, so it is straightforward to produce an algorithm that will generate an instance of this structure for given parameters n, k, and s. Such an algorithm is given as Algorithm 1. Algorithm 2 is a modification of Algorithm 1 that adds a mirror circuit to Figure 3.10, so that only the single function $\mathrm{S}_n^{\mathrm{DIP}(k,s)}$ is realized and all other qubits are restored to their original states.

If we assume that extracting a single bit from an integer is an $\mathcal{O}(1)$ operation, which is the case on virtually all modern digital computers, then both Algorithms 1 and 2 have

Algorithm 1: Realize the set of DIPS $\left\{ \mathrm{S}_n^{\mathrm{DIP}(j, s \bmod 2^j)} \;\middle|\; 0 \leq j \leq k \right\}$ for given n, k, s.

Function `RealizeDips`(n, k, s)

Input: A positive integer n, nonnegative integer k, and nonnegative integer s with $s < 2^k$.

Output: A quantum circuit, operating on n input qubits x_1 through x_n together with k ancillary qubits y_1 through y_k, that behaves as follows: x_n is left in a state corresponding to the exclusive-OR of x_1 through x_n, and when each y_j begins in the state $|c_j\rangle$, with $c_j \in \{0, 1\}$, the circuit leaves it in the state $\left|c_j \oplus \mathrm{S}^{\mathrm{DIP}(k, s \bmod 2^j)}(x_1, \ldots, x_n)\right\rangle$.

Let Q be a quantum circuit, initially containing no gates, that operates on qubits x_1 through x_n and y_1 through y_k.

for *j from k to* 1 **do**
  for *i from 1 to n* **do**
    Append a controlled-X_j gate with control x_i and target y_j to Q.
  end
end
for *i from 1 to* $n - 1$ **do**
  Append a controlled-NOT gate with control x_i and target x_{i+1} to Q.
end
Designate y_0 as an alias for the qubit x_n.
for *j from 1 to k* **do**
  for j' *from* 0 *to* $j - 1$ **do**
    if $\mathrm{b}_{j'}(s) = 1$ **then**
      Let $U = X_{j-j'}$.
    else
      Let $U = X_{j-j'}^{-1}$.
    end
    Append a controlled-U gate with control $y_{j'}$ and target y_j to Q.
  end
end
Output the circuit Q.
end

run-time complexities proportional to the total number of gates in the circuits that they generate. In other words, since neither algorithm involves any searching or backtracking, each gate is generated in $\mathcal{O}(1)$ time. The total number of gates generated by Algorithms 1 and 2 are given by (3.23) and (3.25), respectively, which are both $\mathcal{O}(kn + k^2)$. Therefore, both algorithms run in $\mathcal{O}(kn + k^2)$ time.

3.4 Use of DIPS as a counter circuit

3.4.1 Counter derived from DIPS of multiple orders

The ability to realize not only a DIPS of order k but also DIPS of lower orders in a single circuit structure can be used to create a *counter* circuit. Specifically, consider the DIPS realized by the circuit structure from Figure 3.10 for offset 0. When realizing $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$, the functions $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ through $\mathrm{S}_n^{\mathrm{DIP}(k-1,0)}$ will also be realized. The order-i DIPS in this sequence has indicator function $\mathrm{b}_i(w)$, meaning that it outputs 1 when the i-th bit of its input weight is 1. Therefore, if we collect the outputs of all the DIPS in the sequence, we can produce the base-two representation of the input weight w.

Table 3.7 shows an example of the the above with $k = 2$. From this table we see that the combination of $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$, $\mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$, in that order, gives the base-two representation of the input weight w when it is in the range 0 to 7. For larger input weights, DIPS of higher orders $\mathrm{S}_n^{\mathrm{DIP}(4,0)}$, $\mathrm{S}_n^{\mathrm{DIP}(5,0)}$ etc. have to be incorporated as well, since the base-two representation of the input weight then exceeds three bits in length. In general an instance of Figure 3.10 with offset 0 and a given k outputs the base-two representation of the input weight $w = \sum x_i$ as long as $w < 2^{k+1}$; this is what is meant by a "counter" circuit. In Chapters 7 and 8 I will make use of counter circuits in several ways when designing quantum oracles for Grover's algorithm.

Algorithm 2: Realize a single DIPS by adding mirror gates to a circuit produced by Algorithm 1.

Function `RealizeDipsWithMirror(`n, k, s`)`

Input: A positive integer n, nonnegative integer k, and nonnegative integer s with $s < 2^k$.

Output: A quantum circuit, operating on n input qubits x_1 through x_n together with k ancillary qubits y_1 through y_k, that behaves as follows: all x_i and y_1 through y_{k-1} are left in the same states in which they started, while if y_k begins in the state $|c_k\rangle$, then the circuit leaves it in the state $\left|c_k \oplus \mathrm{S}^{\mathrm{DIP}(k,s)}(x_1, \ldots, x_n)\right\rangle$.

Let the quantum circuit $Q =$ `RealizeDips(`n, k, s`)`.
Designate y_0 as an alias for the qubit x_n.
for *j from* $k-1$ *to 1* **do**
  for j' *from* $j-1$ *to 0* **do**
    if $\mathrm{b}_{j'}(s) = 1$ **then**
      Let $U = X_{j-j'}$.
    else
      Let $U = X_{j-j'}^{-1}$.
    end
    Append a controlled-U gate with control $y_{j'}$ and target y_j to Q.
  end
end
for *i from* $n-1$ *to 1* **do**
  Append a controlled-NOT gate with control x_i and target x_{i+1} to Q.
end
for *j from 1 to k* **do**
  for *i from n to 1* **do**
    Append a controlled-X_j gate with control x_i and target y_j to Q.
  end
end
Output the circuit Q.
end

Table 3.7: Computation of the base-two representation of the input weight w using a collection of DIPS.

$w = \sum_{i=1}^{n} x_i$	$\mathrm{S}^{\mathrm{DIP}(2,0)}(x_1, \ldots, x_n)$ $= \mathrm{S}^{4,5,6,7,\cdots}(x_1, \ldots, x_n)$ $= \mathrm{b}_2(w)$	$\mathrm{S}^{\mathrm{DIP}(1,0)}(x_1, \ldots, x_n)$ $= \mathrm{S}^{2,3,6,7,\cdots}(x_1, \ldots, x_n)$ $= \mathrm{b}_1(w)$	$\mathrm{S}^{\mathrm{DIP}(0,0)}(x_1, \ldots, x_n)$ $= \mathrm{S}^{1,3,5,7,\cdots}(x_1, \ldots, x_n)$ $= \mathrm{b}_0(w)$
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

The concept of a counter circuit is not new, and one can be constructed more simply by using *incrementer* circuits of the type shown in Figure 3.13. An incrementer circuit is defined to operate as follows: the circuit accepts one control qubit as together with some number r of additional qubits, designated *register* qubits, and increments the natural number represented in base-two by the register qubits when the control qubit has value 1. In other words, if the register qubits' values initially form the base-two representation of a number n, then the circuit updates their values to form the base-two representation of $n + 1$ if the control qubit has value 1. Otherwise, the register qubits remain unchanged. The incrementer circuit of Figure 3.13 has c as its control qubit, x_r as its most significant register qubit, and x_1 as its least significant register qubit.

By beginning with r register qubits initialized to 0 and repeatedly applying the incrementer circuit from Figure 3.13 with a different control qubit each time, we can create a circuit that counts the total number of ones present among the control qubits. However, since each incrementer circuit uses Toffoli gates with 1 through $r - 1$ control qubits, such a circuit has the disadvantage of using many $r - 1$-control Toffoli gates, which result in a large

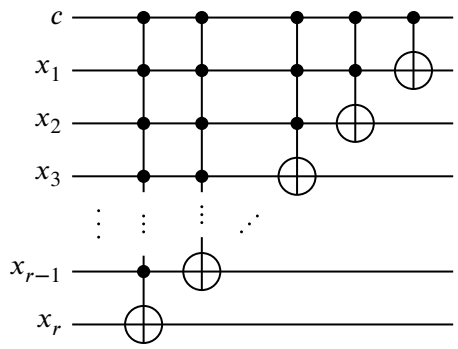


Figure 3.13: An incrementer circuit constructed from a sequence of Toffoli gates.

number of two-qubit gates when decomposed. In contrast, a DIPS-based circuit derived from the structure shown in Figure 3.10 accomplishes the same function but does not require any Toffoli gates at all, being already specified at the level of two-qubit gates.

With some modification, the counter circuit previously described can also be used to count the number of 1s from the outputs of any number of arbitrary Boolean functions. In other words, given Boolean functions f_1 through f_m, each of n variables x_1 through x_n, the counter circuit will output the base-two representation of

$$\sum_{i=1}^{m} f_i(x_1, \ldots, x_n) \tag{3.26}$$

where, as elsewhere in this chapter and Chapter 2, the sum denotes an arithmetic sum where the output of each Boolean function is simply treated as an integer, either 0 or 1. One way to construct such a function-counting circuit is to simply allocate m ancillary qubits, one to store the output of each function, and then feed these m qubits to a normal counter circuit. This has the disadvantage of requiring a number of ancillary qubits equal to the number of functions whose outputs are being counted. Instead, using the circuit structure shown in Figure 3.14, the 1s present among the outputs of f_1 through f_m can be counted using only

one ancillary qubit, not including the qubits used as the counter's final output. In Figure 3.14, each output y_j for $0 \leq j \leq k$ can be expressed as

$$y_j = X_j^{\sum_{i=1}^{m} f_i(x_1,\ldots,x_n)}(|0\rangle); \tag{3.27}$$

in other words, the circuit applies an X_j gate $\sum_{i=1}^{m} f_i(x_1, \ldots, x_n)$ times to the qubit labeled y_j. Figure 3.14 therefore performs the same overall operation as the first part of Figure 3.10 consisting of the "$\sum$" and "$\bigoplus$" blocks, but with x_1 through x_n replaced by arbitrary Boolean functions $f_1(x_1, \ldots, x_n)$ through $f_m(x_1, \ldots, x_n)$. Adding the second part of Figure 3.10 (consisting of the controlled-$X_i^{\pm j}$ gates) to the end of Figure 3.14 then produces the final output of the counter.

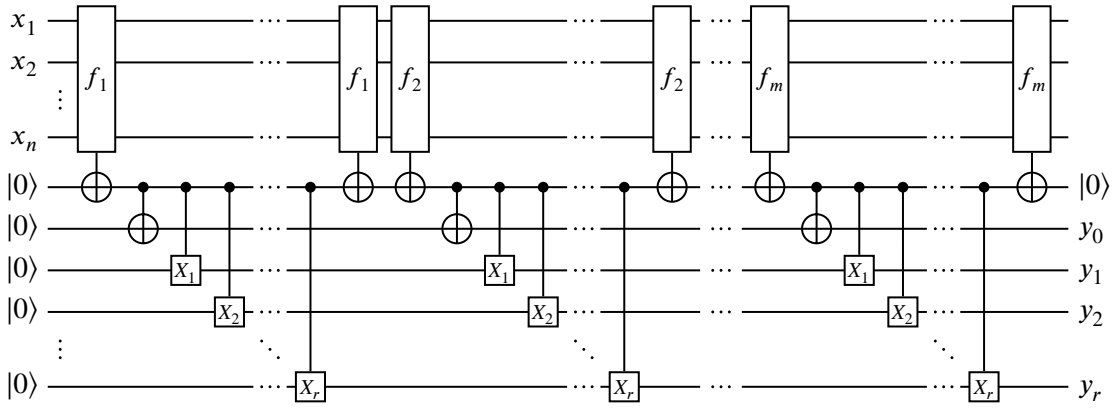


Figure 3.14: Circuit structure used to count 1s from the outputs of arbitrary functions using only one ancillary qubit. The controlled-$X_i^{\pm j}$ gates from the second part of Figure 3.10 must be added to the end of this structure to generate the final output of the counter.

3.4.2 Performance of the DIPS-based counter

Szyprowski and Kerntopf [14] demonstrated realizations for incrementer circuits of any size, like the one from Figure 3.13 (which they call a "generalized Peres gate"), using two-

qubit controlled gates. Their realizations have significantly lower quantum cost than that obtained by simply decomposing the individual Toffoli gates in Figure 3.13 into two-qubit gates. Interestingly, the approach used by Szyprowski and Kerntopf is similar in spirit to the approach I independently presented in this and the previous chapter for realizing DIPS, because it also directly uses controlled-2^k-th-root-of-NOT gates in a recursive circuit structure. However, Szyprowski and Kerntopf's objective is different than mine, because they are concerned only with the family of generalized Peres gates and not with symmetric functions at all. Although the method of Szyprowski and Kerntopf is targeted towards a different ultimate goal than the method I present here, it is worthwhile to compare the two methods since they both use two-qubit controlled-rotation gates and they may both be used to create counter circuits.

Quantum cost of counter based on Szyprowski and Kerntopf's method

Szyprowski and Kerntopf report a quantum cost of r^2 for a circuit equivalent to the single incrementer circuit shown in Figure 3.13 [14, Table I]. However, a counter circuit that counts the number of ones present among n input qubits requires n incrementer circuits, since each incrementer circuit can only count one of the input qubits. Therefore, the total quantum cost is r^2n. Furthermore, with n input qubits, the maximum possible number of 1s is of course n, so the output register of the counter requires $r = \lceil \log_2(n+1) \rceil$ qubits (since an r-qubit register is capable of representing integers up to $2^r - 1$). Therefore, using Szyprowski and Kerntopf's generalized Peres gates to implement a counter results in a total quantum cost of

$$r^2 n = \lceil \log_2(n+1) \rceil^2 n \tag{3.28}$$

when counting the 1s from n input qubits.

If the 1s are to be counted from the outputs of m Boolean functions instead of directly from input qubits, then a single ancillary qubit can be used, as shown in Figure 3.15. In this figure, the notation of a double line with slash is used to indicate that the line in the schematic represents a whole register of r qubits. In other words, this double line corresponds to the collection of qubits x_1 through x_r in Figure 3.13. We can see that m incrementer circuits are required, but addition, each of the included functions must be realized twice: once to compute its output and once as a mirror circuit to restore the ancillary qubit to its original value. Hence, if we let C_i denote the quantum cost of realizing the function f_i, then the quantum cost of the entire counter circuit becomes $2\sum_{i=1}^{m} C_i + r^2 m = 2\sum_{i=1}^{m} C_i + \lceil \log_2(m+1) \rceil^2 m$.

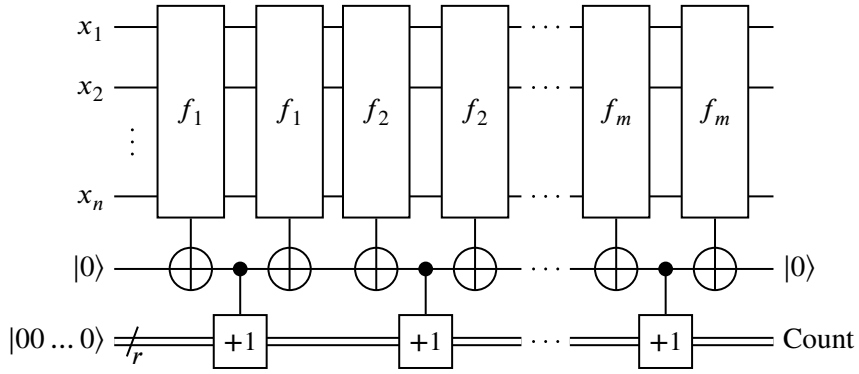


Figure 3.15: A counter of m Boolean functions using incrementer circuits.

Quantum cost of my method

Again, to count the 1s from n input qubits requires an output register of size $r = \lceil \log_2(n+1) \rceil$, since this is just a property of the base-two representation of integers and does not depend on any details of how the counter itself is implemented. As previously mentioned in Section 3.3.2, to match the behavior of an incrementer-based counter, we may want the outputs of the counter to be produced on completely separate qubits from the inputs, necessitating the use of the circuit structure from Figure 3.9 instead of Figure 3.10. For

Table 3.8: Quantum cost comparison of a DIPS-based counter and an incrementer-based counter using Szyprowski and Kerntopf's generalized Peres gates.

n	$r = \lceil \log_2(n+1) \rceil$	DIPS-based	Szyprowski & Kerntopf
3	2	7	12
4	3	15	36
5	3	18	45
10	4	46	160
15	4	66	240
20	5	110	500

a given k, the number of outputs produced in Figure 3.9 is $k + 1$; therefore, we require $k = r - 1$. Substituting into (3.24) then gives the total quantum cost of the counter as

$$r\left(n + \frac{r-1}{2}\right) = \lceil \log_2(n+1) \rceil \left(n + \frac{\lceil \log_2(n+1) \rceil - 1}{2}\right). \tag{3.29}$$

From Figure 3.14, we can see that my method also allows the counting of 1s from arbitrary functions by realizing each function twice and otherwise without any additional gates. Therefore, the cost of count the outputs from m Boolean functions, where C_i denotes the quantum cost of realizing the function f_i, is $2\sum_{i=1}^{m} C_i + r\left(n + (r-1)/2\right)$.

3.4.3 Comparison of costs

For both methods, the quantum cost of counting the outputs of a collection of Boolean functions is $2\sum_{i=1}^{m} C_i$ more than the cost of simply counting inputs. Therefore, we only need to compare the latter costs, which are given by (3.28) and (3.29). Table 3.8 shows that the quantum cost of the DIPS-based counter design is lower in all cases, and that the amount of improvement increases with the number of inputs n.

3.5 Conclusion

In this chapter, I demonstrated an extension to the realization method presented in Chapter 2 that allows for arbitrary sequences of positive and negative gates to be used as target gates. I introduced the concept of *offset* of a DIPS to characterize the types of symmetric functions that can be realized with this extension. I also presented a simplified and optimized circuit structure that performs the same function as the circuit structures from Chapter 2, but requires far fewer gates. This simplified circuit structure has the additional advantage that it is able to realize a whole sequence of symmetric functions at no additional cost compared to realizing only the final member of the sequence. In other words, when realizing a DIPS of a given order, the circuit structure also provides DIPS of lower orders "for free".

I demonstrated one potential application for the ability to realize multiple orders of DIPS at once with no additional cost, in the form of a *quantum counter* circuit that counts the number of 1s present among its inputs and outputs the result as an integer represented in base-two form. I further showed how this counter circuit could, with a minor modification, also count the 1s present among the outputs of any number of arbitrary Boolean functions while using only one additional ancillary qubit. The performance of these counter circuits, as measured by their quantum cost, was compared to that of a counter circuit based on Szyprowski and Kerntopf's optimized realization of a "generalized Peres gate", which is essentially the same as an incrementer circuit. Under the same conditions—only one ancillary qubit used and a gate library consisting of two-qubit controlled-root-of-NOT gates—the DIPS-based counter I demonstrated in this chapter gives quantum costs that are significantly lower.

Chapter 4

Realization of arbitrary symmetric functions using the Walsh-Hadamard transform

In the previous two chapters, I introduced a method for realizing any dyadic indicator-periodic symmetric function (DIPS) of any number of variables using only two-qubit gates. I furthermore demonstrated that it is possible to realize a whole sequence of DIPS of orders 0 through k, where k is some positive integer, all at once using an efficient circuit structure. This circuit structure has a quantum cost, and therefore execution time complexity, that grows linearly with the number of variables. The number of ancillary qubits required grows linearly with k. This means that, using only the methods presented in Chapters 2 and 3, it is already possible to realize many useful symmetric functions with low quantum cost. However, the set of all DIPS still does not include the majority of possible symmetric functions—any symmetric function whose indicator function does not follow the "2^k-on, 2^k-off" pattern that characterizes DIPS cannot be realized using only the methods from Chapters 2 and 3. For instance, $S_7^{0,1,3,6}(x_1, \ldots, x_7)$ is one such function.

The first objective of this chapter is to remedy the above shortcoming by showing how, using a method based on the Walsh-Hadamard transform, DIPS can be combined to realize any symmetric function of any number of variables. I show that this combining of DIPS can itself be done in a highly efficient manner that maintains the linear growth in quantum cost of the resulting circuits and requires only a logarithmically-growing number of ancillary

qubits (both with respect to the number of variables). In the second part of this chapter, I also show how the method presented in the first part can be made even more powerful through repetition of variables, which allows arbitrary Boolean functions, symmetric or not, to be realized by "symmetrizing" them. Remarkably, this symmetrization process can be done with absolutely no increase to the quantum cost of the final circuit whatsoever, at least in an idealized quantum computer where all gates can be performed with no error.

4.1 The Walsh-Hadamard transform

4.1.1 Walsh-Hadamard basis functions

The Walsh-Hadamard transform (a.k.a. simply the "Walsh" or "Hadamard" transform) [39, 40] is a discrete spectral transform which is similar to the Discrete Fourier Transform (DFT). Like the DFT, the Walsh-Hadamard transform decomposes an input function or signal into a linear combination of basis functions. In the DFT, the basis functions are complex exponentials, or equivalently combinations of sines and cosines, but in the Walsh-Hadamard transform the basis functions are all real-valued, and in fact only take on the values 1 and -1. Since the Walsh-Hadamard transform is a discrete transform, its input data is a discrete signal, which can also be thought of as a sequence of real numbers or a real-valued function whose domain is a subset of the natural numbers: given a sequence of real numbers $x_0, x_1, \ldots, x_n$, the corresponding function has domain $\{0, 1, \ldots, n\}$ and maps a natural number $i \leq n$ to x_i. The transform always operates on input data whose length is a power of two, and the number of basis functions is equal to the length of the input data.

Several definitions of the basis functions for the Walsh-Hadamard transform are possible, all of which are equivalent. Here I follow the definition given by Golubov *et al.* [40, §1.1]. Although Golubov *et al.* formulate the basis functions as continuous functions on a real

interval, their definition can easily be converted to a discrete version. For a transform operating on data of size 2^k, where k is a positive integer, let the 2^k basis functions be indexed by natural numbers from 0 through $2^k - 1$. Then, if i is a power of two, $i = 2^j$ where $j < k$, the i-th basis function h_i is defined as a square wave with a period of $2i = 2^{j+1}$. Specifically, using the notation from Chapters 2 and 3 where $\mathrm{b}_j(x)$ denotes the j-th bit of the base-two representation of x, we define

$$h_i(x) = (-1)^{\mathrm{b}_j(x)}. \tag{4.1}$$

If i is not a power of two, then we define h_i as a product of square waves of the form given in (4.1). Specifically, we have

$$h_i(x) = \prod_{j=0}^{k-1} h_{2^j}(x)^{\mathrm{b}_j(i)}. \tag{4.2}$$

In other words, $h_i(x)$ is a product of some subset of h_1, h_2, h_4, h_8, etc., where each of these is included in the product if and only if the corresponding bit in the base-two representation of i is 1, and the empty product (which occurs for $i = 0$) is considered to evaluate to 1. Figure 4.1 shows the basis functions for the example of $k = 3$, corresponding to a transform that operates on data of size 8. We may observe that the functions h_1, h_2, and h_4 are square waves with periods of 2, 4, and 8, respectively, as defined by (4.1), while all other f_i can be expressed as products of f_1, f_2, and f_4. For instance, for $i = 5$, the base-two representation of 5 is $5 = (101)_2$, so $h_5(x) = h_4(x)^1 h_2(x)^0 h_1(x)^1 = h_4(x)h_1(x)$. The above statement that an empty product is considered to evaluate to 1 is reflected in the fact that h_0 is just a constant function, $h_0(x) = 1$.

The basis functions of the Walsh-Hadamard transform are all orthogonal to each other,

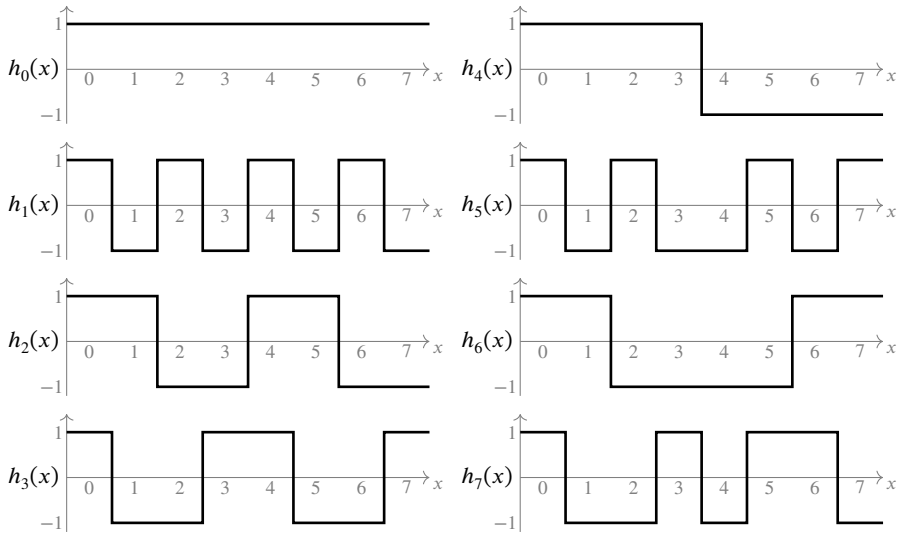


Figure 4.1: The Walsh-Hadamard basis functions for $k = 3$, corresponding to a transform operating on input data of size 8.

so given an input function or signal, its spectral coefficients can be obtained by taking its inner product with each of the basis functions. This is equivalent to arranging the basis functions as rows of a matrix and multiplying that matrix by the input signal represented as a column vector. For a transform operating on data of size 2^k, this matrix is therefore a $2^k \times 2^k$ matrix. It is known that this matrix is always symmetric [40, §1.3], which implies that its columns are also the Walsh-Hadamard basis functions and it is self-inverse up to a multiplicative normalization constant. The forward and inverse Walsh-Hadamard transforms are therefore identical up to this normalization constant.

As an example, consider the matrix representation of the Walsh-Hadamard transform

for $k = 3$, which is an 8×8 matrix

$$\mathbf{H}_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}. \tag{4.3}$$

Each row and column of this matrix has a squared norm of 8, so a normalization factor of $\frac{1}{8}$ must be applied to either the forward or inverse transform in order for the combined forward-inverse sequence to exactly recover the input data. For instance, given the input vector $\mathbf{v} = \begin{bmatrix} 1 & -2 & 3 & 0 & -1 & -5 & 1 & -2 \end{bmatrix}^\mathsf{T}$, matrix-vector multiplication gives

$$\mathbf{H}_8\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \\ 3 \\ 0 \\ -1 \\ -5 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} -5 \\ 13 \\ -9 \\ 1 \\ 9 \\ -1 \\ 1 \\ -1 \end{bmatrix}. \tag{4.4}$$

If $f_\mathbf{v}$ is the function representation of $\mathbf{v}$, *i.e.*, $f_\mathbf{v}(0) = 1$, $f_\mathbf{v}(1) = -2$, $f_\mathbf{v}(2) = 3$, etc., then (4.4) gives the decomposition of $f_\mathbf{v}$ into a weighted sum of Walsh-Hadamard basis functions:

$$\begin{aligned} f_\mathbf{v}(x) = \frac{1}{8}\big(&-5h_0(x) + 13h_1(x) - 9h_2(x) + h_3(x) \\ &+ 9h_4(x) - h_5(x) + h_6(x) - h_7(x)\big). \end{aligned} \tag{4.5}$$

4.2 Realization of arbitrary Boolean symmetric functions using the Walsh-Hadamard transform

4.2.1 Correspondence between Walsh-Hadamard basis functions and indicator functions of DIPS

The link between the Walsh-Hadamard transform and symmetric functions lies in the indicator functions of the DIPS defined and realized in Chapters 2 and 3. If we plot the indicator functions of zero-offset DIPS of the first few orders, as shown in Figure 4.2, we see a striking resemblance with some of the Walsh-Hadamard basis functions from Figure 4.1. Specifically, the indicator functions of $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$, $\mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$ appear to have the same "shape" as the Walsh-Hadamard basis functions h_1, h_2, and h_4 from Figure 4.1.

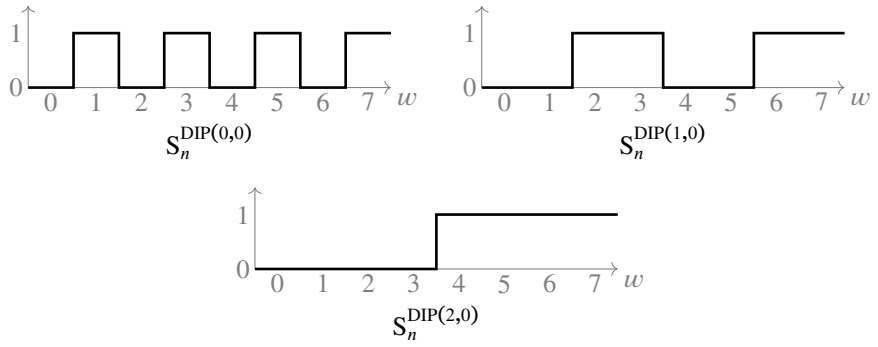


Figure 4.2: Indicator functions of $\mathrm{S}_n^{\mathrm{DIP}(j,0)}$ for $j = 0, 1, 2$, showing resemblance with Walsh-Hadamard basis functions h_1, h_2, and h_4 from Figure 4.1.

To make the notion of functions having the "same shape" more precise, observe that the correspondence between the aforementioned indicator functions and Walsh-Hadamard basis functions can be expressed a mapping between their output values. The indicator function of $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$, for instance, takes on a value of 0 (resp. 1) whenever h_1 takes on a value of 1 (resp. -1), and the same applies to the pairs of $\mathrm{S}_n^{\mathrm{DIP}(1,0)}$ with h_2 and $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$ and h_4. This

relationship becomes obvious when we consider the definition of a DIPS and the definition of the Walsh-Hadamard basis functions: $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$ has the indicator function $\mathrm{b}_k(w)$, while the Walsh-Hadamard basis function h_{2^k} is defined by $h_{2^k}(x) = (-1)^{\mathrm{b}_k(x)}$. We can express the relationship in equation form by defining a mapping $t : \{0, 1\} \to \{1, -1\}$ by

$$t(y) = (-1)^y. \tag{4.6}$$

Then for all natural numbers j we have

$$t\big(I^{\mathrm{DIP}(j,0)}(w)\big) = (-1)^{\mathrm{b}_j(w)} = h_{2^j}(w), \tag{4.7}$$

where $I^{\mathrm{DIP}(j,0)}(w)$ denotes the indicator function of $\mathrm{S}_n^{\mathrm{DIP}(j,0)}$, $I^{\mathrm{DIP}(j,0)}(w) = \mathrm{b}_j(w)$.

The mapping t defined in (4.6) is furthermore an isomorphism between the group $\{0, 1\}$ under modulo-2 addition, which is the same as the exclusive-OR operation, and the group $\{1, -1\}$ under multiplication. Since the Walsh-Hadamard basis functions whose indices are not powers of two are defined as products of those whose indices are powers of two, as per (4.2), the isomorphism t allow us to obtain a correspondence between the remaining Walsh-Hadamard basis functions and exclusive-ORs of indicator functions of DIPS. For instance, $h_5(x) = h_1(x)h_4(x)$, which corresponds to an exclusive-OR of indicator functions via t:

$$\begin{aligned} t\big(I^{\mathrm{DIP}(0,0)}(w) \oplus I^{\mathrm{DIP}(2,0)}(w)\big) &= t\big(\mathrm{b}_0(w) \oplus \mathrm{b}_4(w)\big) \\ &= t\big(\mathrm{b}_0(w)\big)t\big(\mathrm{b}_4(w)\big) \\ &= h_1(w)h_4(w). \end{aligned} \tag{4.8}$$

More generally, if i is any natural number, let $\text{Ones}(i)$ be the set consisting of the positions in the base-two representation of i where 1s occur. In other words, $\text{Ones}(i) = \{j \in \mathbb{N} \mid \text{b}_j(i) = 1\}$. Then h_i corresponds via t to the exclusive-OR of the indicator functions of the DIPS whose orders are in $\text{Ones}(i)$:

$$t\left(\bigoplus_{j \in \text{Ones}(i)} I^{\text{DIP}(j,0)}(w)\right) = t\left(\bigoplus_{j \in \text{Ones}(i)} \text{b}_j(w)\right)$$
$$= \prod_{j \in \text{Ones}(i)} t\big(\text{b}_j(w)\big) = \prod_{j \in \text{Ones}(i)} h_{2^j}(w) = h_i(w). \tag{4.9}$$

Thus, every Walsh-Hadamard basis function corresponds to the exclusive-OR of the indicator functions of some set of DIPS. The constant function h_0 corresponds to the exclusive-OR of the empty set, that being conventionally defined as 1 in that same way that an empty product in (4.2) is defined as 1.

The correspondence between Walsh-Hadamard basis functions and indicator functions of DIPS can be exploited to create circuits that realize Walsh-Hadamard basis functions as an effective number of gates applied to a target qubit. Consider the circuit shown in Figure 4.3, where U is an arbitrary single-qubit gate. If the function $\text{S}^{\text{DIP}(j,0)}(x_1, \ldots, x_n)$ evaluates to 1, then the circuit applies a U gate followed by a U^{-2} gate to qubit y, therefore applying an effective number of -1 U gates. If $\text{S}^{\text{DIP}(j,0)}(x_1, \ldots, x_n) = 0$, then the controlled-$U^{-2}$ gate is inactive, so the circuit applies only the single uncontrolled U gate, which amounts to an effective number of 1. The mapping from the output of $\text{S}^{\text{DIP}(j,0)}(x_1, \ldots, x_n)$ to the effective number of U gates applied to y is the same as t from (4.6), so the circuit shown in Figure 4.3 in fact applies an effective number of U gates equal to $h_{2^j}(w)$, where w is the input weight $w = \sum x_i$. We can represent the operation of the circuit from Figure 4.3 using an operational

equation in the same spirit as those from Chapters 2 and 3,

$$1 - 2\mathrm{b}_j(w) = t\big(\mathrm{b}_j(w)\big) = h_{2^j}(w). \tag{4.10}$$

This equation shows that the circuit effectively implements the mapping t using $t(y) = 1-2y$, where the constant term 1 corresponds to the uncontrolled U gate and the factor of -2 in the second term corresponds to the target gate of the controlled gate being U^{-2}.

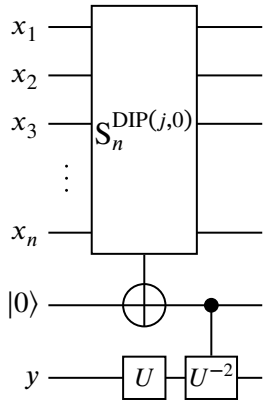


Figure 4.3: A circuit that uses the correspondence between indicator functions of DIPS and Walsh-Hadamard basis functions to apply an effective number of U gates to qubit y equal to $h_{2^j}(w)$, where $w = \sum x_i$ is the input weight.

The circuit from Figure 4.3 can also be represented by the equivalent circuit of Figure 4.4, which uses the same "double-bar" notation from Figures 2.8 and 3.6. We recall that this notation indicates that $h_{2^j}(w)$ is not a normal control function that simply activates or deactivates the target gate U, but rather is an integer-valued function that controls the effective number of U gates applied to y and can take on values other than 0 and 1.

A similar construction to Figure 4.3 can be used for Walsh-Hadamard basis functions with indices other than powers of two. For instance, based on (4.8), we can construct the circuit shown in Figure 4.5, which applies to y an effective number of U gates equal to

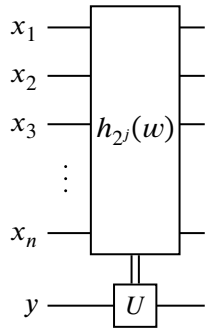


Figure 4.4: Equivalent representation of the circuit from Figure 4.3 using the "double-bar" notation from Chapters 2 and 3 to represent an effective number of U gates applied to qubit y.

$h_5(w)$. This circuit has the operational equation

$$1 - 2(\mathrm{b}_0(w) \oplus \mathrm{b}_2(w)) = t\big(\mathrm{b}_0(w) \oplus \mathrm{b}_4(w)\big) = h_{2^j}(w) \tag{4.11}$$

and can be represented by an equivalent circuit like the one in Figure 4.4 with the control function replaced by $h_5(w)$.

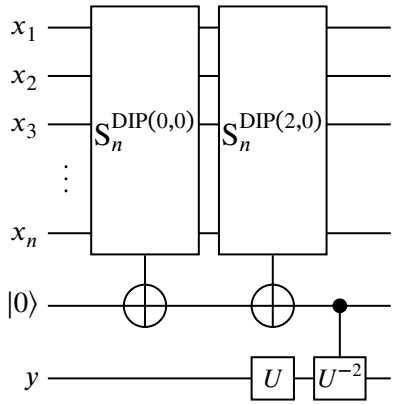


Figure 4.5: A circuit analogous to Figure 4.3 using the exclusive-OR of two DIPS to apply an effective number of U gates to qubit y equal to $h_5(w)$.

Since every Walsh-Hadamard basis function corresponds to an exclusive-OR of indicator functions of some set of DIPS, as given by (4.9), every basis function can be realized as an effective number of U gates applied to a target qubit y using circuits like the ones shown in

Figures 4.3 and 4.5.

4.2.2 Realization of single-output symmetric functions

Figure 4.5 is drawn in a way that suggests that the two DIPS $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ and $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$ are independently realized by separate circuits, but this does not need to be the case. Recall that the circuit structure derived in Chapter 3 and shown in Figure 3.10 simultaneously realizes DIPS of all orders from 0 to k, and in particular can realize $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ through $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$ when $s = 0$. If this circuit structure is first used to realize the DIPS $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ through $\mathrm{S}_n^{\mathrm{DIP}(k,0)}$, then we can obtain the exclusive-OR of any subset of those DIPS using only CNOT gates. This in turn allows any Walsh-Hadamard basis function up to $h_{2^{k+1}-1}$ to be obtained using a Figure 4.5-like construction, but without needing to realize the DIPS separately. For instance, suppose that we have already used an instance of Figure 3.10 with $k = 2$ and $s = 0$ to simultaneously realize $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$, $\mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$. Then the effect of the circuit from Figure 4.5 can be achieved using the very simple circuit shown in Figure 4.6, which consists of only three two-qubit controlled gates.

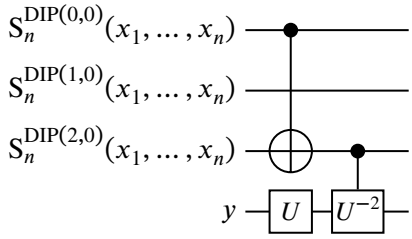


Figure 4.6: A circuit that achieves the same effect as the one from Figure 4.5, assuming that an instance of Figure 3.10 has first been used to realize $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ through $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$.

By concatenating circuits like the one from Figure 4.6, we can effectively create linear combinations of Walsh-Hadamard basis functions, which through the Walsh-Hadamard transform allows us to reproduce any desired function of the input weight. In particular,

the indicator function of any desired symmetric function can be decomposed into a linear combination of Walsh-Hadamard basis functions and each of those basis functions realized using a Figure 4.6-like circuit. This process allows for the realization of any symmetric function whatsoever.

To illustrate the process described above in more detail, consider the symmetric function $\mathrm{S}^{0,1,3,6}(x_1, \ldots, x_7)$, which was given at the beginning of this chapter as an example of a function that is not a DIPS and so cannot be realized using only the methods presented in Chapters 2 and 3. The indicator function of this symmetric function can be represented in vector form as $\mathbf{v} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}^{\mathsf{T}}$, and its Walsh-Hadamard spectral coefficients are therefore given by

$$\mathbf{H}_8\mathbf{v} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ 0 \\ 0 \\ 2 \\ -2 \\ 2 \\ 2 \end{bmatrix}. \tag{4.12}$$

The indicator function $I^{0,1,3,6}$ therefore has the representation

$$I^{0,1,3,6}(w) = \frac{1}{8}\big(4 + 2h_4(w) - 2h_5(w) + 2h_6(w) + 2h_7(w)\big) \tag{4.13}$$

as a linear combination of Walsh-Hadamard basis functions. Multiplying (4.13) by 8 and switching the two sides gives

$$4h_0(w) + 2h_4(w) - 2h_5(w) + 2h_6(w) + 2h_7(w) = 8I^{0,1,3,6}(w), \tag{4.14}$$

which can be used as an operational equation that specifies exactly what circuits of the form shown in Figure 4.6 should be combined to realize the desired symmetric function. If we think of this operational equation as representing the total number of W gates applied to a target qubit y, where the identity of W is yet to be determined, then the first term of (4.14) represents an uncontrolled W^4 gate acting on the target qubit y because this always adds 4 to the total effective number of W gates. The second term, $2h_4(w)$, represents a subcircuit that applies 2 W gates, or equivalently a single W^2 gate, to qubit y when $h_4(w) = 1$ and applies a W^{-2} gate otherwise. Such a subcircuit can be created by removing the CNOT gate from Figure 4.6 and letting $U = W^2$ (which implies $U^{-2} = W^{-4}$). Similarly, the third term, $-2h_5(w)$, of (4.14) represents a subcircuit that applies a W^{-2} gate to y when $h_4(w) = 1$ and a W^2 gate otherwise. This subcircuit is obtained by letting $U = W^{-2}$ in Figure 4.6. Continuing in this manner allows the whole of (4.14) to be translated into a sequence of Figure 4.6-like subcircuits where U is a possibly different power of W in each subcircuit.

It remains to determine the identity of W. The right-hand side of (4.14) tells us that a circuit constructed as described in the previous paragraph will apply a total effective number of $8I^{0,1,3,6}(w)$ W gates to its target qubit y. Since our goal is to realize the symmetric function $\mathrm{S}^{0,1,3,6}(x_1, \ldots, x_7)$, this effective number of W gates should be equivalent to a NOT gate when $I^{0,1,3,6}(w) = 1$. In other words, we should have $W^8 = X$, which means that W is an eighth-root-of-NOT gate, $W = X_3$. Figure 4.7 shows the final circuit that realizes $\mathrm{S}^{0,1,3,6}(x_1, \ldots, x_7)$ assuming that $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$, $\mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $\mathrm{S}_n^{\mathrm{DIP}(2,0)}$ are available at the start. These three DIPS of course have to be realized using an instance of the circuit structure from Figure 3.10. The full circuit that realizes $\mathrm{S}^{0,1,3,6}(x_1, \ldots, x_7)$ therefore consists of an instance of Figure 3.10 with $k = 2$ and $s = 0$, followed by the circuit shown in Figure 4.7. I will refer to the former as the *DIPS-initialization* stage and the latter as the *output stage*.

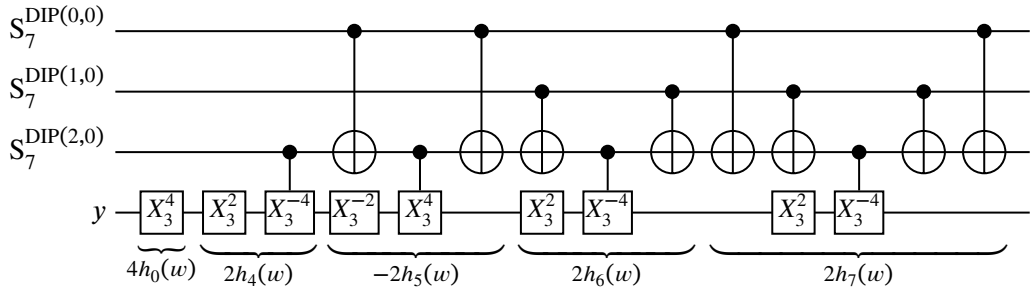


Figure 4.7: Output stage for realization of $S^{0,1,3,6}(x_1, \ldots, x_7)$, with subcircuits corresponding to terms of (4.14) labeled.

The procedure demonstrated by the preceding example readily applies to symmetric functions with more variables. With a 7-input function like $S^{0,1,3,6}(x_1, \ldots, x_7)$, the vector representation of the indicator function contains 8 components, so the Walsh-Hadamard transform of size 8 is sufficient to handle 7-input symmetric functions. A transform of size 16 is needed to handle symmetric functions with up to 15 inputs, a transform of size 32 is needed for up to 31 inputs, and so on. When using a Walsh-Hadamard transform of size 2^k, zero-offset DIPS of orders of 0 through $k-1$ are needed to be able to realize all of the Walsh-Hadamard basis functions. Since a transform of size 2^k handles functions with up to $2^k - 1$ inputs, we see that the number of DIPS needed grows only logarithmically with the number of inputs to the symmetric function being realized.

Note that if the vector representation of a symmetric function's indicator function does not have a size that is a power of two, it can be padded with zeroes to increase its size to next power of two. For instance, a 5-variable symmetric function, say $S_5^{1,2,4}$, has an indicator function that is represented in vector form as $\begin{bmatrix}0 & 1 & 1 & 0 & 1 & 0\end{bmatrix}^T$, which only has 6 components. To apply the Walsh-Hadamard transform to this function, we pad it with zeroes to form the 8-component vector $\begin{bmatrix}0 & 1 & 1 & 0 & 1 & 0 & 0 & 0\end{bmatrix}^T$.

One issue that was not addressed in Figure 4.7 is the question of how to most efficiently

create the exclusive-ORs of subsets of DIPS needed to realize the Walsh-Hadamard basis functions. In Figure 4.7 this was simply done by generating the exclusive-OR for each basis function individually and immediately using a mirror circuit to restore the original values of $S_n^{DIP(0,0)}$ through $S_n^{DIP(2,0)}$ in between subcircuits, but this is not the most efficient method.

In the general case, for a given value of k, the DIPS used are $S_n^{DIP(0,0)}$ through $S_n^{DIP(k-1,0)}$. There are 2^k possible subsets of these DIPS, and the exclusive-ORs of all of them might potentially be needed if every Walsh-Hadamard basis function has a nonzero coefficient. (This was not the case in 4.7, as the coefficients of h_1 through h_3 were zero, so only 5 out of the 8 possible exclusive-ORs were used.) To see how these exclusive-ORs might be generated in the most efficient way possible, first consider the case where $k = 1$, so that only $S_n^{DIP(0,0)}$ is present. This is trivial, since then the only exclusive-OR of interest is just this one function itself, so no additional controlled-NOT gates are needed. Next, for $k = 2$, two functions are present, $S_n^{DIP(0,0)}$ and $S_n^{DIP(1,0)}$. I will label these as g_0 and g_1, respectively, for convenience. Then the only exclusive-OR that needs to be generated is $g_0 \oplus g_1$, which is easily achieved with a single controlled-NOT gate, plus a second controlled-NOT gate that restores both qubits to their original values. This case is shown in Figure 4.8.

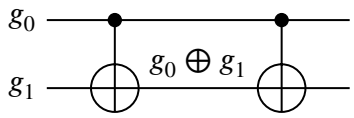


Figure 4.8: Computation of all exclusive-ORs of two functions.

It might appear that Figure 4.8 does not accomplish anything useful, since the two controlled-NOT gates clearly cancel, but the key is that the value $S_n^{DIP(0,0)} \oplus S_n^{DIP(1,0)}$ is created between the two controlled-NOT gates, where other gates can be inserted to use this value as a control input.

Continuing to the case $k = 3$, we now have three functions—$g_0 = S_n^{DIP(0,0)}$, $g_1 =$

$\mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $g_2 = \mathrm{S}_n^{\mathrm{DIP}(2,0)}$—and we need to generate four nontrivial exclusive-ORs: $g_0 \oplus g_1$, $g_0 \oplus g_2$, $g_1 \oplus g_2$, and $g_0 \oplus g_1 \oplus g_2$. We can use the existing circuit from Figure 4.8 to generate $g_0 \oplus g_1$, leaving only the exclusive-ORs containing g_2 to be generated. These can be generated by making use of a cyclic Gray code: a sequence of binary words where each word differs in only one bit from the next, the last word in the sequence differs in only one bit from the first, and each possible word appears once. In this case, we can use a 2-bit Gray code, one possibility being given by the sequence 00, 01, 11, 10. The 2-bit words of this sequence will correspond to combinations of g_0 and g_1: 00 indicates that neither g_0 nor g_1 is present, giving only g_2, while 01 indicates that g_0 is not present but g_1 is, giving $g_1 \oplus g_2$. Since each word of the Gray code differs in only one bit from the next, each corresponding exclusive-OR only differs from the previous one in the presence or absence of a single function g_i. Therefore, each of these exclusive-ORs may be obtained by either adding or removing a single g_i from the previous one, which can be accomplished with a single controlled-NOT gate. The circuit resulting from this procedure is shown in Figure 4.9.

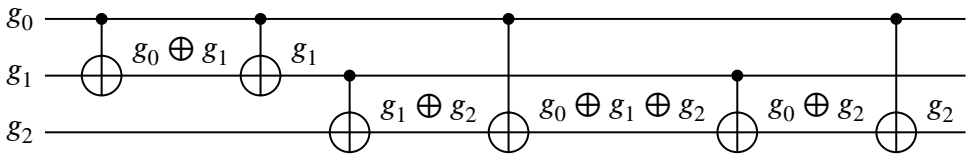


Figure 4.9: Computation of all exclusive-ORs of three functions.

The first two gates in Figure 4.9 are the same as in Figure 4.8. The remaining gates are obtained using the Gray code as described above: the first two words of the code are 00 and 01, which differ in only the second bit, so we add a controlled-NOT gate with target g_2 and control g_1, which adds g_1 to the subset of functions whose exclusive-OR is currently being generated. The next word is 11, which differs from 01 in the first bit, so we use a controlled-NOT gate with target g_2 and control g_0 to add g_0 to this subset. After that is 10,

which differs from 11 in the second bit, so we use a controlled-NOT gate with target g_2 and control g_1 again, which now removes g_1 from the subset. Finally, one more controlled-NOT gate is used to restore the last qubit to its original value of g_2.

The following theorem shows that the above-described procedure using a Gray code can be extended to any number of functions or variables whose exclusive-ORs are to be computed, and also gives a formula for the number of controlled-NOT gates required.

Theorem 13. *Given n qubits x_1 through x_n, it is possible to construct a quantum circuit that consists of $2^n - 2$ controlled-NOT gates and has the following properties:*

1. *The circuit uses only the n provided qubits and no others;* i.e.*, it uses no ancillary qubits.*

2. *The circuit leaves all n qubits in the same states in which they started.*

3. *For each nonempty subset of $\{x_1, \ldots, x_n\}$, the exclusive-OR of this subset appears on one of the qubits at some point in the circuit.*

Proof. We proceed by induction on n. The case $n = 1$ is trivial, since in that case the only subset is $\{x_1\}$ itself and no gates are required. Since $2^1 - 2 = 0$, the statement of the theorem is satisfied.

Now suppose that statement of the theorem is satisfied for some n, so that $2^n - 2$ controlled-NOT gates suffice to compute the exclusive-OR of every nonempty subset of $\{x_1, \ldots, x_n\}$. Take a circuit that performs this task and call it Subcircuit 1. By definition, Subcircuit 1 consists of $2^n - 2$ controlled-NOT gates.

Let the sequence $G_1, G_2, \ldots, G_{2^n}$ be a cyclic Gray code on n bits, where each G_i is an n-bit binary word and $G_1 = 000 \ldots 0$. Since $\{G_i\}$ is a Gray code, G_i and G_{i+1} differ in exactly one bit for $1 \leq i \leq 2^n - 1$, and since $\{G_i\}$ is also cyclic, G_n and G_1 also differ in exactly

one bit. Define a function $d(i)$ as follows: $d(i)$ is the position of the bit that differs between G_i and G_{i+1} for $1 \leq i \leq 2^n - 1$, and $d(2^n)$ is the position of the bit that differs between G_n and G_1. For instance, if G_1 and G_2 differ in their first bits, then $d(1) = 1$. From this definition, we always have $1 \leq d(i) \leq n$. Construct an $n+1$-qubit quantum circuit with 2^n controlled-NOT gates, where the i-th gate in this circuit targets qubit x_{n+1} and is controlled by qubit $x_{d(i)}$. Call this circuit Subcircuit 2.

I claim that Subcircuit 2 has the following property (call it Property A for convenience): for $1 \leq i \leq n$, just prior to the i-th gate of the circuit, qubit x_{n+1} contains the exclusive-OR of x_{n+1} together with the subset of $\{x_1, \ldots, x_n\}$ indicated by the word G_i. By "indicated by the word G_i", I mean that this subset contains a particular x_j if and only if the corresponding bit in G_i is 1. For instance, the word $110 \ldots 0$ indicates the subset $\{x_1, x_2\}$, since only the first two bits of the word have the value 1. In other words, if $G_i = g_{i,1} g_{i,2} \ldots g_{i,n}$, then I claim that just prior to the i-th gate in Subcircuit 2, qubit x_{n+1} contains the value

$$x_{n+1} \oplus \left(\bigoplus_{1 \leq j \leq n} g_{i,j} \, x_j \right). \tag{4.15}$$

Property A is certainly true for $i = 1$: $G_1 = 00 \ldots 0$ by definition, which indicates the empty set, and at the start of the circuit, the qubit x_{n+1} does indeed contain the exclusive-OR of x_{n+1} together with the empty subset of $\{x_1, \ldots, x_n\}$, this exclusive-OR being just x_{n+1} itself. Furthermore, if Property A is true for a particular i, then, since the i-th gate of Subcircuit 2 targets x_{n+1} and is controlled by $x_{d(i)}$, qubit x_{n+1} will contain the value

$$x_{n+1} \oplus \left(\bigoplus_{\substack{1 \leq j \leq n \\ j \neq d(i)}} g_{i,j} \, x_j \right) \oplus \overline{g_{i,d(i)}} \, x_{d(i)}, \tag{4.16}$$

where $\overline{g_{i,d(i)}}$ denotes the logical negation of $g_{i,d(i)}$, $\overline{g_{i,d(i)}} = \neg g_{i,d(i)}$. Since, by definition, G_i and G_{i+1} differ in their $d(i)$-th bits and no others, we have

$$g_{i+1,j} = \begin{cases} g_{i,j} & \text{if } j \neq d(i), \\ \overline{g_{i,j}} = \overline{g_{i,d(i)}} & \text{otherwise.} \end{cases} \tag{4.17}$$

Substituting (4.17) into (4.16)—in other words, replacing both $g_{i,j}$ in the second term of (4.17) and $\overline{g_{i,d(i)}}$ in the last term with $g_{i+1,j}$—we obtain

$$x_{n+1} \oplus \left(\bigoplus_{1 \leq j \leq n} g_{i+1,j} \, x_j \right), \tag{4.18}$$

which is the value of qubit x_{n+1} following the i-th gate and before the $(i+1)$-th gate of Subcircuit 2, which means that Property A is true for $i+1$ as well. Therefore, Property A is true for $1 \leq i \leq n$. Figure 4.11 illustrates how (4.15) being true prior to the i-th gate of Subcircuit 2 implies that (4.18) is true following that gate.

In addition, since G_n and G_1 differ only in their $d(n)$-th bits, an identical argument to the above shows that qubit x_{n+1} contains the value

$$x_{n+1} \oplus \left(\bigoplus_{1 \leq j \leq n} g_{1,j} \, x_j \right) = x_{n+1} \tag{4.19}$$

after the n-th and final gate of Subcircuit 2, which means that at the end of Subcircuit 2, all qubits retain their original values from the start of the circuit. (Qubits x_1 through x_n are unchanged since they are only used as control qubits in Subcircuit 2.)

To summarize, Property A implies that, for $1 \leq i \leq n$, qubit x_{n+1} attains a value equal to the exclusive-OR of x_{n+1} together with the subset of $\{x_1, \ldots, x_n\}$ indicated by the word G_i at some point in Subcircuit 2. Since $\{G_i\}$ is an n-bit Gray code, this means that all possible n-bit

binary words appear as one of the G_i, so that qubit x_{n+1} attains the values of all possible exclusive-ORs of x_{n+1} with a subset of $\{x_1, \ldots, x_n\}$ at some point in Subcircuit 2. This is the same as saying that qubit x_{n+1} attains the exclusive-OR of each subset of $\{x_1, \ldots, x_{n+1}\}$ containing x_{n+1}.

To complete the induction step, construct a $n+1$-qubit circuit by concatenating Subcircuits 1 and 2. (Subcircuit 1 was defined as an n-qubit circuit, so it operates on qubits x_1 through x_n while ignoring qubit x_{n+1} completely.) Since Subcircuit 1 contains $2^n - 2$ gates and Subcircuit 2 contains 2^n gates, this new circuit contains $2^n + 2^n - 2 = 2^{n+1} - 2$ controlled-NOT gates. Furthermore, by the induction hypothesis, the exclusive-ORs of all nonempty subsets of $\{x_1, \ldots, x_n\}$ are attained at some point in Subcircuit 1—these are the same as the nonempty subsets of $\{x_1, \ldots, x_{n+1}\}$ that do not contain x_{n+1}. Property A then shows that the exclusive-ORs of all subsets of $\{x_1, \ldots, x_{n+1}\}$ that do contain x_{n+1} are attained at some point in Subcircuit 2. Therefore, the exclusive-ORs of all nonempty subsets of $\{x_1, \ldots, x_{n+1}\}$ are attained at some point in this new circuit. In addition, both Subcircuits 1 and 2 individually leave all qubits in the same states in which they started, so the combination of the two subcircuits also leaves all the qubits in their starting states. □

To help illustrate the use of this theorem, I now show a few cases for small n. As mentioned in the proof, $n = 1$ is trivial. For $n = 2$, the proof instructs us to take the circuit for $n = 1$ as Subcircuit 1, which in this case is empty. We then consider a 1-bit Gray code starting with 0, which is trivially given by the sequence $0, 1$. We have $d(1) = d(2) = 1$ since the code is on only one bit. Therefore, the final circuit consists of two identical controlled-NOT gates with control qubit x_1 and target x_2, as shown in Figure 4.8.

For $n = 3$, we take the circuit from Figure 4.8 as Subcircuit 1, and use a 2-bit Gray code to generate Subcircuit 2. One possible Gray code is given by the sequence $00, 01, 11, 10$,

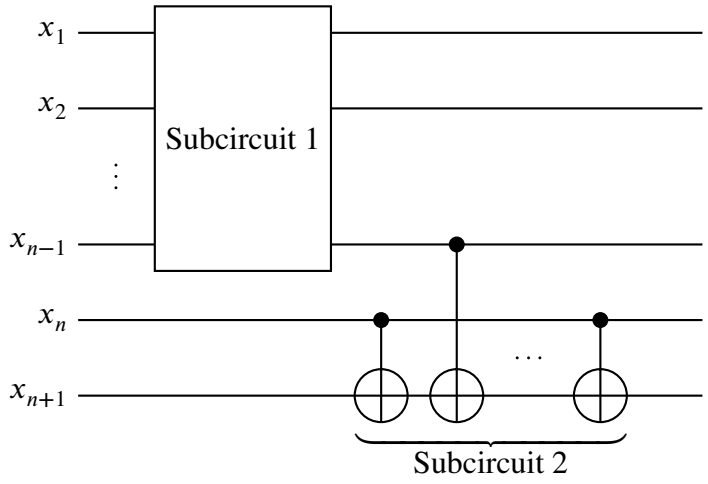


Figure 4.10: Circuit to compute all exclusive-ORs of $n + 1$ variables as described in the proof of Theorem 11: the circuit for n variables is recursively used as Subcircuit 1, while the additional Subcircuit 2 computes exclusive-ORs containing the new variable x_{n+1}.

which gives $d(1) = 2$, $d(2) = 1$, $d(3) = 2$, and $d(4) = 1$. Then, Subcircuit 2 consists of a sequence of four controlled-NOT gates, all targeting x_3, where the i-th gate of this sequence is controlled by $x_{d(i)}$. This leads to the circuit shown in Figure 4.9. Notice that the first two gates in this circuit, which make up Subcircuit 1, compute the exclusive-OR of x_1 and x_2, while the remaining gates, which make up Subcircuit 2, compute all of the exclusive-ORs that contain x_3. In addition, the order in which the exclusive-ORs containing x_3 appear corresponds to the Gray code used: first, x_3 is present at the start, corresponding to 00 (neither x_1 nor x_2 are present); second, $x_2 \oplus x_3$ appears, corresponding to 01 (x_1 not present but x_2 is); third, the exclusive-OR of all three variables appears, corresponding to 11 (both x_1 and x_2 present); and finally, $x_1 \oplus x_3$ appears, corresponding to 10 (x_1 present but x_2 is not). The fact that x_3 is restored at the end also corresponds to the fact that the Gray code is cyclic, so that 10 can be followed by 00 again.

The case $n = 4$ is similar to $n = 3$, and Subcircuit 2 can be generated using the Gray code 000, 001, 011, 010, 110, 111, 101, 100, which results in the circuit shown in Figure 4.12.

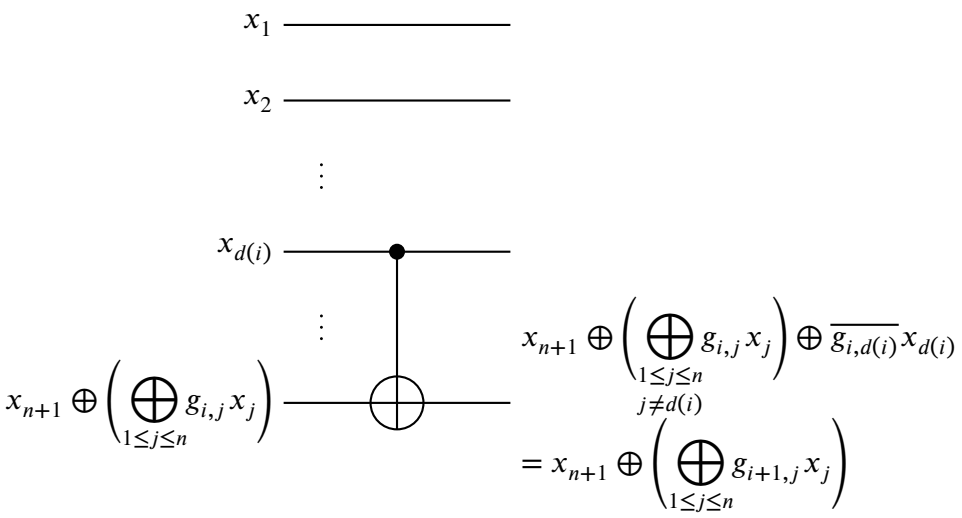


Figure 4.11: Illustration of (4.15) through (4.18) in the proof of Theorem 13.

In Figure 4.12, the exclusive-ORs computed by Subcircuit 1 are not shown because the subcircuit is identical to the $n = 3$ circuit shown in Figure 4.9. We may observe that the circuits for $n = 2$, $n = 3$, and $n = 4$ require 2, 6, and 14 gates, respectively, in accordance with Theorem 13.

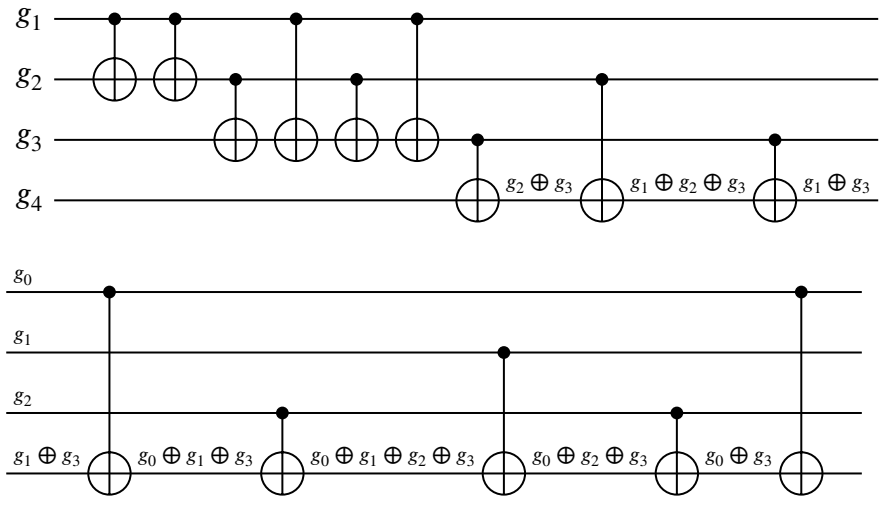


Figure 4.12: Computation of all exclusive-ORs of four variables.

Using the circuit from Figure 4.9 as a template, we can rearrange Figure 4.7 to reduce

the number of CNOT gates required. Specifically, keeping in mind that $g_0 = \mathrm{S}_n^{\mathrm{DIP}(0,0)}$, $g_1 = \mathrm{S}_n^{\mathrm{DIP}(1,0)}$, and $g_2 = \mathrm{S}_n^{\mathrm{DIP}(2,0)}$, the exclusive-ORs of DIPS used in Figure 4.7 correspond to g_2, $g_0 \oplus g_2$, $g_1 \oplus g_2$, and $g_0 \oplus g_1 \oplus g_2$. This leads to the circuit shown in Figure 4.13. Note that the parts of the circuit corresponding to individual Walsh-Hadamard basis functions are not in the same order as in Figure 4.7. This is a consequence of the use of a Gray code in generating Figure 4.9.

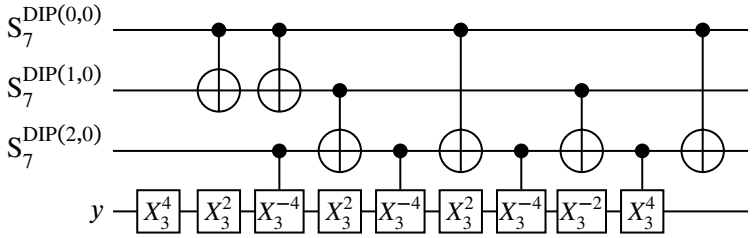


Figure 4.13: The output stage from Figure 4.7 with number of CNOT gates reduced using the scheme of Figure 4.9.

Figure 4.13 can be further simplified by removing the first two CNOT gates, which are unused and cancel. These CNOT gates were taken from the first two gates of Figure 4.9, and they would be used if the Walsh-Hadamard spectral coefficient of h_3, which corresponds to $g_0 \oplus g_1 = \mathrm{S}_n^{\mathrm{DIP}(0,0)} \oplus \mathrm{S}_n^{\mathrm{DIP}(1,0)}$, were nonzero. In addition, all of the uncontrolled gates on qubit y can be combined into a single gate, giving the circuit shown in Figure 4.14. Note that in this figure, X_3^8 is just a NOT gate and X_3^4 and X_3^{-4} are V and $V^\dagger$ gates, respectively, but I have chosen to keep the X_3^m notation to emphasize the derivation of these gates from Figure 4.7 and ultimately from the coefficients in (4.14). Of course, all of the procedures described here can be applied to any other symmetric function as well, giving an output-stage circuit with a similar form to Figure 4.14.

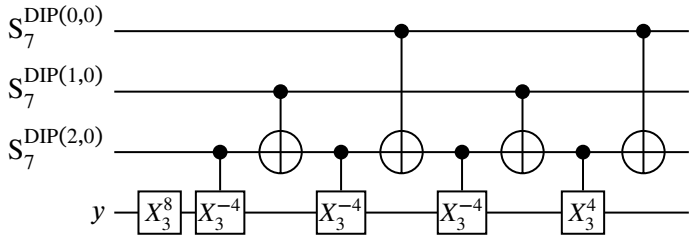


Figure 4.14: Further simplification of Figure 4.13 by removing unused CNOT gates and combining all uncontrolled gates acting on y.

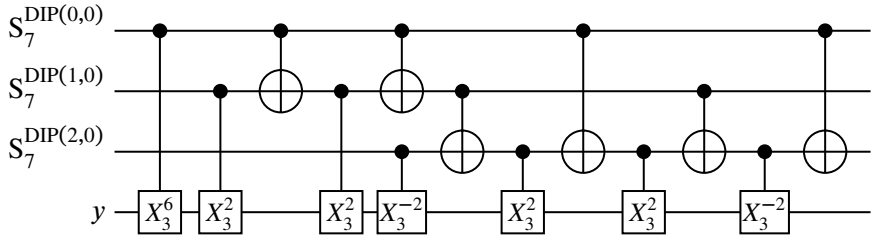
4.2.3 Realization of multi-output symmetric functions

The circuit structure exemplified by Figures 4.7, 4.13, and 4.14 lends itself very well to the realization of multi-output symmetric functions. A multi-output Boolean symmetric function is one that satisfies the same condition of symmetry defined in Chapter 2, but produces multiple Boolean value as outputs. It is thus mathematically equivalent to a collection of several single-output symmetric functions. The additional challenge in realizing multi-output symmetric functions is to find methods of sharing the same subcircuits between different outputs, as doing so reduces the complexity of the resulting circuits compared to realizing each output independently as a separate function.

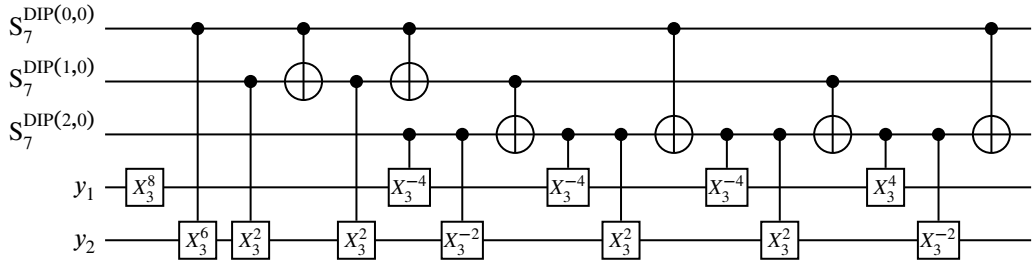
The realization method for single-output symmetric functions described in Section 4.2.2 allows for drastic sharing of subcircuits if used for multi-output functions. Specifically, the entire DIPS-initialization stage can be shared between all outputs, since it is always the same for every symmetric function with a given number of inputs—it is always just an instance of Figure 3.10 with $s = 0$ and with k being sufficiently large to handle a Walsh-Hadamard transform of the size needed for that number of inputs. Part of the output stage can be shared as well, in particular the arrangement of CNOT gates used to compute all possible exclusive-ORs of the DIPS being used, as shown in Figures 4.9 and 4.12. Therefore, the only parts of the circuit that differ between outputs are the gates that directly target the output

qubits, like the five gates targeting qubit y in Figure 4.14.

As an example of the reuse of circuit components possible with the realization method described in Section 4.2.2, consider now realizing a two-output symmetric function in which the first output is the same function $S_7^{0,1,3,6}$ from before and the second output is $S_7^{1,2,3,5,7}$. The Walsh-Hadamard spectral coefficients of the second output are found to be (in order) 5, -3, -1, -1, 1, 1, -1, and -1, and applying the same procedure used for $S_7^{0,1,3,6}$ eventually produces the output stage shown in Figure 4.15a. This circuit can be combined with the one from Figure 4.14 to form the output stage for the two-output symmetric function shown in Figure 4.15b.



(a) Output stage for $S_7^{1,2,3,5,7}$.



(b) Combined output stage for a two-output symmetric function incorporating (a) together with Figure 4.14.

Figure 4.15: Example of realization of a two-output symmetric function showing reuse of DIPS and CNOT gates.

While the reuse of a few CNOT gates in Figure 4.15b may not seem like much, one should also keep in mind that the entire DIPS-initialization stage that produces the three functions $S_7^{DIP(0,0)}$, $S_7^{DIP(1,0)}$, and $S_7^{DIP(2,0)}$, which consists of an instance of Figure 3.10 with

$n = 7$, $k = 2$ and $s = 0$, is reused as well because the DIPS themselves are shared between the two outputs. In other words, realizing the two outputs independently would require two copies of the DIPS-initialization stage while the realization using Figure 4.15b only requires one. Therefore, Figure 4.15b in fact demonstrates a significant reduction in quantum cost when realizing the two outputs together as compared to realizing them independently as separate functions. Another way to see the efficiency of this multi-output realization method is to note that Figure 4.15b contains 18 gates as compared to 9 gates in Figure 4.14 and 13 gates in Figure 4.15a. Hence, an entirely new output has been added to a 7-input symmetric function for the cost of less than 10 additional two-qubit gates.

4.2.4 Algorithms for generating circuits to realize symmetric functions

I now formulate the methods described in this chapter in terms of two algorithms that together generate the realization of a symmetric function from its specification. First, the constructive proof of Theorem 13 leads directly to an algorithm for generating a circuit, containing only CNOT gates, that computes exclusive-ORs of all nonempty subsets of a set of existing qubits. From such a circuit, an output stage for a desired symmetric function, like the ones shown in Figures 4.14 and 4.15, can then be created by inserting appropriate controlled-root-of-NOT gates between the CNOT gates. The target gates for these controlled gates are determined as follows: since each controlled-root-of-NOT gate in Figures 4.14 and 4.15 originates from a subcircuit of the form shown in Figure 4.6, the target gate of each such gate must have exponent equal to -2 times the corresponding Walsh-Hadamard spectral coefficient.

In addition to controlled-root-of-NOT gates, the output stage may contain an initial uncontrolled root-of-NOT gate for each output, as seen in Figure 4.14. This uncontrolled root-of-NOT gate derives from combining all of the uncontrolled root-of-NOT gates originating

from Figure 4.6, as shown by the simplification of Figure 4.13 to Figure 4.14. The exponent of the initial uncontrolled root-of-NOT gate for a given output is therefore obtained as the sum of all Walsh-Hadamard spectral coefficients for that output.

Based on the above discussion, Algorithm 3 creates an output stage for an m-output symmetric function when given Walsh-Hadamard spectral coefficients for each output of the function. This algorithm has converted the recursion from the proof of Theorem 13 into a loop with the loop variable q running from 2 to k. The qubits u_0 through u_{k-1} in Algorithm 3 are used to feed the DIPS $\mathrm{S}_n^{\mathrm{DIP}(0,0)}$ through $\mathrm{S}_n^{\mathrm{DIP}(k-1,0)}$, respectively, into the output stage.

Algorithm 4 now performs the task of combining a DIPS-initialization stage and output stage to create a circuit that realizes a desired single- or multi-output symmetric function. The DIPS-initialization stage is created using Algorithm 1 from Chapter 3 as a subroutine, while the output stage is created using Algorithm 3. Algorithm 4 also creates a mirror circuit Q_{mirror} to restore the input qubits x_1 through x_n and all ancillary qubits to their original states. If the mirror circuit is not needed, the steps involving Q_{mirror} can simply be skipped.

The run-time complexities of Algorithm 3 and 4 are easily analyzed. First, considering Algorithm 3, it is known [41] that a j-bit Gray code can be generated in $\mathcal{O}(2^j)$ time, *i.e.*, requiring only $\mathcal{O}(1)$ time to output each code word. For a given k, Algorithm 3 generates Gray codes of 1 through $k-1$ bits, so the total time required to generate these codes is $\mathcal{O}\left(\sum_{j=0}^{k-1} 2^j\right) = \mathcal{O}(2^k)$. The algorithm given in [41] also allows the bit-position differing between each pair of consecutive code words (d in Algorithm 3) to be obtained in $\mathcal{O}(1)$ time per code word. Therefore, once all the necessary Gray codes have been generated, Algorithm 3 generates each CNOT gate of its circuit in $\mathcal{O}(1)$ time. From Theorem 13, the total number of CNOT gates is $2^k - 2$, so generation of the CNOT gates requires $\mathcal{O}(2^k)$ time.

Remaining contributions to the run time of Algorithm 3 are the calculation of the β_j

Algorithm 3: Create an output stage for a symmetric function.

Function `CreateOutputStage(`k, m, α`)`

Input: Integers $k, m \geq 1$ together with a set of integers $\alpha = \{\alpha_{i,j}\}$ with $0 \leq i \leq 2^k - 1, 1 \leq j \leq m$, where $\alpha_{0,j}$ through $\alpha_{2^{k-1},j}$ represent the Walsh-Hadamard spectral coefficients of the indicator function of the j-th output of a Boolean symmetric function.

Output: A quantum circuit that acts as an output stage to realize the symmetric function.

External subroutines: `GrayCode(`j`)`, which generates the code words G_0 through G_{2^j-1} of a j-bit cyclic Gray code, beginning with $G_0 = 00 \ldots 0$.

Let Q be a quantum circuit, initially containing no gates, that acts on qubits u_0 through u_{k-1} and y_1 through y_m.

for *j from 1 to m* **do**

Let $\beta_j = \sum_{i=0}^{2^k-1} \alpha_{i,j}$.

Append an $X_k^{\beta_j}$ gate, acting on y_j, to Q.

Append a controlled-$X_k^{-2\alpha_{1,j}}$ gate, with control u_0 and target y_j, to Q.

end

for *q from 2 to k* **do**

Let $(G_0, G_1, \ldots, G_{2^{q-1}-1})$ = `GrayCode(`$q-1$`)`.

Let $G_{2^{q-1}} = 00 \ldots 0$.

for *p from 1 to* 2^{q-1} **do**

Let d be the position of the unique bit that differs between G_{p-1} and G_p, where the least significant bit has position 0.

Append a controlled-NOT gate, with control u_d and target u_{q-1}, to Q.

Let i be the integer whose base-two representation is obtained by adding an initial bit 1 to G_p.

for *j from 1 to m* **do**

Append a controlled-$X_k^{-2\alpha_{i,j}}$ gate, with control u_{q-1} and target y_j, to Q.

end

end

Output the circuit Q.

end

end

and the generation of controlled-root-of-NOT gates. Each β_j is a sum of 2^k integers, and j runs from 1 to m, so calculation of all β_j requires $\mathcal{O}(2^k m)$ time. Similarly, each controlled-root-of-NOT gate requires $\mathcal{O}(1)$ time to generate, and there are $\mathcal{O}(2^k)$ of them per output, so generation of all such gates also requires $\mathcal{O}(2^k m)$ time. Since $\mathcal{O}(2^k m)$ dominates all of the other contributions to the run time of Algorithm 3, we see that the total run time of Algorithm 3 is $\mathcal{O}(2^k m)$ as well.

Algorithm 4: Realize a Boolean symmetric function with an arbitrary number of inputs and outputs.

Input: An n-input, m-output symmetric Boolean function, specified as a collection of bits $a_{w,j}$ for $0 \leq w \leq n$ and $1 \leq j \leq m$, where the j-th output of the function has the indicator function $I_j : \mathbb{N}_{\leq n} \to \{0, 1\}$ defined by $I_j(w) = a_{w,j}$.

Output: A quantum circuit, operating on qubits x_1 through x_n and y_1 through y_m together with a number of ancillary qubits, that realizes the given function in the following manner: for $1 \leq j \leq m$, the circuit inverts qubit y_j when the j-th output of the function is 1.

External subroutines: `WHTransform(`A, k`)`, which computes the size-2^k Walsh-Hadamard transform of input data expressed as an array of real numbers A and outputs the spectral coefficients as another array of size 2^k.

Let $k = \lceil \log_2(n + 1) \rceil$.
Let the quantum circuit $Q_{\text{init}} =$ `RealizeDips(`$n, k - 1, 0$`)`.
Relabel the qubits y_1 through y_{k-1} of Q_{init} as u_1 through u_{k-1}.
Designate u_0 as an alias for the qubit x_n of Q_{init}.
Let Q_{mirror} be the inverse of Q_{init}.
Let $\tilde{n} = 2^k$.
for *j from 1 to m* **do**
    Let $A_j = [a_{0,j}, a_{1,j}, \ldots, a_{n,j}]$.
    Enlarge A_j to size $\tilde{n}$ by padding with zeroes on the right.
    Let $[\alpha_{0,j}, \alpha_{1,j}, \ldots, \alpha_{\tilde{n}-1,j}] =$ `WHTransform(`A, k`)`.
end
Let $\alpha = \{\alpha_{i,j}\}$ with $0 \leq i \leq \tilde{n} - 1$ and $1 \leq j \leq m$.
Let $Q_{\text{out}} =$ `CreateOutputStage(`k, m, α`)`.
Let Q be the concatenation of Q_{init}, Q_{out}, and Q_{mirror}, in that order.
Output the circuit Q.

For Algorithm 4, the Walsh-Hadamard transform of a single input vector can be computed in $\mathcal{O}(n \log n)$ time [42], and for an m-output function, m such transforms must be computed. Generation of the output stage is performed by Algorithm 3, which runs in $\mathcal{O}(2^k m)$ time. Since $k = \mathcal{O}(\log n)$, the output stage requires $\mathcal{O}(mn)$ time to generate. Meanwhile, the DIPS-initialization stage of the circuit, which is generated using Algorithm 1 from Chapter 3, requires $\mathcal{O}(kn+k^2) = \mathcal{O}\big(n \log n+(\log n)^2\big)$ time, hence $\mathcal{O}(n \log n)$ since $(\log n)^2$ is dominated by $n \log n$. The mirror circuit is just the inverse of the DIPS-initialization stage, so it also requires $\mathcal{O}(n \log n)$ time to generate. We see that the time required to compute the Walsh-Hadamard transforms, $\mathcal{O}(mn \log n)$, dominates all other contributions to the run time of Algorithm 4, and therefore gives the run-time complexity of the whole algorithm.

4.3 Realization of non-symmetric Boolean functions using repeated variables

So far, in this chapter as well as in Chapters 2 and 3, I have considered only the realization of totally symmetric Boolean functions. However, the circuit structures I introduced for this purpose can also be used for realizing arbitrary non-symmetric functions through *repetition of variables*. Given a symmetric function of n variables, $f(x_1, x_2, \ldots, x_n)$, if one lets some of x_1 through x_n in fact be repetitions of the same variable, then one obtains another function of less than n variables that is not necessarily symmetric. In this section I show that the circuit structures I previously introduced for the realization of symmetric functions work especially well with this repetition of variables—in particular, variables can be repeated with *no increase in quantum cost at all*.

4.3.1 Repetition of variables

To give a concrete example of a repeated variable used to derive a non-symmetric Boolean function from a symmetric one, consider the function $S^{2,3,4}(x_1, x_2, x_3, x_4)$. A Karnaugh map for this function is shown in Figure 4.16a. Suppose that we let x_1 and x_2 in fact be the same variable, call it a, while x_3 and x_4 are distinct variables b and c respectively. The identification of x_1 and x_2 with a single variable a is reflected in Figure 4.16a by the shaded rows of the Karnaugh map: the shaded rows represent combinations where $x_1 \neq x_2$, impossible when $x_1 = x_2 = a$. Removing the shaded rows and relabeling the resulting three-variable Karnaugh map with a, b, and c, as shown in Figure 4.16b, gives the new function $S^{2,3,4}(a, a, b, c) = a \vee (b \wedge c)$, which is a non-symmetric function of three variables. We therefore see that repetition of the variable a has created a three-input non-symmetric function from a four-input symmetric function.

x_1x_2 \ x_3x_4	00	01	11	10
00	0	0	1	0
01	0	1	1	1
11	1	1	1	1
10	0	1	1	1

(a) A Karnaugh map for the symmetric function $S^{2,3,4}(x_1, x_2, x_3, x_4)$.

a \ bc	00	01	11	10
0	0	0	1	0
1	1	1	1	1

(b) The function created by repeating a variable: $x_1 = x_2 = a$, $x_3 = b$, and $x_4 = c$.

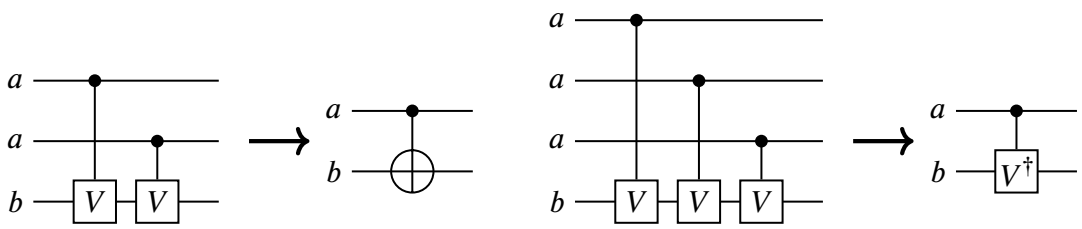
Figure 4.16: Karnaugh maps for a symmetric function with a repeated variable.

The circuit structures introduced in this and the previous chapters can readily implement repeated variables with no increase, and in some cases even a decrease, in quantum cost. To see this, observe that gates directly involving the input qubits fall into one of two types: they are either part of a cascade of controlled-U gates, where U is some root-of-NOT gate, in

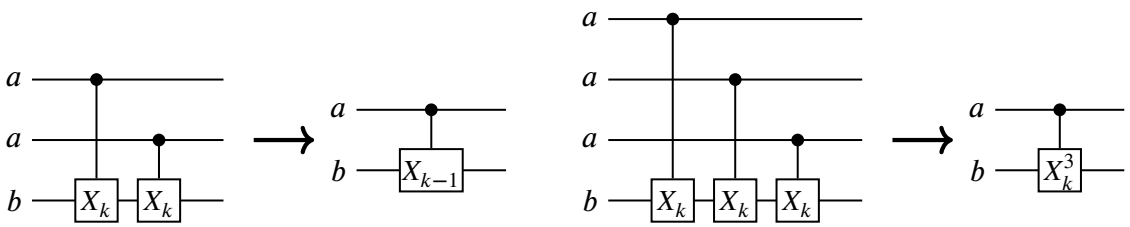
which each controlled-gate targets the same qubit, or they are part of a cascade of CNOT gates that is used to compute the exclusive-OR of the inputs. In both cases, when a variable is repeated, the involved gates can be simplified in such a manner as to emulate the effect of a repeated variable without actually using additional qubits to hold copies of that variable. For instance, Figure 4.17a shows how, when the variable a is repeated twice, the two controlled-V gates both controlled by a collapse into a single CNOT gate, while three repetitions result in a controlled-$V^\dagger$ gate. More generally, Figure 4.17b shows how any two controlled-X_k gates with the same control variable and same target qubit collapse into a single controlled-X_{k-1} gate, and any three such gates collapse into a controlled-X_{k-1}^3 gate. Similar simplifications can be made if a variable is repeated four or more times. For instance, if the variable a is repeated m times, then a cascade of m controlled-X_k gates, all controlled by a, collapses to a controlled-X_k^m gate.

Cascades of CNOT gates simplify slightly differently: in a cascade of CNOT gates, if a variable is repeated an odd number of times, then it is included in the cascade and otherwise excluded. This follows from the fact that the exclusive-OR of a variable a with itself an odd number of times is simply a, while the exclusive-OR of a with itself an even number of times is 0. Examples of the simplification of CNOT cascades for a variable a repeated two or three times are shown in Figure 4.17c. In any case, no more than one instance of the variable is needed. From these examples it is clear that, no matter how many times a variable is repeated, that variable's involvement in any cascade of gates can be reduced to either one gate or no gates at all.

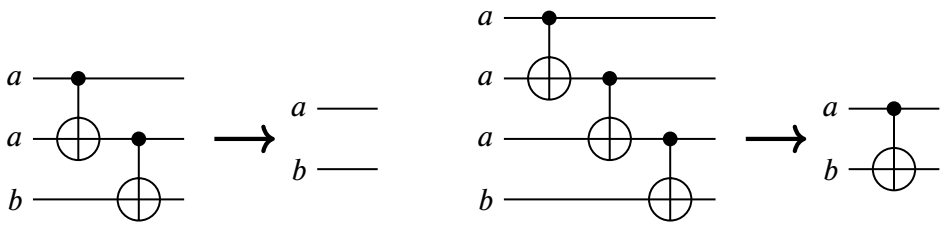
We may now consider an example of how these simplifications apply within an actual circuit that realizes a symmetric function, to be made non-symmetric through a repeated variable. Figure 4.18a shows the realization of the symmetric function $\mathrm{S}^{2,3,4}(x_1, x_2, x_3, x_4)$



(a) Simplification of two or three controlled-V gates when controlled by the same variable.



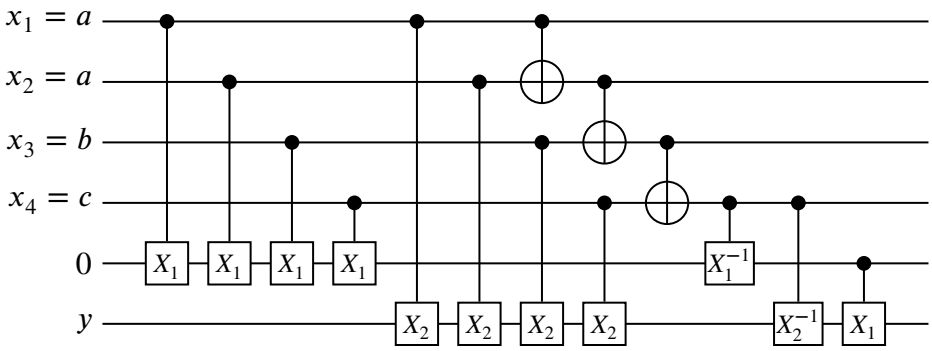
(b) Simplification of two or three controlled-X_k gates when controlled by the same variable.



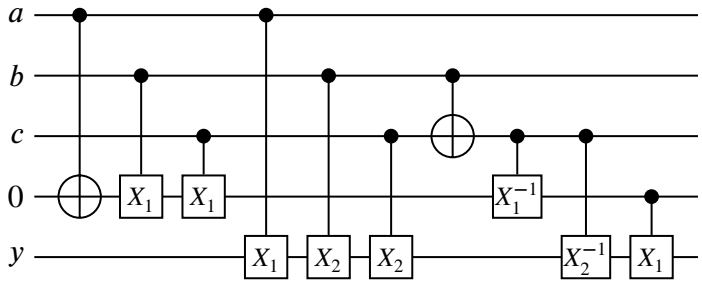
(c) Simplification of cascades of CNOT gates containing a repeated variable.

Figure 4.17: Simplifications of gate cascades that allow a variable to be repeated without using additional qubits.

according to the method presented in Chapter 3. When $x_1 = x_2 = a$, the pairs of adjacent X_1 $(= V)$ and X_2 gates that are both controlled by a can be simplified as in Figure 4.17, giving a CNOT and controlled-X_1 gate respectively. The cascade of CNOT gates is also simplified, by excluding a entirely since it is repeated an even number of times. This results in the circuit shown in Figure 4.18b. In this case, we can see that the final resulting circuit has a quantum cost that is no greater than the original—in fact, the quantum cost is even less than that required to realize an order-2 DIPS of three variables.



(a) Realization of the four-variable symmetric function $S^{2,3,4}(x_1, x_2, x_3, x_4)$.



(b) Result of simplifying gate cascades with repeated variables as in Figure 4.17.

Figure 4.18: Demonstration of simplification of the resulting circuit following repetition of variables in a symmetric function.

Variables can also be repeated in non-DIPS symmetric functions that require the use of the Walsh-Hadamard transform to realize. In this case, the output stage of the circuit is

completely unaffected because it does not involve any of the inputs to the symmetric function directly, instead only working with DIPS that are produced by the DIPS-initialization stage. The DIPS-initialization stage then simplifies as previously described and demonstrated in Figures 4.18.

There remains the issue of how an arbitrary Boolean function can be expressed as a symmetric function with repeated variables. This question has received significant attention in the past [43, 44, 45, 46, 47], and in particular, experimental results from Chrzanowska-Jeske *et al.* [48] show that each variable on average only needs to be repeated about twice for several standard benchmark functions. Therefore, when such a symmetrization procedure is combined with my method for realizing symmetric functions, any arbitrary Boolean function can be realized. In fact, the symmetrization algorithm presented in Chrzanowska-Jeske *et al.* also handles multiple-output symmetric functions and symmetrizes them using the same number of repetitions of each variable for all outputs. This result allows my method to be applied to arbitrary multiple-output Boolean functions as well: as long as each variable is repeated the same number of times across all outputs, such a function can be realized in the same manner as a multiple-output symmetric function.

4.3.2 Quantum cost impact of repeating variables

At the beginning of this section, and in the introduction to this chapter, I claimed that repetition of variables can be done with no increase in quantum cost at all. In the example given in the previous subsection, we saw that the resulting circuit for a three-input non-symmetric function, generated by repeating a variable in a four-input order-2 DIPS, in fact has a lower quantum cost than a circuit for a three-input order-2 DIPS.

In the general case, suppose that we have an order-k DIPS of n' variables, $S_{n'}^{\mathrm{DIP}(k,s)}$, some

of which are repeated. Suppose that the number of distinct variables is n. For the example given in Figure 4.18, $n' = 4$ and $n = 3$. To realize this DIPS, we begin with an appropriate instance of the circuit structure from Figure 3.10 and simplify the gate cascades at the start of the circuit (represented by the "$\sum$" and "$\bigoplus$" blocks in Figure 3.10). After simplification, each input variable can act as a control input to at most one gate per cascade. In other words, the number of gates per cascade is no more than n, and the number of gates in the cascade of CNOT gate is no more than $n - 1$. But realizing an n-input DIPS using Figure 3.10 also requires n gates for each of the X_i cascades and $n - 1$ gates for the cascade of CNOT gates. Therefore, the first part of the circuit for $\mathrm{S}_{n'}^{\mathrm{DIP}(k,s)}$, containing the gate cascades, is no more expensive than the first part of the circuit for $\mathrm{S}_{n}^{\mathrm{DIP}(k,s)}$. The second parts of both circuits, containing the controlled-X_i^j gates in Figure 3.10 have identical gate counts because the size of this part of the circuit depends only on k and not on the number of inputs. As a result, after simplification according to the scheme shown in Figure 4.17, the circuit for $\mathrm{S}_{n'}^{\mathrm{DIP}(k,s)}$ contains no more two-qubit controlled gates than the circuit for $\mathrm{S}_{n}^{\mathrm{DIP}(k,s)}$.

If the function to be realized is not symmetric, the above result is unaffected since, as previously noted, the DIPS-initialization stage of the resulting circuit simplifies in the same way while the output stage does not directly involve the inputs at all, and so is insensitive to variable repetitions.

4.3.3 Algorithm to realize symmetric functions with repeated variables

Based on the discussion in Section 4.3.1 and the simplifications illustrated in Figure 4.17, I introduce Algorithm 5, which generates circuits to realize DIPS with repeated variables. This algorithm is a slight modification of Algorithm 1 from Chapter 3. Specifically, the algorithm takes an additional input R, which is an array of integers $[r_1, r_2, \ldots, r_n]$ that specifies how

many times each variable is repeated.

One particular aspect of Algorithm 5 requires further elaboration: the requirement that at least one of the r_i is odd. This requirement is needed to ensure that the loop of i over the range 1 to $n-1$, which generates a cascade of CNOT gates that computes the exclusive-OR of those variables with an odd number of repetitions, works properly. This requirement does not lead to any loss of generality because, as the following reasoning shows, any symmetric function with repeated variables can be expressed in a form where at least one variable is repeated an odd number of times. Suppose that we have a symmetric function where each variable is repeated an even number of times. Then the input weight is always even as well, so odd elements of the function's ON-set have no relevance. We can therefore simplify this function by discarding all odd elements of the ON-set, dividing the remaining elements by 2, and also dividing the number of repetitions by 2 for each variable. If the new resulting function still has an even number of repetitions for each variable, this process can be repeated until at least one variable has an odd number of repetitions. For instance, given the function $\mathrm{S}^{3,4,6,7}(a, a, b, b, c, c, c, c)$, we can discard 3 and 7 from the ON-set, giving $\mathrm{S}^{4,6}(a, a, b, b, c, c, c, c)$. Dividing both the elements of the ON-set and the variable repetitions by 2 then produces the equivalent function $\mathrm{S}^{2,3}(a, b, c, c)$.

Algorithm 4 can be adapted to handle repeated variables as well by replacing the call to `RealizeDips(`$n, k-1, 0$`)` with a call to `RealizeDipsWithRepeatedVars(`$n, k-1, 0, R$`)`, where R defines the repetition count of each variable as in Algorithm 5. There is one small caveat: if the function to be realized has n unique input variables but n' variables when counting all repetitions, then this n is the same as the n in Algorithm 5, but the k in both Algorithms 4 and 5 is calculated based on n', *i.e.*, $k = \lceil \log_2(n' + 1) \rceil$. (This same caveat is encountered when analyzing quantum cost, as discussed in Section 4.4.) Combining this

Algorithm 5: Realize a set of DIPS with repeated variables.

Function `RealizeDipsWithRepeatedVars(`n, k, s, R`)`

Input: A positive integer n, nonnegative integer k, nonnegative integer s with $s < 2^k$, and a length-n array of positive integers $R = [r_1, r_2, \ldots, r_n]$, where at least one of the r_i is odd.

Output: A circuit that realizes $\mathrm{S}^{\mathrm{DIP}(k, s \bmod 2^j)}({x_1}^{r_1}, \ldots, {x_n}^{r_n})$ for $0 \leq j \leq k$, where the notation ${x_i}^{r_i}$ indicates that the variable x_i is repeated r_i times.

Let Q be a quantum circuit, initially containing no gates, that operates on qubits x_1 through x_n and y_1 through y_k.

for *j from k to* 1 **do**
  for *i from 1 to n* **do**
    Append a controlled-$X_j^{r_i}$ gate with control x_i and target y_j to Q.
  end
end

Let $i' = 0$.

for *i from 1 to* $n - 1$ **do**
  if r_i *is odd* **then**
    if $i' \neq 0$ **then**
      Append a controlled-NOT gate with control $x_{i'}$ and target x_i to Q.
    end
    Let $i' = i$.
  end
end

Designate y_0 as an alias for the qubit $x_{i'}$.

for *j from 1 to k* **do**
  for j' *from* 0 *to* $j - 1$ **do**
    if $\mathrm{b}_{j'}(s) = 1$ **then**
      Let $U = X_{j-j'}$.
    else
      Let $U = X_{j-j'}^{-1}$.
    end
    Append a controlled-U gate with control $y_{j'}$ and target y_j to Q.
  end
end

Output the circuit Q.

end

modified Algorithm 4 with an algorithm for symmetrizing Boolean functions, such as that described in [48], gives an algorithm for realizing any Boolean function using controlled-root-of-NOT gates.

Like Algorithm 1 from Chapter 3, Algorithm 5 generates quantum gates in a straightforward way without performing any searching or backtracking, so its run-time complexity is the same as the number of gates in the circuit that it generates, namely $\mathcal{O}(kn + k^2)$. If Algorithm 4 is modified to handle repeated variables, as described in the previous paragraph, then its run-time complexity can be analyzed as in Section 4.2.4. As in that section, the time required to compute the Walsh-Hadamard transforms is found to dominate the run time of the algorithm, so it runs in $\mathcal{O}(mn' \log n')$ time, where, as in the previous paragraph, n' is the number of variables of the function to be realized, including all repetitions.

4.4 Calculation and comparison of circuit complexity

We can now derive an upper bound on the quantum cost required to realize any single-output or multi-output symmetric function. To realize a symmetric function of n input variables using the Walsh-Hadamard transform-based method of Section 4.2, a transform of size 2^k such that $n \leq 2^k - 1$ is required. This means that $k = \lceil \log_2(n+1) \rceil$. Now, recall that the circuit used to realize an arbitrary symmetric function consists of a DIPS-initialization stage followed by an output stage, where the DIPS-initialization stage is an instance of the circuit structure from Figure 3.10 and the output stage is of the form shown in Figures 4.14 and 4.15. For a Walsh-Hadamard transform of size 2^k, DIPS of orders 0 through $k-1$ are required, so from the formula given in (3.23), we get a cost of

$$k\left(n + \frac{k-1}{2}\right) - 1 \tag{4.20}$$

for the DIPS-initialization stage. From Theorem 13, the output stage requires up to $2^k - 2$ CNOT gates to compute all possible exclusive-ORs of subsets of DIPS, and it also requires one gate per output for each nonzero Walsh-Hadamard spectral coefficient, so the total cost of the output stage can be up to

$$2^k(m+1) - 2 \tag{4.21}$$

for an m-output function. Adding (4.21) to (4.20) results in a total cost of

$$k\left(n + \frac{k-1}{2}\right) + 2^k(m+1) - 3 \tag{4.22}$$

for an n-input, m-output symmetric function. The space complexity, as measured by the number of ancillary qubits, is simply $k - 1$ because one ancillary qubit is required to hold each of the DIPS from order 1 to $k - 1$. If these ancillary qubits are additionally to be restored to their original state of $|0\rangle$, then a mirror circuit must be used. The mirror circuit is simply the inverse of the DIPS-initialization stage; the output stage does not require a mirror circuit since, when designed using a pattern of CNOT gates obtained from Theorem 13, it leaves the ancillary qubits in the same states in which they started. Therefore, when a mirror circuit is included, the cost of (4.20) must be counted twice, which brings the total cost up to

$$k(2n + k - 1) + 2^k(m+1) - 3. \tag{4.23}$$

We can make the following observations with respect to the growth rate of the costs given in both (4.22) and (4.23):

- Since $k = \mathcal{O}(\log n)$, the total cost of realizing an n-input symmetric function grows as $\mathcal{O}(n \log n)$.

- For a fixed value of n and therefore a fixed k, the cost per output is constant, regardless of the number of outputs.

- As n increases, k only increases by 1 every time n reaches a power of two, so the cost per output in fact stays constant over wide ranges of n.

For non-symmetric functions realized through repetition of inputs in a symmetric function, the result from 4.3.2 can be applied to get an upper bound for quantum cost expressed in terms of the numbers of inputs when repetitions are and are not counted. If n is the number of unique input variables and n' is the number of inputs including repetitions, then a Walsh-Hadamard transform of size 2^k where $k = \lceil \log_2(n' + 1) \rceil$ is required. In other words, the size of the required transform depends on the number of inputs including repetitions, since that is the actual number of inputs to the symmetric function being realized. The maximum cost of the output stage in turn depends on k via (4.21). However, the cost of the DIPS-initialization stage depends on n, not n', because the simplifications shown in Figure 4.17 ensure that repetitions of inputs do not increase the total cost—this was the essence of the result from Section 4.3.2. Therefore, the maximum quantum cost of realizing an arbitrary non-symmetric Boolean function is also given by (4.22) or (4.23), depending on whether a mirror circuit is used, but with the caveat that $k = \lceil \log_2(n' + 1) \rceil$ and not $\lceil \log_2(n + 1) \rceil$. Based on the experimental findings of Chrzanowska-Jeske *et al.* [48], it appears that n' is often in the neighborhood of $2n$.

Table 4.1 presents some sample values obtained for various values of n and m in (4.22) and (4.23). These upper bounds can be compared to the ones obtained by Maslov [33] with a realization method that only uses NOT, CNOT, and Toffoli gates. Whereas Maslov's upper bound [33, Theorem 4] on quantum cost for an n-input, m-output symmetric function grows at least as fast as $n^2 + nm \log n$, the bounds (4.22) and (4.23) contain no nm term

Table 4.1: Upper bounds on quantum cost for symmetric functions realized using the Walsh-Hadamard transform-based method presented in this chapter.

n	k	Cost of DIPS-init. stage = (4.20)	Cost per add'l output $= 2^k$	Total cost for num. outputs without mirror			with mirror		
				1	2	3	1	2	3
5	3	17	8	31	39	47	48	56	64
10	4	45	16	75	91	107	120	136	152
15	4	65	16	95	111	127	160	176	192
20	5	109	32	171	203	235	280	312	344
25	5	134	32	196	228	260	330	362	394
30	5	159	32	221	253	285	380	412	444
35	6	224	64	350	414	478	574	638	702

and only increase as $\mathcal{O}(n \log n)$ when n is increased with m fixed. In addition, Maslov's method uses up to $n + 2^{\lfloor \log n \rfloor} - 1$ ancillary qubits[1] while the method presented here uses only $k - 1 = \lceil \log_2(n+1) \rceil - 1$. The costs given in Table 4.1 also compare favorably to the costs obtained by Maslov for some actual benchmark functions [33, Table 2]. For instance, the 15-input, single-output function "sym15" is realized with a cost of 569 while the upper bound of 95 from Table 4.1 is nearly six times smaller. Even for 15 inputs and 3 outputs with a mirror circuit, my upper bound of 192 is still about a third of Maslov's cost for the 15-input, single-output function. For the 35-input, single-output function "dbruijn_5", Maslov reports a cost of 2990, which is more than eight times higher than my upper bound of 350 for such a function. These findings therefore support the idea that realizing functions directly on the level of two-qubit gates can provide significant improvements over realization methods that use only Toffoli gates together with concepts from Boolean algebra.

[1] Maslov calls these "garbage bits" because they are not restored to their original states using a mirror circuit. Hence, I use the upper bound given by (4.22), which assumes no mirror circuit, for comparison.

4.5 Conclusion

In this chapter, I accomplished two objectives that greatly increase the power of the realization methods presented in Chapters 2 and 3. In the first part of the chapter, I showed that there is a close correspondence between DIPS and the basis functions of the Walsh-Hadamard transform. I showed that by taking advantage of this correspondence, the indicator function of an arbitrary symmetric function can be broken down into a combination of Walsh-Hadamard basis functions, which in turn allows the arbitrary symmetric function to be realized using a combination of DIPS. The ability to realize DIPS of multiple orders simultaneously, introduced at the end of Chapter 3, is especially useful for this purpose as the DIPS obtained in that way can conveniently be combined using exclusive-OR operations (implemented by CNOT gates) to create the corresponding symmetric functions for every Walsh-Hadamard basis function without any additional ancillary qubits.

I also considered the realization of multiple-output symmetric functions. This appears to be the first time that the realization of multiple-output symmetric functions directly using non-permutative two-qubit gates has been studied. For multiple-output symmetric functions, the newly introduced Walsh-Hadamard transform-based method again performs very well because it allows each additional output to be realized with an increase in quantum cost of at most twice the number of inputs. This upper bound for the quantum cost applies regardless of the functions being realized. In other words, there are no "pathological" symmetric functions that result in abnormally high quantum costs (compared to most other functions) when realized using my method. I calculated an upper bound for the cost of realizing any symmetric Boolean function, and this upper bound was found to compare favorably with the bound obtained by a method of Maslov that only uses Toffoli gates.

In the second part of the chapter, I further increased the power of the presented realization

method by showing that it can also be applied to non-symmetric functions through repetition of variables. Every Boolean function can be symmetrized, that is, it can be expressed as a symmetric function with repeated variables. Previous work has shown that in practice, this can often be done using only two or three repetitions of each variable, making it a viable method of realizing arbitrary Boolean functions if low-cost realizations of symmetric functions are available. My method provides these low-cost realizations of symmetric functions and in fact is able to repeat variables without any increase in quantum cost of the resulting circuit at all. In fact, multiple-output Boolean functions can also be symmetrized using the same number of repetitions for each variable, which consequently allows my method to apply to multiple-output Boolean functions as well. I have therefore now demonstrated a realization method for arbitrary Boolean functions directly using two-qubit controlled gates.

Chapter 5

Cycle-based synthesis of binary permutative quantum circuits

The ability to synthesize permutative circuits is of paramount importance in realizing a viable quantum computing system. Realizing permutative functions using quantum circuits is analogous to constructing Boolean functions using AND, OR, and NOT gates in classical digital logic; it is a crucial step in designing and constructing circuits for use in quantum algorithms, such as oracles for Grover's algorithm.

In Chapters 2 through 4, I demonstrated a realization method for both single- and multiple-output functions, which may or may not be reversible. Here, I consider the different scenario of realizing a function that is specified as a permutation and is therefore inherently reversible. Such a function could simply be treated as a multiple-output function that happens to have the same number of inputs and outputs, and could therefore be realized using the methods presented in the previous chapters as well. However, doing so entails essentially treating the function as just a collection of single-output functions, and therefore requires allocating an ancillary qubit for each output. In contrast, the realization method I present here realizes permutative functions "in place", that is, using the same qubits for both the inputs and outputs of the function. This requires treating the function as a permutation and not as a collection of single-output functions, and therefore involves a wholly different set of concepts to those presented in Chapters 2 through 4.

Previously proposed approaches to the synthesis of binary reversible and quantum circuits are of a variety of different types. One notable class of synthesis methods are transformation-based methods, which operate on the truth table of a reversible function and attempt to transform the truth table in such a way as to produce the identity map. The sequence of transformations performed is then mapped in some way to a reversible circuit. The well-known MMD method [49] and its numerous variants are examples of transformation based-methods.

Another class of synthesis methods are cycle-based methods, which first express a permutation as a set of cycles and then realize individual cycles or small sets of cycles at a time. These methods often use a predetermined set of subcircuits, each designed to realize a specific type of cycle or set of cycles. Examples of cycle-based methods include those of Shende *et al.* [50] and Saeedi *et al.* [51].

Yet other synthesis methods for permutative functions include positive-polarity Reed-Muller (PPRM)- and search-based methods such as that proposed by Gupta *et al.* [52], and binary decision diagram (BDD)-based methods such as the one by Wille and Drechsler [53]. For a more thorough discussion of these and other methods, I refer the reader to [54].

Although, as can be seen from the preceding examples, there exists diversity of proposed methods for reversible circuit synthesis, they generally assume that the circuit to be synthesized is binary. Much less work has been done on the synthesis of quantum circuits involving multiple-valued qudits. In particular, there is little to no published work on synthesis algorithms for binary functions that can also be generalized to a multiple-valued setting. To address this gap, in this chapter I present a systematic method for synthesizing binary quantum circuits to realize arbitrary permutative functions. I introduce a new type of reversible gate, the *distance gate*, which is defined to realize a permutation consisting

of exactly one transposition. Distance gates generalize in a straightforward manner to the case of multiple-valued quantum circuits, and they can even be used in mixed circuits that contain both binary and multiple-valued elements. I examine multiple-valued distance gates in detail in Chapter 6.

5.1 Representation of a permutation in terms of cycles

Throughout this chapter, I assume that a reversible function with n Boolean inputs and outputs, where n is any positive integer, is to be realized using a quantum circuit. Such a function can be given in the form of a truth table, but in order to apply a cycle-based synthesis algorithm, it is necessary to first express the function in terms of cycles. A *cycle* is a specific type of permutation that can be expressed as a sequence of items, in which each item in the sequence maps to the next and the last item maps to the first, forming a closed cycle (hence the name). A cycle containing n elements is written as $(x_1\ x_2\ x_3 \ldots x_n)$, so that x_1 maps to x_2, x_2 to x_3, and so on. Because x_n also maps to x_1, such a cycle can be written in n different ways, all of which are equivalent: $(x_1\ x_2\ x_3 \ldots x_n)$ denotes the same cycle as $(x_2\ x_3 \ldots x_n\ x_1)$, $(x_3 \ldots x_n\ x_1\ x_2)$, and so on all the way to $(x_n\ x_1\ x_2\ x_3 \ldots)$. A cycle consisting of only one element—e.g., (x)—is called a *singleton* and represents a trivial permutation. A cycle consisting of two elements, $(x_1\ x_2)$, is called a *transposition* and represents a permutation that swaps x_1 and x_2.

Table 5.1 gives an example of a three-input, three-output reversible function in the form of a truth table. Here, I first review how a permutation can be expressed as a sequence of cycles, using this function as an example. It is convenient for the sake of discussion to interpret the bit-sequences in Table 5.1 as base-two representations of integers. This interpretation is only for convenience and is unimportant to the operation of a cycle-based

Table 5.1: A reversible Boolean function to be expressed in terms of cycles.

Input bits			Output bits		
a	b	c	a'	b'	c'
0	0	0	0	0	1
0	0	1	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	0	1	1

synthesis method. In order to express the permutation represented by the truth table as a sequence of cycles, we consider a possible input value and trace out the cycle produced starting from that value. First, beginning with an input value of 0 (corresponding to the input bit-sequence 000), Table 5.1 shows that the output value 1 (corresponding to the bit-sequence 001) is generated. Similarly, with the input value 1, the output value 4 (corresponding to 100) is generated. With the input value 4, the output value 0 is generated, which completes the cycle. Hence, the resulting cycle is (0 1 4).

Other cycles are obtained in a similar manner. If we choose 2 as an initial input value, then the cycle (2 6 7 3) is produced, and with 5 as the initial input value the cycle (5) is produced. Therefore, the cycle representation of the function defined by Table 5.1 is (0 1 4)(2 6 7 3)(5). Usually, by convention, singleton cycles (5) may be omitted (since any input value not written in the cycle notation is assumed to map to itself and therefore form a singleton cycle), which results in the cycle representation (0 1 4)(2 6 7 3).

In general, the following algorithm represents a permutation, defined by the function f, as a sequence of cycles.

Algorithm 6: Express a permutation as a list of cycles

Input: A permutative function f with domain and range S.
Output: A list of cycles C.
begin
  Create a list of cycles C, and initialize it to an empty list.
  Create a list L, the *already-visited* list, and initialize it to an empty list.
  while *there exists an element of* S *not in* L **do**
    Let x_0 be an element of S not in L.
    Let $i = 0$.
    while $f(x_i) \neq x_0$ **do**
      Let $x_{i+1} = x_i$.
      Add x_{i+1} to L.
      Increment i by 1.
    end
    Add the cycle $(x_0\ x_1\ \ldots\ x_i)$ to the list C.
  end
  Output C.
end

5.2 Realization of transpositions using distance gates

5.2.1 Uncontrolled distance gates

I now introduce the concept of a distance gate and show how distance gates may be implemented using quantum circuits consisting of controlled-NOT and Toffoli gates.

Definition 14. A quantum gate G that acts on n qubits, x_1 through x_n, is an *uncontrolled distance gate* if there exist $a_i, b_i \in \{0, 1\}$ for $1 \leq i \leq n$ satisfying the following properties:

1. $\neg a_i = b_i$ for all i.

2. The gate G acting on the state $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$ (in other words, the qubit x_i has initial state $|a_i\rangle$) produces the state $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$, and vice versa.

3. The gate G acting on any other basis state leaves that state unchanged.

The bit strings $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$ are called the *active values* of the distance gate.

In other words, a distance gate is a gate that realizes the function expressed by the single transposition $(a_1 a_2 \ldots a_n \;\; b_1 b_2 \ldots b_n)$.

Figure 5.1 shows the structure of circuits that implement uncontrolled distance gates. In this figure, p is an arbitrary index with $1 \leq p \leq n$. The CNOT gates in this structure may either be positive- or negative-control variants. Figure 5.1 represents these allowed variants by showing the control inputs with half-filled dots. However, all CNOT gates must be of the same type. In other words, the CNOT gates may be all positive-control or all negative-control, but a mixture of the two types is not allowed. The central Toffoli gate may have any combination of positive and negative controls, with no constraints. In words, the circuit structure represented by Figure 5.1 may be described as follows:

- A particular qubit x_p is chosen from the qubits participating in the circuit.
- The circuit then begins with a sequence of CNOT gates, all with x_p as their control qubit and targeting each of the other qubits in turn. Every CNOT gate in this sequence has the same control polarity.
- These CNOT gates are followed by a Toffoli gate that targets x_p and has all other qubits in the circuit as control qubits.
- Finally, the circuit ends with another sequence of CNOT gates identical to the first. In Figure 5.1, to highlight the symmetry of the circuit structure, these gates are shown in reverse order compared to the ones at the start of the circuit, but the order is unimportant since each of them targets a different qubit.

Suppose that we have a particular instance of the circuit structure from Figure 5.1. "Particular instance" means that a number of inputs n is chosen and the control polarities

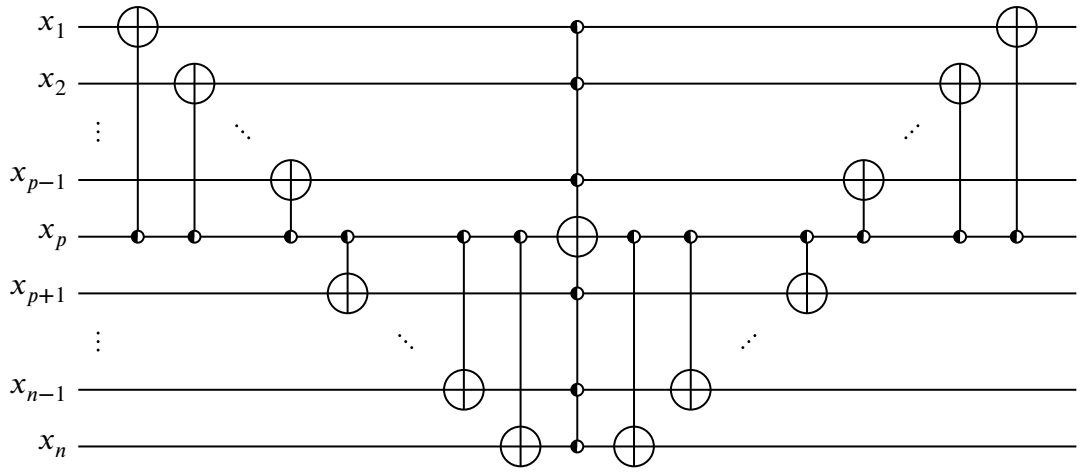


Figure 5.1: A circuit structure that implements uncontrolled distance gates.

(positive or negative) are specified for all gates, so a single quantum circuit is created. This circuit then contains one Toffoli gate and $2(n-1)$ CNOT gates. Define Boolean variables $c_1, c_2, \ldots, c_n$ as follows: for $i \neq p$, $c_i = 1$ if the corresponding control input to the Toffoli gate is of positive polarity and $c_i = 0$ if it is of negative polarity. Define $c_p = 1$ if the CNOT gates are of positive-control type and $c_p = 0$ if they are of negative-control type. Figure 5.2 graphically shows how c_1 through c_n correspond to control polarities in the circuit structure from Figure 5.1: each control input is of positive (resp. negative) polarity if the c_i it is labeled with has a value of 1 (resp. 0).

Given the conditions stated in the previous paragraph, the following proposition then characterizes the behavior of the circuit:

Proposition 15. *Let a quantum circuit be an instance of the circuit structure from Figure 5.1 and let c_1 through c_n be as defined above. Then this quantum circuit implements an uncontrolled distance gate with active values $c_1 c_2 \ldots c_{p-1}\, \overline{c_p}\, c_{p+1} \ldots c_{n-1} c_n$ and $\overline{c_1}\, \overline{c_2} \ldots \overline{c_{p-1}}\, c_p\, \overline{c_{p+1}} \ldots \overline{c_{n-1}}\, \overline{c_n}$, where $\overline{x}$ denotes the logical negation of x, $\overline{x} = \neg x$.*

Proof. We first make the following observation: referring to Figure 5.1, if the central Toffoli

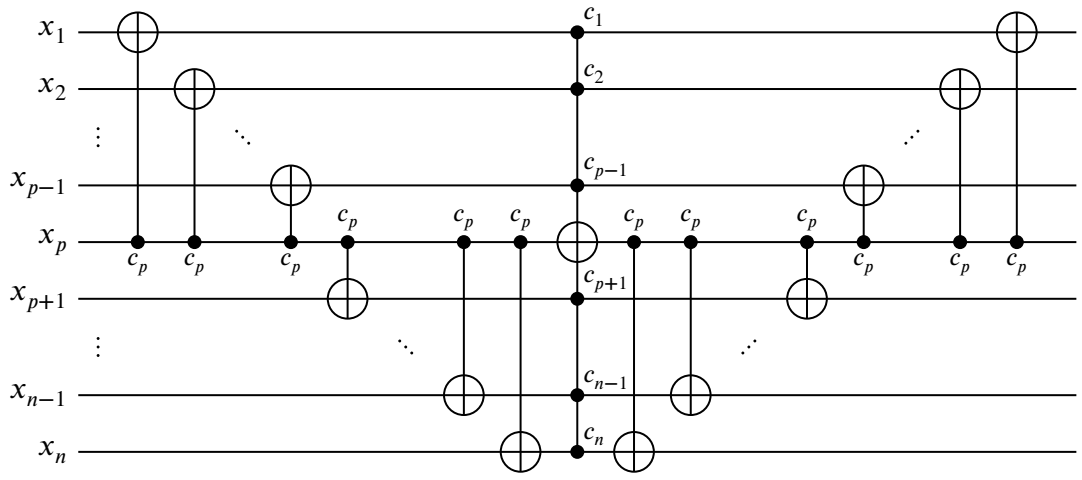


Figure 5.2: The circuit structure from Figure 5.1 with control polarities indicated by c_1 through c_n.

gate in the circuit is not active, then the entire circuit will perform no net change to any of the qubits' states. This fact is clear from the symmetrical arrangement of CNOT gates around the Toffoli gate: since the CNOT gate is self-inverse, when the Toffoli gate is not active, each CNOT gate on the left cancels with the corresponding identical CNOT gate on the right. Therefore, we only need to consider the cases where the Toffoli gate is active.

If c_1 through c_n are as defined above, then the central Toffoli gate is only active when, just prior to this gate, each qubit x_i is in the state $|c_i\rangle$, with the exception of x_p, which may be in any state. We can then work backwards to determine the starting state of the qubits. Qubit x_p only acts as a control qubit to the CNOT gates and never as a target qubit, so its state just prior to the Toffoli gate is the same as its state at the start of the circuit. The CNOT gates on the left side of the circuit are active if x_p begins in the state $|c_p\rangle$ and not active if x_p begins in the state $|\overline{c_p}\rangle$.

First consider the case where x_p begins in the state $|\overline{c_p}\rangle$. None of the CNOT gates on the left side of the circuit are active, so the starting state of each x_i for $i \neq p$ must be $|c_i\rangle$ in order for the Toffoli gate to be active. The Toffoli gate then inverts the state of the qubit x_p,

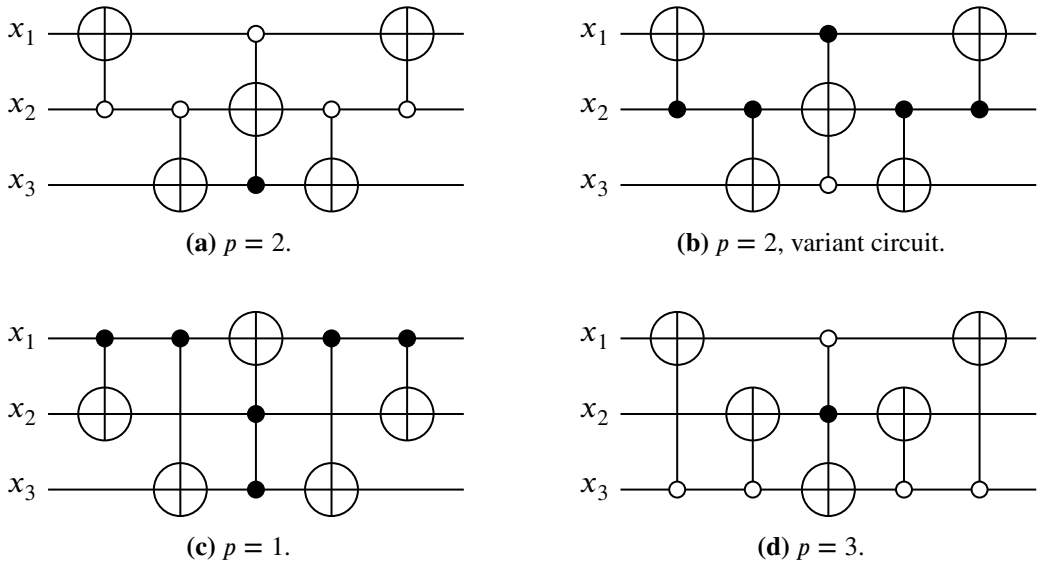
bringing it to $|c_p\rangle$. As a result, the CNOT gates on the right side of the circuit are active, and the states of all of the other qubits are inverted as well, resulting in a final state $|\overline{c_i}\rangle$ for qubit x_i. To summarize, when the qubit x_p begins in the state $|\overline{c_p}\rangle$ and every other qubit x_i begins in the state $|c_i\rangle$, the final states of the qubits after the circuit will be $|c_p\rangle$ for x_p and $|\overline{c_i}\rangle$ for all other x_i.

Now consider the case where x_p begins in the state $|c_p\rangle$. In this case, the CNOT gates on the left side of the circuit are all active. Every qubit x_i, for $i \neq p$, must then begin in the state $|\overline{c_i}\rangle$ in order for the Toffoli gate to be active: the active CNOT gates will invert these qubits so that qubit x_i is in the state $|c_i\rangle$ just prior to the Toffoli gate. The Toffoli gate then inverts the state of qubit x_p, bringing it to $|\overline{c_p}\rangle$. This causes the CNOT gates on the right side of the circuit to be inactive, so no further changes are made to the states of any of the qubits. Therefore, when x_p begins in the state $|c_p\rangle$ and every other qubit x_i begins in the state $|\overline{c_i}\rangle$, the final states of the qubits after the circuit will be $|\overline{c_p}\rangle$ for x_p and $|c_i\rangle$ for all other x_i.

In all other cases besides the above two, the Toffoli gate is not active and therefore the circuit makes no overall change to any qubit's state, as discussed above. The circuit acting on the state $|c_1 c_2 \ldots c_{p-1}\, \overline{c_p}\, c_{p+1} \ldots c_{n-1} c_n\rangle$ produces $|\overline{c_1}\, \overline{c_2} \ldots \overline{c_{p-1}}\, c_p\, \overline{c_{p+1}} \ldots \overline{c_{n-1}}\, \overline{c_n}\rangle$ and vice versa, so it satisfies the definition of a distance gate with those active values. □

Proposition 15 gives a recipe for implementing an uncontrolled distance gate acting on any number of qubits, for any pair of active values. If one wishes to implement an n-qubit uncontrolled distance gate with active values $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$, where $\neg a_i = b_i$ in accordance with Definition 14, one may use an instance of the circuit structure from Figure 5.2 with $c_i = a_i$ for $i \neq p$ and $c_p = b_p$. For instance, consider the task of implementing an uncontrolled distance gate acting on three qubits, with active values $a_1 a_2 a_3 = 011$ and $b_1 b_2 b_3 = 100$. Taking $p = 2$, $c_1 = a_1 = 0$, $c_2 = b_2 = 0$, and $c_3 = a_3 = 1$ gives the circuit

shown in Figure 5.3a.



(a) $p = 2$. **(b)** $p = 2$, variant circuit.

(c) $p = 1$. **(d)** $p = 3$.

Figure 5.3: Instances of the structure from Figure 5.1 with different choices of p, all implementing a distance gate with active values 011 and 100.

The same uncontrolled distance gate may be implemented by multiple distinct circuits. Considering the same active values as before, $a_1a_2a_3 = 011$ and $b_1b_2b_3 = 100$, one can also take $c_1 = b_1 = 1$, $c_2 = a_2 = 1$, and $c_3 = b_3 = 0$, giving the circuit shown in Figure 5.3b. From Proposition 15, this circuit still implements the same uncontrolled distance gate with active values 011 and 100. One also has the freedom to choose any value of p for the circuit structure from Figure 5.1, leading to yet more implementations of the same distance gate, such as those shown in Figures 5.3c and 5.3d.

In general, when implementing a distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$, one has the freedom to make the following choices when implementing an uncontrolled distance gate using a circuit of the type shown in Figure 5.1:

1. One can choose any p with $1 \leq p \leq n$.

2. One can take either $c_i = a_i$ for $i \neq p$ and $c_p = b_p$, or $c_i = b_i$ for $i \neq p$ and $c_p =$

a_p. Proposition 15 shows that the circuits corresponding to both of these choices implement the same distance gate.

The first degree of freedom stated above gives n distinct choices for an uncontrolled distance gate acting on n qubits. When a distance gate is implemented with a particular choice of p, I will refer to x_p (as shown in Figures 5.1 and 5.2) as the *pivot* qubit. The second degree of freedom gives 2 possible implementations for every choice of pivot qubit—in one of these implementations, $c_p = a_p$, and in the other, $c_p = b_p$. Since $\neg a_p = b_p$, it follows that one of the implementations has $c_p = 0$ and the other has $c_p = 1$. The choice of c_p determines the control polarity of the CNOT gates as seen in Figure 5.2. Therefore, for a given choice of pivot qubit, a distance gate has one implementation using negative-control CNOT gates and another using positive-control CNOT gates. I will refer to these two implementations as *CNOT-negative* and *CNOT-positive* respectively. For instance, the circuit from Figure 5.3a is a CNOT-negative implementation of a distance gate with active values 011 and 100, while the circuit from Figure 5.3b is a CNOT-positive implementation of the same distance gate. From item 2 of the above list, we can see that CNOT-negative and CNOT-positive implementations can be obtained from each other by inverting the control polarity of every control input of every gate in the circuit.

An exception to the above reasoning occurs when $n = 1$. In this case, there is only one possible distance gate, which has active values 0 and 1. The whole circuit structure from Figure 5.1 reduces to just a single inverter: x_p must be the only qubit in the circuit, the CNOT gates disappear since there are no other qubits to be targeted by them, and the Toffoli gate has zero control qubits and degenerates into an inverter. Since there are no CNOT gates, there is no distinction between CNOT-negative and CNOT-positive implementations, so a single inverter is the only possible implementation of this distance gate.

To summarize, Proposition 15 leads to $2n$ different implementations for every uncontrolled distance gate acting on n qubits with $n \geq 2$, and for $n = 1$, the only possible distance gate and its implementation is the NOT gate.

5.2.2 Controlled distance gates

The circuit structure from Figure 5.1 has a very useful property. If an uncontrolled distance gate is implemented by an instance of this structure, call it G, then one can implement a controlled-G gate using a corresponding instance of the circuit structure shown in Figure 5.4. This structure is clearly derived from the earlier one shown in Figure 5.1 by adding additional control qubits q_1 through q_m. However, the control qubits act as control inputs only to the central Toffoli gate and do not participate in any of the other gates at all. This is contrary to the expectation that, in order to create a controlled variant of G, one needs to add control inputs to every gate in the circuit that implements G.

The circuit structure shown in Figure 5.4 operates correctly with the control qubits participating only in the central Toffoli gate for the following reason. As previously observed in the proof of Proposition 15, if this Toffoli gate is not active, then the remaining CNOT gates will cancel with each other, therefore producing no overall change to the qubits' states. This observation remains valid when the Toffoli gate is deactivated as the result of one or more of the newly introduced control qubits q_1 through q_m not being in the appropriate state. If these qubits are not all in the appropriate states, the Toffoli gate will not be active, the CNOT gates will cancel with each other, and the effect will be the same as if the qubits q_1 through q_m also acted as control inputs to the CNOT gates (making them into Toffoli gates as well). I call this phenomenon *implicit control*: the CNOT gates in Figure 5.4 are implicitly controlled because the circuit acts as if q_1 through q_m participated as control inputs for these

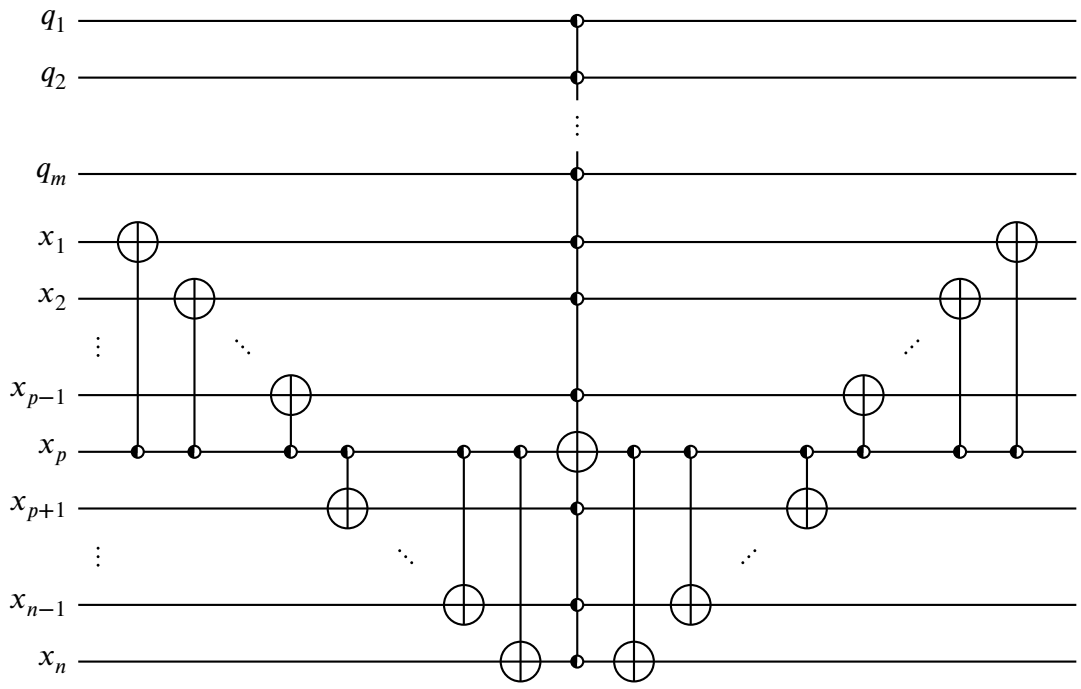


Figure 5.4: A variant of the circuit structure from Figure 5.1 with m additional control qubits.

gates, even though they do not actually so participate. In general, whenever a quantum gate G is implemented by a circuit with the structure ABA^{-1}, where A and B are arbitrary subcircuits, a controlled-G gate can be implemented by a controlled-B gate surrounded by an uncontrolled A and A^{-1} gate, as shown in Figure 5.5. The A and A^{-1} gates are implicitly controlled because, when the controlled-B gate is not active, the A and A^{-1} gates cancel, acting as if they were also controlled by q_1 through q_m even though they are not.

Any controlled distance gate realizes a function expressed by a single transposition. For instance, consider the three-qubit uncontrolled distance gate with active values 011 and 100, implemented as in Figure 5.3a. Adding a single control qubit q_1 gives the circuit shown in Figure 5.6a. Since the behavior of the original uncontrolled distance gate is known, the behavior of the circuit from Figure 5.6a immediately follows. When q_1 is in the state $|1\rangle$, the circuit acts as an uncontrolled distance gate on qubits x_1 through x_3, exchanging the states

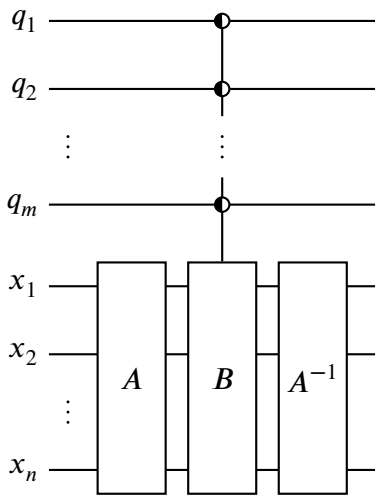
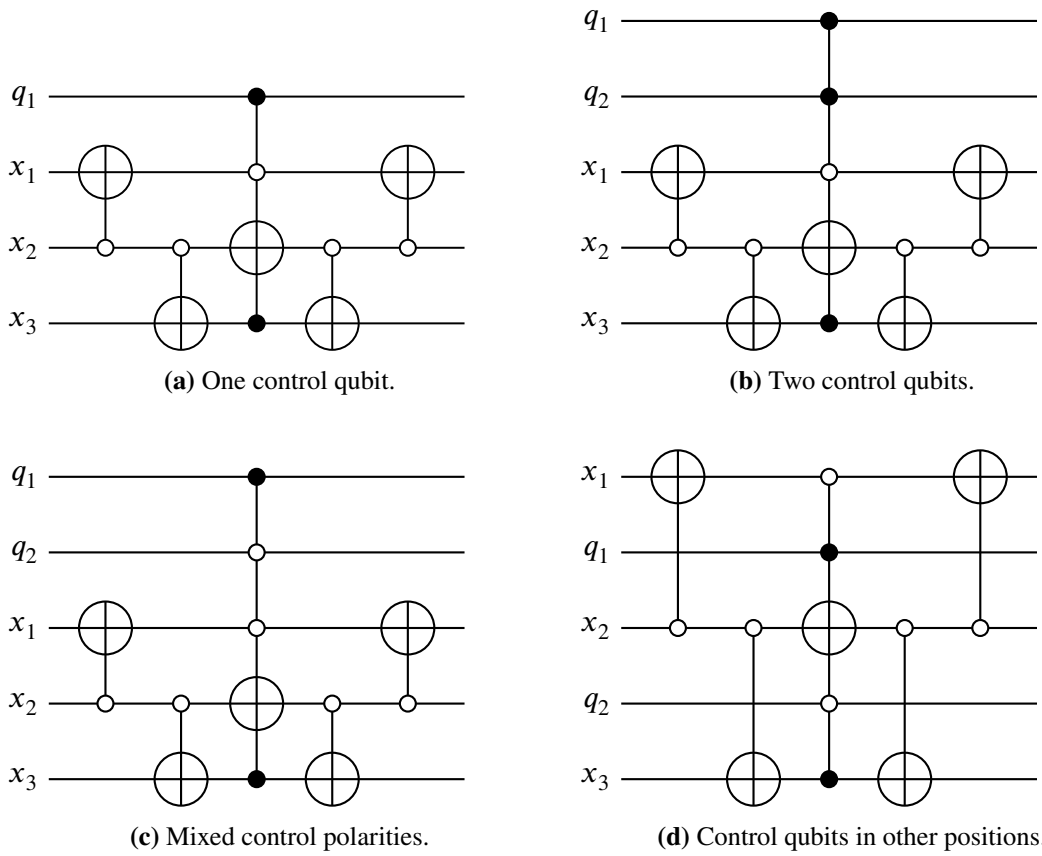


Figure 5.5: The general setup for an implicitly controlled gate.

$|011\rangle$ and $|100\rangle$. When q_1 is in the state $|0\rangle$, the circuit is deactivated and no net change is made to any of the qubits' states. Therefore, considering the state of all four qubits together, the circuit acts to exchange the states $|1011\rangle$ and $|1100\rangle$, which means that it realizes the function expressed by the transposition (1011 1100).

Controlled distance gates with multiple control qubits operate in a similar fashion. If another control qubit q_2 is added to the circuit from Figure 5.6a, the same reasoning as before shows that the resulting doubly-controlled distance gate realizes the function expressed by (11011 11100). If q_2 is modified to have negative polarity, the resulting transposition is instead (10011 10100). There is no requirement that the control qubits must be considered the most significant qubits in the circuit. If, for instance, q_1 and q_2 are now assigned to the second and fourth most significant positions, respectively, so that the order of qubits from most to least significant is x_1, q_1, x_2, q_2, x_3, then the order of bits in the resulting transposition is correspondingly affected, giving (01101 11000). All of these possibilities are shown in Figures 5.6b through 5.6d.

The following proposition provides a generalized form of the observations made in the



(a) One control qubit.

(b) Two control qubits.

(c) Mixed control polarities.

(d) Control qubits in other positions.

Figure 5.6: Controlled variants of the circuit from Figure 5.3a.

previous examples.

Proposition 16. *Let G be an uncontrolled distance gate that acts on qubits x_1 through x_n, with active values $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$. Suppose that a controlled-G gate has m additional control qubits, q_1 through q_m, with corresponding control polarities p_1 through p_m, so that $p_i = 1$ if q_i acts as a positive-polarity control and $p_i = 0$ otherwise. Then such a controlled-G gate realizes the function expressed by the single transposition*

$$(p_1 p_2 \ldots p_m a_1 a_2 \ldots a_n \quad p_1 p_2 \ldots p_m b_1 b_2 \ldots b_n) \tag{5.1}$$

when the qubits are arranged in the order $q_1, \ldots, q_m, x_1, \ldots, x_n$.

Proof. If every qubit q_i is in the corresponding state $|p_i\rangle$, then the controlled gate is active, so it acts as the uncontrolled distance gate G on qubits x_1 through x_n. Therefore, if x_1 through x_n begin in the states $|a_1\rangle$ through $|a_n\rangle$, respectively, then they will end in the states $|b_1\rangle$ through $|b_n\rangle$, respectively, and vice versa. The controlled gate thus acts to exchange the states $|p_1 p_2 \ldots p_m a_1 a_2 \ldots a_n\rangle$ and $|p_1 p_2 \ldots p_m b_1 b_2 \ldots b_n\rangle$. If x_1 through x_n begin in any other states, from Definition 14, their states will remain unchanged.

If one or more of the qubits q_i is not in the corresponding state $|p_i\rangle$, then the controlled gate is not active, so it will not make any net change to the states of x_1 through x_n. We see that the controlled gate exchanges $|p_1 p_2 \ldots p_m a_1 a_2 \ldots a_n\rangle$ and $|p_1 p_2 \ldots p_m b_1 b_2 \ldots b_n\rangle$ and leaves all other states unchanged, which means that it realizes the function expressed by (5.1). □

If the qubits in Proposition 16 are arranged in an order other than $q_1, \ldots, q_m, x_1, \ldots, x_n$, then the transposition realized by the gate G is obtained by rearranging the bits of (5.1) in the corresponding order. For instance, if the order is $q_1, \ldots, x_1, q_m, \ldots, x_n$, then q_m and x_1 have been swapped, so the transposition realized is $(p_1 p_2 \ldots a_1 p_m a_2 \ldots a_n \quad p_1 p_2 \ldots b_1 p_m b_2 \ldots b_n)$.

5.2.3 Realization of arbitrary transpositions using uncontrolled and controlled distance gates

From Propositions 15 and 16, we can see that uncontrolled and controlled distance gates are together capable of realizing any transposition with any number of bits. Specifically, suppose that one wishes to realize the n-bit transposition $(u_1 u_2 \ldots u_n \; v_1 v_2 \ldots v_n)$.[1] One can then separate the u_i's and v_i's into two sets: one for which $u_i = v_i$ and another for which

[1] The new letters u and v are used here to avoid suggesting a fixed relationship with the a_i's, b_i's, and p_i's used before. In other words, u_1 does not necessarily correspond to an a_1, b_1 or p_1 from before, but may end up corresponding to a_i (or b_i, or p_i) for any index i when the procedure described in the main text is applied.

$u_i \neq v_i$ (which implies $\neg u_i = v_i$). If the first set is empty, *i.e.*, $u_i \neq v_i$ for all i, then the transposition can be realized by an uncontrolled distance gate, because u_1 through u_n and v_1 through v_n then correspond to a_1 through a_n and b_1 through b_n in Definition 14. On the other hand, if the first set is nonempty, *i.e.*, $u_i = v_i$ for one or more i, then the transposition $(u_1 u_2 \ldots u_n \; v_1 v_2 \ldots v_n)$ is of the form (5.1), possibly with different ordering of bits: letting m be the size of the first set and n be the size of the second set, the u_i's in the first set, which are equal to the corresponding v_i's by definition, correspond to p_1 through p_m in (5.1), while the remaining u_i's and v_i's correspond to a_1 through a_n and b_1 through b_n, respectively. The transposition can then be realized using the controlled-G gate described in Proposition 16.

As an example, consider again the controlled distance gate from Figure 5.6a. We previously saw that this controlled distance gate realizes the transposition (1011 1100). However, suppose that we were instead presented with this transposition without knowledge of the distance gate used to realize it. To realize the transposition using a distance gate, we proceed as follows. Letting $(1011\ 1100) = (u_1 u_2 u_3 u_4 \;\; v_1 v_2 v_3 v_4)$, we see that $u_1 = v_1$ while $u_i \neq v_i$ for $i = 2, 3, 4$. This transposition is therefore of the form (5.1) with $m = 1$ and $n = 3$: we can take $p_1 = u_1 = v_1$ and $a_i = u_{i+1}, b_i = v_{i+1}$ for $i = 1, 2, 3$. Proposition 16 then tells us that the transposition can be realized by a controlled-G gate where G is an uncontrolled distance gate: G should have active values $a_1 a_2 a_3 = u_2 u_3 u_4 = 011$ and $b_1 b_2 b_3 = v_2 v_3 v_4 = 100$, and it should be controlled by a single control qubit with positive polarity.

Note that the above procedure does not give a unique realization of the transposition in terms of CNOT and Toffoli gates, because the resulting uncontrolled distance gate G itself has multiple possible implementations. The circuit from Figure 5.6a is one possible realization of the transposition (1011 1100). Other realizations can be obtained by adding a control qubit to any of the circuits from Figure 5.3, since all of them realize a distance gate

with active values 011 and 100, which is the gate G that was needed in the above example. A controlled distance gate therefore has as many implementations as its target gate, because control qubits can be added to any of the target gate's possible implementations to produce an implementation of the controlled gate.

There is one case in which a controlled distance gate has only one implementation, that being when the two elements of the transposition realized by the gate differ in only a single bit. In that case, the target uncontrolled distance gate G is then just a NOT gate. For instance, the transposition $(1010\ 1011)$ is put into the form of (5.1) by letting $m = 3$ and $n = 1$, which means that p_1 through p_3 correspond to the first three bits and the target uncontrolled distance gate G realizes the transposition $(a_1\ b_1) = (0\ 1)$. Then, G is just an inverter acting on the fourth qubit and the other three qubits are control qubits, which makes the whole controlled distance gate just a Toffoli gate. This reasoning also happens to show that Toffoli gates are a special case of controlled distance gates appearing when the two elements of a transposition differ by a single bit.

Like uncontrolled distance gates, the implementation of a controlled distance gate can be described in terms of a choice of pivot qubit and whether the implementation is CNOT-negative or CNOT-positive. These parameters simply apply to the target gate of the controlled distance gate, which is an uncontrolled distance gate. The choice of pivot qudit is therefore limited to the qubits that participate in that target gate. In other words, when implementing a controlled distance gate that realizes a transposition $(u_1u_2 \ldots u_n\ \ v_1v_2 \ldots v_n)$, the possible pivot qubits are limited to the positions i for which $u_i \neq v_i$. For a fixed choice of pivot qubit, the CNOT-negative and CNOT-positive implementations of a controlled distance gate differ in that the control polarities for the gate's control qubits are *not* inverted, while all other control polarities are. For example, the circuit from Figure 5.6b is a CNOT-negative

implementation of a controlled distance gate that has x_1, x_2 and x_3 as target qubits and q_1 and q_2 as control qubits. The corresponding CNOT-positive implementation is therefore obtained by inverting the control polarities for x_1, x_2, and x_3 everywhere in the circuit, as demonstrated by Figure 5.7, but the control polarities for q_1 and q_2 are unaffected because they are control qubits for the controlled distance gate.

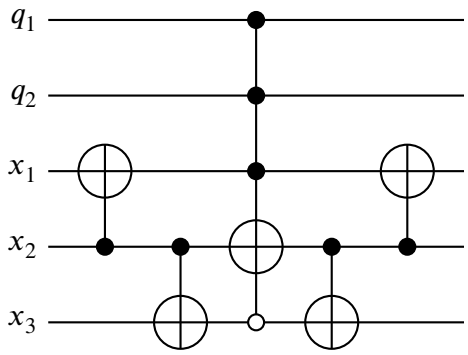


Figure 5.7: CNOT-positive counterpart of the CNOT-negative implementation of a controlled distance gate from Figure 5.6b.

In summary, if the two elements of a transposition differ in all of their bits, then the transposition can be realized using an uncontrolled distance gate; otherwise, the transposition can be realized using a controlled distance gate where the control qubits correspond to the bits that do not differ between the two elements of the transposition. The measure of number of bits that differ between two binary strings is known as the *Hamming distance*. We may therefore say that an uncontrolled distance gate always realizes a transposition where the Hamming distance between the two elements is the maximum possible. This is the origin of the name "distance gate". If the Hamming distance between the two elements of a transposition is d, then the uncontrolled or controlled distance gate that realizes that transposition has $2d$ possible implementations, unless $d = 1$, in which case there is only one possible implementation.

We can see that uncontrolled and controlled distance gates perform similar functions, so from now on, I will refer to both as "distance gates" without further qualification unless the distinction is necessary. In Chapter 6, I will demonstrate how, in a multiple-valued setting, uncontrolled and controlled distance gates can be both seen as particular instances of a single circuit structure.

5.3 Straightforward realization of permutative functions using distance gates

In Section 5.1, I reviewed how a permutative function can be expressed as a set of cycles, and in Section 5.2, I showed that all transpositions can be realized using distance gates, which can in turn be implemented using CNOT and Toffoli gates. Therefore, any permutative function can be realized by breaking down its constituent cycles into transpositions. Since every cycle is itself a permutative function, an arbitrary sequence of cycles can be combined via the operation of function composition. Conversely, breaking down a cycle into a sequence of other cycles is equivalent to decomposing a function into the composition of two or more other functions. For instance, any cycle of length n with $n \geq 3$ can be broken down into a sequence[2] of a transposition followed by a cycle of length $n-1$ using the formula

$$(x_1\ x_2\ x_3 \ldots x_n) = (x_1\ x_2) \cdot (x_1\ x_3\ x_4\ \ldots x_n), \tag{5.2}$$

[2]In group theory and other areas of mathematics where cycles are encountered, it is customary to notate the composition of cycles (and all other permutations) in a right-to-left manner. For instance, if P_1 and P_2 are two permutations, then $P_1 P_2$ denotes the result of applying P_1 *following* P_2. This convention is by analogy with function composition, which is usually also notated in a right-to-left manner; *e.g.*, $f(g(x))$ denotes the result of applying the function f *following* the function g. However, notating the composition of cycles in a left-to-right fashion is more convenient here because it matches the notation used for quantum circuits: gates in a quantum circuit execute from left to right. I therefore use a left-to-right convention for combining cycles throughout this and the following chapter. I describe this notation as a *sequence* rather than *composition* of cycles to acknowledge that it is a deviation from the established convention with regards to the ordering of cycles when they are composed.

which can be verified by observing that the permutations on both sides act identically on the values x_1 through x_n.

By repeatedly applying (5.2), any cycle can eventually be reduced to a sequence of transpositions. For instance, consider the function given by Table 5.1, whose cycle representation was obtained in Section 5.1 as (0 1 4)(2 6 7 3). The two cycles making up this permutation can be broken down using (5.2) as

$$(0\ 1\ 4) = (0\ 1) \cdot (0\ 4), \tag{5.3}$$

$$(2\ 6\ 7\ 3) = (2\ 6) \cdot (2\ 7\ 3) = (2\ 6) \cdot (2\ 7) \cdot (2\ 3), \tag{5.4}$$

whereupon the resulting transpositions can be realized using distance gates, resulting in the circuit shown in Figure 5.8.

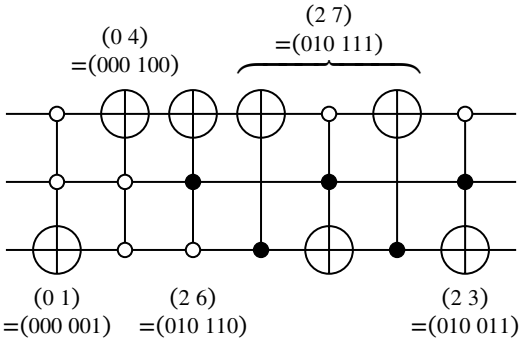


Figure 5.8: Realization of (0 1 4)(2 6 7 3) using distance gates, with the transposition realized by each individual distance gate marked.

The decomposition of a cycle into a sequence of transpositions is not unique, and the circuit resulting from distance-gate-based realization of a permutative function is sensitive to the choice of decompositions. The cycle (0 1 4), for instance, can also be decomposed into transpositions as (0 4) · (1 4). Using this decomposition for (0 1 4) but keeping the

same decomposition from (5.4) for (2 6 7 3) produces another realization of the function (0 1 4)(2 6 7 3), which is shown in Figure 5.9. Comparing Figures 5.8 and 5.9, we can see that the use of a different decomposition has increased the size of the circuit because the distance gate corresponding to (1 4) requires three gates to implement, while the ones corresponding to (0 1) and (0 4) are just single Toffoli gates.

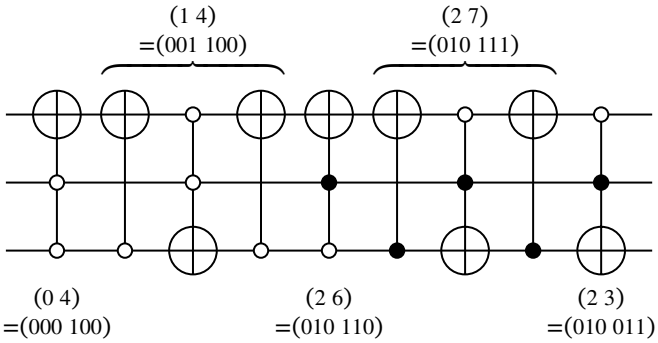
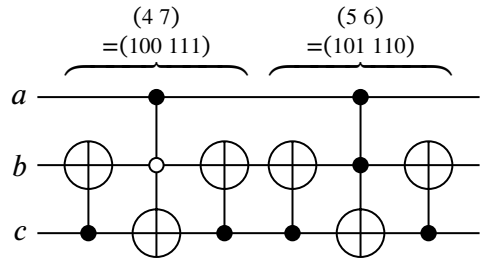


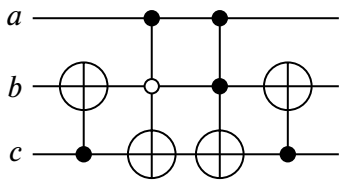
Figure 5.9: Another realization of (0 1 4)(2 6 7 3) using a different decomposition of (0 1 4) than the one used in Figure 5.8.

Another degree of freedom in distance-gate-based realization is the selection of an implementation for each distance gate. Consider realizing the permutation (4 7)(5 6) using distance gates. In terms of individual bits, these transpositions are (100 111) and (101 110). In both transpositions, the Hamming distance between the two elements is greater than one, so both corresponding distance gates have multiple implementations. If the distance gates are implemented as shown in Figure 5.10a, then two adjacent CNOT gates are formed at the juncture between the two distance gates, and these CNOT gates can be canceled to produce the circuit shown in Figure 5.10b. If, on the other hand, the distance gates are implemented using different pivot bits, as in Figure 5.10c, then this simplification is not possible.

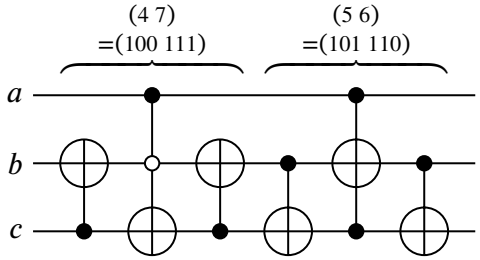
From the preceding examples, we see that distance-gate-based realization of a permutative function involves many choices, all of which potentially affect the quantum cost of



(a) Two distance gates implemented with the same pivot bit.



(b) Result of canceling adjacent CNOT gates in Figure 5.10a.



(c) The same two distance gates but implemented with different pivot bits.

Figure 5.10: Different implementations of the same distance gates, showing that the choice of implementation affects whether simplifications are possible.

the circuit that is ultimately produced. The next section describes an algorithm that attempts to make choices allowing for simplifications of the resulting circuit, similar to the one demonstrated in Figure 5.10.

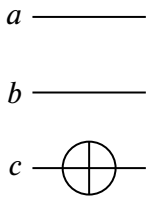
5.4 Distance gate reduction and compound distance gates

5.4.1 Motivation for and demonstration of distance gate reduction

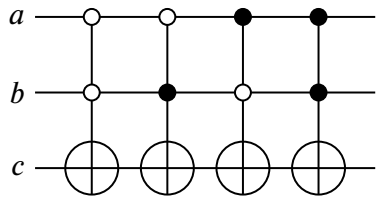
In some cases, attempting to realize a Boolean permutative function in a straightforward manner using distance gates leads to a clearly and drastically suboptimal result. For instance, consider a permutative function that takes three inputs a, b, c and transforms them as follows:

$$a' = a \qquad b' = b \qquad c' = \neg c \tag{5.5}$$

where a', b', and c' are the outputs of the function. In other words, the function simply leaves inputs a and b unchanged and inverts c. The optimal realization of this function using a quantum circuit is obviously a single inverter acting on qubit c, as shown in Figure 5.11a. However, consider what happens when the function is realized using distance gates. In permutation form, this function is represented by the cycles (0 1)(2 3)(4 5)(6 7). These transpositions are realized using controlled distance gates, which in this case turn out to just be Toffoli gates as qubits a and b both function as control qubits, leaving a single inverter as the distance gate targeting c. The realization procedure using distance gates leads to the circuit shown in Figure 5.11b. This circuit is obviously highly non-optimal as it uses four Toffoli gates for a function that can be realized by just a single inverter.



(a) Optimal realization of the function.



(b) Circuit generated by realizing individual transpositions using distance gates.

Figure 5.11: Comparison of realizations of the function described in (5.5).

To remedy the shortcoming exemplified by Figure 5.11, I introduce the concept of *distance gate reduction* in this section. Distance gate reduction allows multiple transpositions to be realized simultaneously by combining their corresponding distance gates into a simplified form, which I call a *compound distance gate*.

We have in fact already seen a glimpse of distance gate reduction, in the form of the simplification demonstrated in Figure 5.10. After cancellation of the CNOT gates in Figure 5.10a, which produces the circuit of Figure 5.10b, a further simplification can in fact be made. The two adjacent Toffoli gates in Figure 5.10b both target qubit c and both have a and

b as control qubits, so the operations they perform can be combined using Boolean algebra. Specifically, the first Toffoli gate is active when $a \wedge \neg b = 1$ and the second is active when $a \wedge b = 1$, so one of the two gates is active when

$$(a \wedge \neg b) \vee (a \wedge b) = a \wedge (\neg b \vee a) = a = 1. \tag{5.6}$$

(It is impossible for both Toffoli gates to be simultaneously active since the conditions $a \wedge \neg b = 1$ and $a \wedge b = 1$ are mutually exclusive.) These two Toffoli gates may therefore be replaced by a CNOT gate with a as its control qubit and c as its target qubit, as shown in Figure 5.12, removing qubit b entirely from the involvement of the gate.

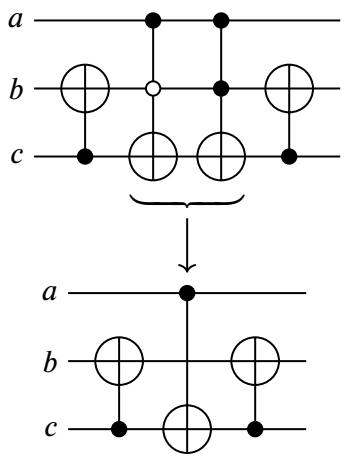
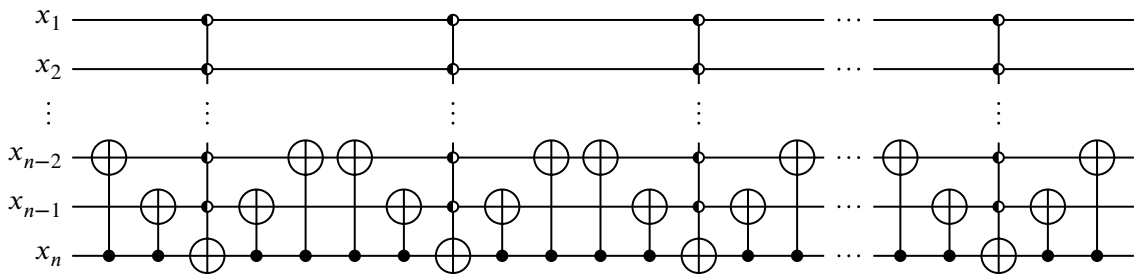


Figure 5.12: Further simplification of the circuit from Figure 5.10b.

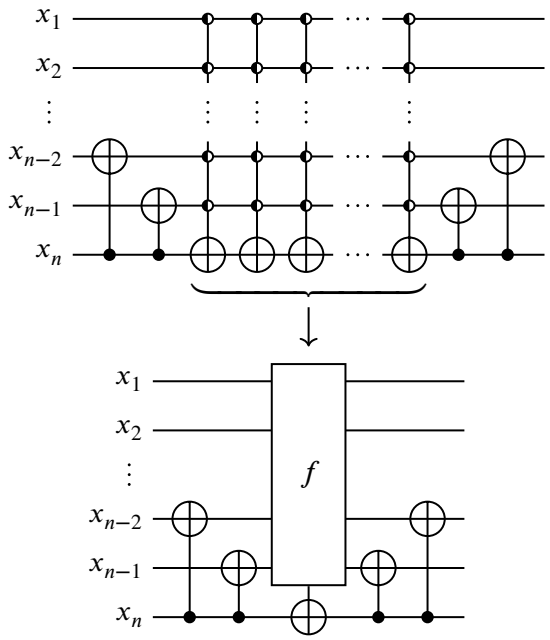
The type of simplification demonstrated in Figures 5.10a, 5.10b, and 5.12 can in general be applied to any sequence of distance gates having the form depicted in Figure 5.13a. This figure shows a sequence of distance gates that are all implemented as CNOT-positive using the same pivot qubit, x_n. The Toffoli gates used to implement the distance gates might have any combination of control polarities, so their control inputs are represented with half-filled

dots like in Figure 5.1. In such a sequence, the CNOT gates from every neighboring pair of distance gates may be canceled, leaving only the CNOT gates at the start and end with consecutive Toffoli gates in the middle. As Figure 5.13b demonstrates, these Toffoli gates then all share the same target qubit and same set of control qubits, so they can be replaced by the realization of a Boolean function f. This function f is determined by the control polarities of the original Toffoli gates—f evaluates to 1 when one of the Toffoli gates would be active and 0 otherwise. The idea is that f will likely have a realization with lower quantum cost than the combined cost of of the original Toffoli gates. For instance, in Figure 5.10a, $f(a, b) = (a \wedge \neg b) \vee (a \wedge b)$, and as (5.6) shows, this can be simplified to $f = a$, which then has a realization as a single CNOT gate as shown in Figure 5.12.

In general, then, distance gate reduction is the process of combining certain sets of distance gates into a circuit that looks like the implementation of a single distance gate, but with the central Toffoli gate replaced by the realization of a Boolean function f, as shown in Figure 5.13b. Such a circuit, being equivalent to the original sequence of distance gates from which it is derived, therefore realizes a whole set of transpositions at once. This set of transpositions always satisfies certain properties that are discussed in the next section. I will refer to any gate that realizes such a set of transpositions as a *compound distance gate*, and I will call the function f in Figure 5.13 the *compounding function* of the corresponding compound distance gate. In an n-qubit circuit, the compounding function always takes $n - 1$ inputs. Using this terminology, Figure 5.13 then shows how an implementation of a compound distance gate that realizes a set of transpositions T can be derived from implementations of the distance gates that realize the individual transpositions belonging to T. Specifically, an implementation of the compound distance gate is obtained by realizing its compounding function and then surrounding the resulting circuit with CNOT gates as in



(a) A sequence of distance gates showing CNOT gates that can be canceled.



(b) The circuit resulting from cancellation of the CNOT gates, which produces a sequence of adjacent Toffoli gates that can be realized as a Boolean function f.

Figure 5.13: Distance gate reduction in the general case.

Figure 5.13b.

5.4.2 Compatible transpositions

The simplification demonstrated in Figure 5.13 is only possible when all of the distance gates in the sequence are implemented using CNOT gates with the same control and target qubits, since only then can those CNOT gates be canceled. Therefore, the use of distance gate reduction requires the following conditions to be met:

1. Each distance gate must have the same sets of control and target qubits. For the purpose of this discussion, an uncontrolled distance gate can be thought of as having an empty set of control qubits.

2. Each distance gate must be implemented using the same choice of pivot qubit.

3. Each distance gate must be implemented using the same CNOT type, *i.e.*, the distance gates must either be all CNOT-positive or CNOT-negative.

Conditions 2 and 3 are only concerned with choosing appropriate implementations for the distance gates, but condition 1 places restrictions on what sets of transpositions can be realized together using distance gate reduction. Specifically, it implies, via Proposition 16 and the discussion following it, that in every transposition being realized, the two elements must differ in the same bit-positions. For instance, the two transpositions realized in Figure 5.10a are $(4\ 7) = (100\ 111)$ and $(5\ 6) = (101\ 110)$. In both of these transpositions, the two elements differ in their second and third bits and agree in their first bits, so that when realized using distance gates, both distance gates have the first qubit (labeled a in Figure 5.10a) as their control qubit and b and c as their target qubits. In contrast, if we replace the second transposition $(5\ 6)$ by $(3\ 6) = (011\ 110)$, then the resulting distance gate has b as its

control qubit and a and c as its target qubits. This distance gate can never undergo distance gate reduction when combined with the distance gate for (4 7), because of the mismatch between the two gates' sets of target qubits. Figure 5.14 shows how the neighboring CNOT gates from the two distance gates are unable to cancel. Although Figure 5.14 shows only one specific implementation for each distance gate, using other implementations makes no difference—no matter what implementations are used, the CNOT gates from the first distance gate must act on qubits b and c while the CNOT gates from the second distance gate act on qubits a and c, so they can never cancel.

If two transpositions do meet the above-described condition, that is, the two element of both transpositions differ in the same set of bits, then distance gate reduction for the corresponding distance gates is always possible. One can choose any pivot qubit and use CNOT-positive implementations for both distance gates. In Figure 5.10c, distance gate reduction is blocked because the two distance gates are implemented using different pivot qubits, but the transpositions realized by those gates agree in the set of bits that differ between their elements. Therefore, distance gate reduction can take place if both distance gates are instead implemented using the same pivot qubit, which is what was done in Figure 5.10a.

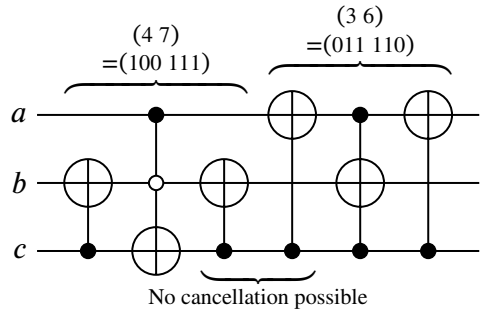


Figure 5.14: Distance gate reduction is blocked when two distance gates have different sets of target qubits.

The preceding discussion applies equally well to sets of more than two transpositions. If the two elements of every transposition in a set differ in the same set of bits, distance gate reduction for the corresponding distance gates will be possible when they are all implemented in CNOT-positive fashion using the same pivot qubit. One can also use CNOT-negative implementations for every distance gate, but this makes no significant difference compared to the CNOT-positive option. In Figure 5.13, for example, switching to CNOT-negative implementations will simply result in the control polarities of x_{n-1} and x_{n-2} being inverted in every Toffoli gate, so the resulting compounding function f will be the same as before but with x_{n-1} and x_{n-2} replaced by $\neg x_{n-1}$ and $\neg x_{n-2}$.

Definition 17. Two transpositions are *compatible* if, for both transpositions, the two elements of the transposition differ in the same set of bits. A set of transpositions is compatible if every transposition in the set is compatible with every other transposition in the set.

Therefore, a compatible set of transpositions is one where the two elements of every transposition in the set differ in the same set of bits. With this definition, a set of transpositions can be realized using distance gate reduction if and only if it is a compatible set. Conversely, a compound distance gate always realizes a compatible set of transpositions. It is easy to see that every transposition is compatible with itself.

Compatible sets of transpositions have the following important property.

Proposition 18. *Suppose that a transposition* $(a\ b)$ *is given, where* $a, b \in \{0,1\}^n$ *for some n. Then the set of all transpositions compatible with* $(a\ b)$ *forms a partition of* $\{0,1\}^n$.

Proof. Define δ to be the bitwise exclusive-OR of a and b, denoted $a \oplus b$. Then $\delta \in \{0,1\}^n$ and the i-th bit of δ is equal to 1 if and only if a and b differ in their i-th bits. Now suppose that an arbitrary $x \in \{0,1\}^n$ is given. Define $y = x \oplus \delta$. Then $\delta = x \oplus y$. But this means

that the i-th bit of δ is equal to 1 if and only if x and y differ in their i-th bits. Therefore, x and y differ in the same set of bits as a and b do, so $(x\ y)$ is compatible with $(a\ b)$. Since x was an arbitrary member of $\{0,1\}^n$, this means that every member of $\{0,1\}^n$ belongs to a transposition that is compatible with $(a\ b)$. Furthermore, no member of $\{0,1\}^n$ can belong to more than one transposition that is compatible with $(a\ b)$: if both $(x\ y)$ and $(x\ z)$ are compatible with $(a\ b)$, then we must have $x \oplus y = \delta$ and $x \oplus z = \delta$, which implies $x \oplus y = x \oplus z$ and $y = z$. Since every member of $\{0,1\}^n$ therefore belongs to exactly one transposition that is compatible with $(a\ b)$, the set of all such transpositions forms a partition of $\{0,1\}^n$. □

The above proof shows that given a transposition $(a\ b)$ and an x (with $a, b, x \in \{0,1\}^n$), it is easy to find y such that $(x\ y)$ is compatible with $(a\ b)$, and furthermore this y is unique.

5.5 Realization of permutative functions with distance gate reduction

5.5.1 Extraction of compatible transpositions from cycles

If we have managed to represent a permutative function as a sequence of compatible sets of transpositions, then every set in this sequence can be realized using a compound distance gate, thereby realizing the whole function. However, it is not yet clear how a function, represented as a set of cycles, can be transformed into such a sequence of sets of compatible transpositions.

In order to form a set of compatible transpositions from an arbitrary set of cycles, we may suppose that we select one transposition from the set of cycles and attempt to find other transpositions that are compatible with the selected transposition. Of course, it may be the case that the set of cycles contains no transpositions at all. In that case, we may select any

cycle from the set and decompose it via (5.2) into a transposition plus another cycle. Once we have obtained a transposition, call it t, we then search for other transpositions that are compatible with t and extract them from the cycles making up the function to be realized.

Proposition 18 shows that it is easy to find all transpositions compatible with t. Why not then take every single one of these transpositions? Doing so would lead to a sequence of distance gates, as in Figure 5.13, where the Toffoli gates together represent all possible combinations of control polarities, giving a compounding function that is just a constant 1. This would produce a compound distance gate where the entire block labeled f in Figure 5.13b reduces to just a single inverter, which would of course have a very low quantum cost. The answer to the above question is that transpositions cannot just be conjured out of thin air—their elements must already be present in the cycle representation of the function to be realized, or else extracting them from the function will only increase the total size of its cycles.

The following example serves to clarify the reasoning of the previous paragraph. Suppose that we are given the 3-bit permutative function with cycle representation (0 3)(1 2 4) and we wish to extract a set of compatible transpositions from this function. We choose $t = (0\ 3)$ since it is the only transposition present. The two elements of $(0\ 3) = (000\ 011)$ differ in their last two bits, so the other transpositions that are compatible with t are $(1\ 2) = (001\ 010)$, $(4\ 7) = (100\ 111)$, and $(5\ 6) = (101\ 110)$, all of which contain two elements that also differ in their last two bits. Attempting to extract all of these compatible transpositions from the

function produces

$$\begin{aligned}(0\ 3)(1\ 2\ 4) &= (0\ 3)(1\ 2)(4\ 7)(5\ 6)(5\ 6)(4\ 7)(1\ 2)(1\ 2\ 4)\\ &= (0\ 3)(1\ 2)(4\ 7)(5\ 6)(5\ 6)(4\ 7)(1\ 4)\\ &= \big[(0\ 3)(1\ 2)(4\ 7)(5\ 6)\big](5\ 6)(1\ 4\ 7),\end{aligned} \tag{5.7}$$

where the bracketed part of (5.7) indicates the transpositions that have been extracted from the function. If we remove the bracketed transpositions, realizing them with a compound distance gate, we are left with (5 6)(1 4 7). Realizing this set of compatible transpositions has therefore not really made any progress towards realizing the original function: there is still one transposition and one cycle of length 3, and the elements 5, 6, and 7, which did not appear in the original cycles (0 3)(1 2 4) at all, have been introduced.

In contrast, if we restrict ourselves to only extracting transpositions whose elements already appear in the original set of cycles, then we obtain much better results. Of the transpositions that are compatible with $t = (0\ 3)$ above, only two are made up entirely of elements that are already present in the function: t itself and $(1\ 2) = (001\ 010)$. Extracting only these two transpositions from the original set of cycles gives

$$\begin{aligned}(0\ 3)(1\ 2\ 4) &= (0\ 3)(1\ 2)(1\ 2)(1\ 2\ 4)\\ &= \big[(0\ 3)(1\ 2)\big](1\ 4),\end{aligned} \tag{5.8}$$

which expresses those cycles as a compatible set of transpositions (0 3)(1 2) followed by another single transposition (1 4). This compatible set can then be realized by a compound distance gate and the single transposition realized by another distance gate, successfully producing a realization of the original function as shown in Figure 5.15.

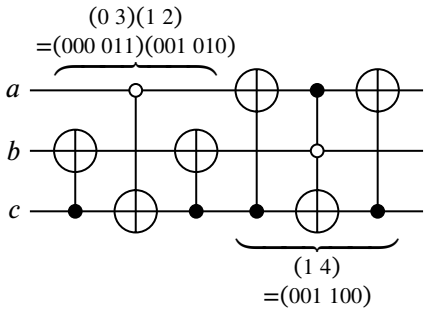


Figure 5.15: Realization of the function (0 3)(1 2 4) by extracting a compatible set of transpositions.

In some cases, extracting all of the transpositions whose elements appear in the existing cycles may not be advantageous either. For instance, suppose we are now faced with a 4-bit function with the cycle representation (0 15)(1 14 3 6)(2 4 11)(5 10 13). Again, this set of cycles contains only one transposition, (0 15), so we take $t = (0\ 15)$. Besides t itself, there are four other transpositions compatible with t and with both elements already appearing in the set of cycles: (1 14), (2 13), (4 11), and (5 10). To avoid tedious repetition of "both elements already appearing in the set of cycles" and similar phrases, I will from now on describe these transpositions as being *available* in the set of cycles. I will also use *t-compatible* as an abbreviation for "compatible with t". Using this terminology, the four transpositions previously listed, together with t itself, are all of the t-compatible transpositions available in (0 15)(1 14 3 6)(2 4 11)(5 10 13).

Now first consider what happens if we realize all five transpositions at once using a single compound distance gate. To do so, we must first select a pivot qubit to be used when implementing the compound distance gate; for the sake of demonstration, suppose that the qubits are labeled a through d in order and we select d (the last qubit). Then the correspondence between transpositions and terms of the compounding function is as given

Table 5.2: Terms of the compounding function for a compound distance gate realizing the transpositions (0 15)(1 14)(2 13)(4 11)(5 10), assuming that d is the pivot qubit used to implement that gate.

Transposition	Control polarities of Tof. gate in CNOT-pos. implementation of distance gate with pivot d			Corresponding term in compounding function
	a	b	c	
(0 15) = (0000 1111)	neg.	neg.	neg.	$\neg a \wedge \neg b \wedge \neg c$
(1 14) = (0001 1110)	pos.	pos.	pos.	$a \wedge b \wedge c$
(2 13) = (0010 1101)	neg.	neg.	pos.	$\neg a \wedge \neg b \wedge c$
(4 11) = (0100 1011)	neg.	pos.	neg.	$\neg a \wedge b \wedge \neg c$
(5 10) = (0101 1010)	pos.	neg.	pos.	$a \wedge \neg b \wedge c$

in Table 5.2, from which we obtain the compounding function

$$f(a,b,c) = (\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge c) \\ \vee (\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c). \tag{5.9}$$

A Karnaugh map for this function is shown in Figure 5.16a, from which we see that an ESOP expression for the function is

$$f(a,b,c) = \neg a \oplus c \oplus (\neg a \wedge \neg b \wedge c). \tag{5.10}$$

Mapping the terms of this ESOP expression to Toffoli gates and substituting the resulting realization of f into a Figure 5.13b-like circuit gives an implementation of the desired compound distance gate, which is shown in Figure 5.17a. Extracting the realized transpositions

from the original set of cycles gives

$$(0\ 5)(1\ 14)(2\ 13)(4\ 11)(5\ 10)(5\ 10)(4\ 11)(2\ 13)(1\ 14)(2\ 4\ 11)(5\ 10\ 13)(1\ 14\ 3\ 6)$$
$$= \big[(0\ 5)(2\ 13)(4\ 11)(5\ 10)(1\ 14)\big](1\ 3\ 6)(2\ 5\ 13\ 4), \quad (5.11)$$

which shows that after realizing the five bracketed transpositions, the remaining cycles to be realized are (1 3 6)(2 5 13 4).

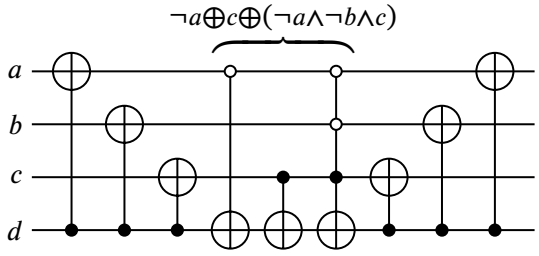
ab \ c	0	1
00	1	1
01	1	0
11	0	1
10	0	1

(a) The function from (5.9).

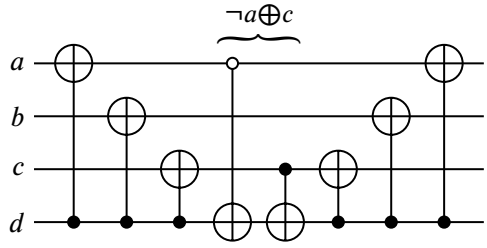
ab \ c	0	1
00	1	0
01	1	0
11	0	1
10	0	1

(b) The function from (5.12).

Figure 5.16: Karnaugh maps for compounding functions arising from realizing either all or some of the transpositions from Table 5.2.



(a) Compounding function (5.9).



(b) Compounding function (5.12).

Figure 5.17: Implementations of compound distance gates resulting from the compounding functions shown in Figure 5.16.

The last term in the ESOP expression (5.10) contains all three variables a, b, and c, and therefore produces a 3-control Toffoli gate as seen in Figure 5.17a. This term can be

eliminated if, instead of realizing all five t-compatible transpositions that are available in the original set of cycles, we choose to realize only four of them: (0 15), (1 14), (4 11), and (5 10). If we realize these four transpositions using a compound distance gate, then the compounding function becomes

$$f(a,b,c) = (\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge c) \vee (\neg a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c), \tag{5.12}$$

whose Karnaugh map is shown in Figure 5.16b, and which can be expressed in ESOP form as

$$f(a,b,c) = \neg a \oplus c. \tag{5.13}$$

Figure 5.17a shows the resulting implementation of the compound distance gate. Extracting the realized transpositions from the original cycles,

$$\begin{aligned}(0\ 5)(1\ 14)(4\ 11)(5\ 10)(5\ 10)(4\ 11)(1\ 14)(2\ 4\ 11)(5\ 10\ 13)(1\ 14\ 3\ 6)& \\ = \big[(0\ 5)(2\ 13)(4\ 11)(5\ 10)(1\ 14)\big](1\ 3\ 6)(2\ 4)(5\ 13),&\end{aligned} \tag{5.14}$$

we see that the remaining cycles to be realized are (1 3 6)(2 4)(5 13).

Comparing Figures 5.17a and 5.17b, we see that the compound distance gate from Figure 5.17b has the implementation with the lower quantum cost, since it lacks the 3-control Toffoli gate seen in Figure 5.17a. Of course, these two compound distance gates are not equivalent since one realizes five transpositions and the other only four. However, if we then compare eqs. (5.11) and (5.14), we see that the remaining cycles in both cases are comparable in size: in (5.11), the remaining cycles consist of one cycle of length 3 and another of length 4, while in (5.14), they consist of a cycle of length 3 and two transpositions. Therefore, given

that the two options—realizing either the set of five or the set of four transpositions—produce comparable sets of remaining cycles, the second option appears to be the better choice since it corresponds to the compound distance gate from Figure 5.17b, whose implementation has lower quantum cost.

The transposition-extraction strategy demonstrated by the preceding example is then to extract not necessarily all t-compatible transpositions that are available in the original set of cycles, but rather to extract a subset of these transpositions that results in a compounding function realizable with low quantum cost. In order to find such subsets of t-compatible transpositions, the compounding function can be given as a so-called *incompletely-specified* function. An incompletely-specified function is not a function in the mathematical sense; rather, it is merely a relation or a specification that can be satisfied by many different mathematical functions. Like a normal Boolean function, an incompletely-specified single-output Boolean function takes a number of Boolean-valued inputs, but in addition to output values of 0 and 1, any of the output values may also be unspecified. Such unspecified values are often called "don't-cares". When realizing an incompletely-specified Boolean function, one has the freedom to assign values of either 0 or 1 to any of the don't-cares in order to achieve a realization with low cost.

If we specify the terms of the compounding function as don't-cares, then a method for realizing Boolean functions using quantum circuits can automatically decide which terms to take in order to produce a circuit with low quantum cost. However, we cannot make every single term a don't-care because then the minimum-quantum-cost realization of the function trivially just makes the function a constant zero, which realizes no transpositions and accomplishes nothing. Instead, since our transposition-extraction strategy is based on first extracting a single transposition t and then searching for other t-compatible transpositions,

we can stipulate that at minimum, the transposition t itself must be realized, but all other t-compatible transpositions are optional. This corresponds to specifying the output of the compounding function as 1 for the term corresponding to t, but as a don't-care for all other terms corresponding to t-compatible transpositions that are available in the original set of cycles.

To see how an incompletely-specified compounding function works in practice, consider the same scenario as before, where we wish to extract a compatible set of transpositions from (0 15)(1 14 3 6)(2 4 11)(5 10 13) starting with $t = (0\ 15)$. Referring to Table 5.2, we assign the compounding function an output of 1 for the term corresponding to t, $\neg a \wedge \neg b \wedge \neg c$, which itself corresponds to the combination of inputs $a = b = c = 0$. All the other terms listed in Table 5.2 are assigned a don't-care output. This produces the Karnaugh map shown in Figure 5.18. An appropriate algorithm for finding ESOP representations of Boolean functions, such as EXORCISM-MV-2 [16, 55], can then determine that the optimal ESOP representation for this function is the one given in (5.13). In doing so, the don't-cares in Figure 5.18 are replaced with 0s and 1s, thereby recreating the previous Karnaugh map shown in Figure 5.16b. We can then determine that the transpositions actually realized are (0 5)(1 14)(4 11)(5 10) and that (2 13) is not realized.

ab \ c	0	1
00	1	–
01	–	0
11	0	–
10	0	–

Figure 5.18: Karnaugh map for an incompletely specified compounding function, derived from the terms listed in Table 5.2.

Although the end result of the procedure described in the previous paragraph is the same as in the example before that—in both cases, the transpositions (0 5)(1 14)(4 11)(5 10) are extracted from the original set of cycles and realized—the cause and effect are reversed from before. In the prior example, an *a priori* decision was made to only extract and realize the four transpositions (0 5)(1 14)(4 11)(5 10), while in this case, an incompletely-specified compounding function was created first and the set of transpositions to be extracted was determined from the result of realizing the compounding function. The task of deciding which t-compatible transpositions to realize is therefore delegated to the method or algorithm used for realizing the compounding function.

In all of the previous examples, the compounding function was realized by representing it in ESOP form, but this was simply done for convenience and there is no reason why other realization methods cannot be used. The only requirements are that the realization method must realize single-output Boolean functions using quantum circuits, and it must be capable of handling incompletely-specified functions. Notably, the DIPS-based realization of Boolean functions presented in Chapter 4 meets these requirements, since it relies on the symmetrization algorithm from [48], which works with incompletely-specified functions. Therefore, the compounding function can be realized using the DIPS-based method, although doing so does introduce additional ancillary qubits while a purely ESOP-based approach does not. Other possible choices of method to realize the compounding function include those described in [56, 57, 58, 59].

Finally, there is one more variable in play when deciding on a set of transpositions to be extracted from a permutative function and realized using a compound distance gate, and that is the choice of pivot qubit for the compound distance gate. Specifically, different choices of pivot qubit produce different compounding functions, and it is possible that the compounding

Table 5.3: Terms of the compounding function for a compound distance gate realizing the transpositions (0 15)(1 14)(2 13)(4 11)(5 10) or a subset thereof, assuming that c is the pivot qubit used to implement that gate.

Transposition	Control polarities of Tof. gate in CNOT-pos. implementation of distance gate with pivot c			Corresponding term in compounding function
	a	b	d	
(0 15) = (0000 1111)	neg.	neg.	neg.	$\neg a \wedge \neg b \wedge \neg d$
(1 14) = (0001 1110)	neg.	neg.	pos.	$\neg a \wedge \neg b \wedge d$
(2 13) = (0010 1101)	pos.	pos.	pos.	$a \wedge b \wedge d$
(4 11) = (0100 1011)	neg.	pos.	neg.	$\neg a \wedge b \wedge \neg d$
(5 10) = (0101 1010)	neg.	pos.	pos.	$\neg a \wedge b \wedge d$

function for one pivot qubit will have a lower-cost realization than the compounding function for another. For instance, consider again the same example from before, where the available t-compatible transpositions are given in Table 5.2. In Table 5.2, d was used as the pivot qubit, but since the two elements of t, $0 = 0000$ and $15 = 1111$, differ in all four bits, any qubit may be used as the pivot qubit to implement a compound distance gate realizing these compatible transpositions or a subset of them. If we choose c (the second-to-last qubit) as the pivot qubit instead, then the correspondence of transpositions to terms of the compounding function is as shown in Table 5.3. Following the same procedure as before, assigning an output of 1 to the term $\neg a \wedge \neg b \wedge \neg d$ that corresponds to the transposition t and assigning don't-cares to the other terms, we arrive at the Karnaugh map shown in Figure 5.19. The compounding function can then be realized using the expression $f(a, b, c) = \neg a$, which corresponds to a circuit containing just a single CNOT gate. The realization of the compounding function in turn produces the circuit shown in Figure 5.20 as the implementation of the resulting compound distance gate, which has even lower quantum cost than the circuit from Figure 5.17b. As a result, when extracting a compatible set of transpositions from a set of cycles, we should

also try all possible pivot qubits and choose the one that results in the compound distance gate implementation with the lowest quantum cost.

ab \ d	0	1
00	1	–
01	–	–
11	0	–
10	0	0

Figure 5.19: Karnaugh map for the compounding function corresponding to the terms listed in Table 5.3.

One last issue that has not been addressed is how the set of all t-compatible transpositions available in a given set of cycles can be found. It is possible to simply make a list of all t-compatible transpositions and then select only the ones that are available in the set of cycles, but from Proposition 18, such a list will contain 2^{n-1} transpositions where n is the number of inputs to the permutative function being realized. It is therefore more computationally efficient to use the following method. For an element x appearing in the set of cycles, there is a unique y such that $(x\ y)$ is t-compatible, which is easily computed as described in the proof of Proposition 18. If y also appears in the set of cycles, then $(x\ y)$ is available in that set. Repeating this process for every x appearing in the set of cycles gives all t-compatible transpositions available in that set, and only requires an amount of time proportional to the total number of elements appearing in the set of cycles, which is the same as the sum of their lengths.

The procedures described in this section can be formulated in terms of several algorithms. First, Algorithm 7 is a subroutine that finds all of the t-compatible transpositions available in a given set of cycles, using the method described in the previous paragraph. Algorithm 8 is

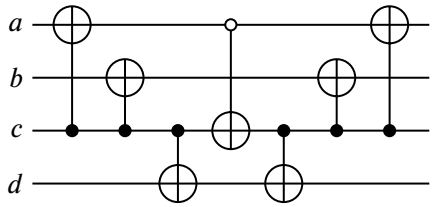


Figure 5.20: Compound distance gate implementation obtained by realizing the compounding function from Figure 5.19 as $f(a, b, c) = \neg a$.

another subroutine that maps transpositions to the terms of a compounding function, as was done in Tables 5.2 and 5.3. Finally, Algorithm 9 uses these two subroutines to choose a set of t-compatible transpositions to be realized using a compound distance gate. It assumes that an external subroutine for realizing incompletely-specified Boolean functions is available, and incorporates the process of trying all possible pivot qubits and choosing the one that produces the compounding function realizable with the lowest quantum cost.

5.5.2 Complete realization algorithm for permutative Boolean functions

Using Algorithm 9, it is now easy to see how any permutative Boolean function, represented as a set of cycles, can be realized using compound distance gates. Given a set of cycles, we first select any transposition t from the set and then use Algorithm 9 to generate a compound distance gate. If the set of cycles contains no transpositions, then, as mentioned at the beginning of Section 5.5.1, we may select any cycle from the set and select a transposition that can be extracted from that cycle via (5.2). In practice, this amounts to letting t be a transposition composed of any two consecutive elements of any cycle in the given set. After obtaining a compound distance gate an associated set of realized transpositions from Algorithm 9, we then extract those transpositions from the original set of cycles, thereby obtaining a new remaining set of cycles to be realized. This process is repeated until no more cycles are left, and the sequence of compound distance gates thus created is the realization

Algorithm 7: Find transpositions that are compatible with a given transposition and available in a given set of cycles.

Function `FindCompatibleTranspositions(`C, t`)`

Input: A nonempty set of cycles C whose elements are members of $\{0, 1\}^n$ for some positive integer n, and a transposition t .

Output: The set T of all t-compatible transpositions that are available in C.

Create a set T of transpositions, initially empty.
Let t_1 and t_2 be the two elements of t.
Let $\delta = t_1 \oplus t_2$ (the bitwise exclusive-OR of t_1 and t_2).
Let U be the set containing all elements of all cycles in C.
while U *is nonempty* **do**
    Let u be any member of U.
    Remove u from U.
    Let $u' = u \oplus \delta$.
    if u' *is in* U **then**
        Add the transposition $(u\ u')$ to T.
        Remove u' from U.
    end
end
Output T.

end

Algorithm 8: Get the term of a compounding function corresponding to a given transposition.

Function `GetCompoundingFunctionTerm(n, p, t)`

Input: A positive integer n, a transposition t with elements in $\{0,1\}^n$, and an integer p with $1 \leq p \leq n$.

Output: The input value corresponding to t in the compounding function when t is one of the transpositions realized by an n-qubit compound distance gate implemented using the p-th bit as the pivot bit.

Let G be the distance gate acting on n qubits x_1 through x_n that realizes t.
Let G_{Tof} be the central Toffoli gate of the CNOT-positive implementation of G.
foreach *i from 1 to n with* $i \neq p$ **do**
    Let c_i be the control polarity of x_i in G_{Tof}, where $c_i = 1$ and $c_i = 0$ indicate positive and negative polarities, respectively.
end
Output $c_1 c_2 \ldots c_{p-1} c_{p+1} \ldots c_{n-1} c_n$.

end

of the original function. The entire procedure is described in Algorithm 10.

The choice of transposition t in Algorithm 10 ensures that the total number of elements of the remaining cycles decreases with each iteration. Specifically, when C contains one or more transpositions, then one of them is selected as t and is guaranteed to be realized, removing it from the list of cycles. The total number of elements thus decreases by at least two. If C does not contain any transpositions, then consider the cycle κ selected in Algorithm 10. Since t is composed of two consecutive elements of κ, extraction of t from C will cause κ to be decomposed as in (5.2), therefore removing one element from κ. In either case, as discussed in Section 5.5.1, no new elements can be introduced to C because the subroutine `RealizeTranspositions` of Algorithm 9 only ever realizes transpositions that are available in C. We therefore obtain the following result.

Theorem 19. *If C is a set of cycles whose total length is L, then Algorithm 10 terminates in at most L iterations and realizes C using at most L compound distance gates.*

Algorithm 9: Find and realize a set of compatible transpositions that are all available in a given set of cycles.

Function `RealizeTranspositions(n, C, t)`

Input: A positive integer n and a set of cycles C and transposition t with elements in $\{0,1\}^n$, where t is available in C.

Output: A set of compatible transpositions T that contains t and are all available in C, together with a quantum circuit that realizes T.

External subroutines: `RealizeFunction(f, n, p)`, which realizes the incompletely-specified, $n-1$-input Boolean function f and outputs a quantum circuit acting on qubits x_1 through x_n. The circuit must accept the inputs to the function through all of the qubits other than x_p and indicate an output of 1 by inverting x_p.

Let $T' =$ `FindCompatibleTranspositions`(C, t).

Let P be the set of all integers p, $1 \leq p \leq n$, such that the elements of t differ in their p-th bits.

foreach *p in P* **do**

  Let f be an incompletely-specified Boolean function with all outputs initially set to 0.

  foreach *t′ in T′* **do**

    Let $X =$ `GetCompoundingFunctionTerm(n, p, t')`.

    if $t' = t$ **then**

      Specify the output of $f(X)$ as 1.

    else

      Let the output of $f(X)$ be unspecified/a don't-care.

    end

  end

  Let $Q_p =$ `RealizeFunction(f, n, p)`.

end

Let $p_{\min}$ be the member of P for which $Q_{p_{\min}}$ has the lowest quantum cost. If multiple such members of P exist, let $p_{\min}$ be any one of them.

foreach *p in P with* $p \neq p_{\min}$ **do**

  To the beginning and end of $Q_{p_{\min}}$, add a positive-control CNOT gate with control qubit $x_{p_{\min}}$ and target qubit x_p.

end

Let T be the set of transpositions realized by $Q_{p_{\min}}$.

Output T and $Q_{p_{\min}}$.

end

Algorithm 10: Realize a set of cycles using a quantum circuit composed of compound distance gates.

Input: A positive integer n and a set of cycles C with elements in $\{0,1\}^n$.
Output: A quantum circuit that realizes the permutative n-input, n-output Boolean function represented by C.

Create an n-qubit quantum circuit Q initially containing no gates.
while *C is nonempty* **do**
    if *C contains at least one transposition* **then**
        Let t be any transposition contained in C.
    else
        Let κ be any cycle contained in C.
        Let t be any transpositions composed of two consecutive elements of κ.
    end
    Let $(T, Q') = \texttt{RealizeTranspositions}(n, C, t)$.
    Append Q' to the end of Q.
    Compute C' such that $C = T \cdot C'$.
    Let $C = C'$.
end
Output Q.

5.6 Conclusion

The first part of this chapter introduced the concept of distance gates for binary reversible and quantum circuits. Distance gates are defined to exchange exactly two of their possible input values, *i.e.*, they realize transpositions. I demonstrated the implementation of distance gates using a symmetrical circuit structure consisting of CNOT and Toffoli gates. This symmetrical structure allows a controlled distance gate to be implemented by adding a control input to only one, rather than all, of the gates in the implementation of the original uncontrolled distance gate. Since any permutative function can be expressed as a set of cycles, and cycles can be broken down into transpositions, distance gates are capable of realizing all permutative functions.

The second part of the chapter introduced the concept of distance gate reduction, which allows the implementations of sets of distance gates meeting certain conditions to be combined. Distance gate reduction creates a circuit that realizes a set of compatible transpositions all at once with lower quantum cost compared to realizing the individual transpositions separately with one distance gate each. I presented an algorithm for realizing permutative Boolean functions that works by searching for sets of compatible transpositions and realizing them using compound distance gates, which are the result of distance gate reduction. The algorithm uses as a subroutine a realization method for single-output, incompletely specified Boolean functions. Therefore, using the presented algorithm, the task of realizing a permutative multiple-input, multiple-output Boolean function can be reduced to realizing a sequence of single-output functions. This achievement will allow future advancements in realization methods for single-output, incompletely specified Boolean functions to potentially also benefit the realization of permutative functions.

Chapter 6

Cycle-based synthesis of multiple-valued permutative quantum circuits

The majority of existing work on quantum computation and information processing assumes the use of qubits, which have two possible basis states. Quantum computation with multiple-valued qudits, which have more than two possible basis states, has been proposed [60, 61, 62, 63, 64, 65] but has not received nearly as much attention. Qudit-based systems possess several potential advantages over their qubit-based counterparts. A qudit-based system stores more information per qudit than a binary system does per qubit; consequently, the qudit-based system requires less qudits than the binary system does qubits to perform a given computation [61, 66]. The execution of a single multiple-valued gate on a qudit-based system may also require less time than an equivalent sequence of binary gates on a qubit-based system [61].

Current efforts toward physically realizing a viable quantum computing system appear to be directed mostly at binary systems. However, physical realizations of multiple-valued qudits, including the ability to manipulate three independent quantum states, have been demonstrated using approaches such as ion trap [61], optical [67], and superconducting [68] systems. A more recently proposed quantum computational framework, topological quantum computation, is in fact naturally represented by a ternary, not binary, model [69, 70]. Even if it is only possible to measure, or read out, two different states at the end of a computation, the

use of more than two states during intermediate stages of the computation can still simplify quantum gates [71].

As pointed out in the introduction to Chapter 5, diverse approaches have been proposed to the problem of realizing permutative functions with binary quantum and reversible circuits. These approaches include transformation-based, cycle-based, PPRM-based, and BDD-based methods. However, the task of synthesizing multiple-valued reversible circuits from multiple-valued specifications is poorly-studied compared to its binary counterpart. Due to the potential advantages of qudit-based quantum information processing, further investigation into the synthesis of multiple-valued reversible circuits may prove valuable to the continued development of quantum computers.

In Chapter 5, I introduced the concept of distance gates and showed how they may be implemented using CNOT and Toffoli gates. In this chapter, I will show how distance gates can be generalized to a multiple-valued setting and implemented using controlled-transposition and multiple-control transpositional gates, which are multiple-valued analogues of CNOT and Toffoli gates. As in the binary case, multiple-valued distance gates realize individual transpositions and therefore can be used to realize any permutative function by expressing it as a sequence of transpositions. This distance gate-based realization approach has the advantage of being applicable to multiple-valued systems of any radix. Multiple-valued distance gates can even be used with quantum computing systems containing arbitrary mixtures of qudits with different radices. No existing published method for the realization of permutative functions using a multiple-valued quantum computing system has this level of generality and flexibility.

6.1 Generalization of distance gates to a multiple-valued setting

The following definition generalizes the distance gates introduced in Chapter 5 to a multiple-valued setting.

Definition 20. A multiple-valued quantum gate G that acts on n qudits, x_1 through x_n, is called a *distance gate* if there exist values a_1 through a_n and b_1 through b_n with the following properties:

1. The gate G acting on the state $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$ (that is, each qudit x_i is in the state $|a_i\rangle$) produces the state $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$, and vice versa.

2. The gate G acting on any other basis state leaves that state unchanged.

3. There exists at least one i for which $a_i \neq b_i$.

The n-digit values $a_1 \ldots a_n$ and $b_1 \ldots b_n$ are called the *active values* of the distance gate.

This definition is nearly identical to Definition 14, which defines uncontrolled distance gate in the binary case, except that the gate now acts on multiple-valued qudits instead of qubits. As in the binary case, this definition can be summarized by saying that a distance gate is any gate that realizes a permutative function consisting of exactly a single transposition $(a_1 a_2 \ldots a_n \;\; b_1 b_2 \ldots b_n)$. Definition 20 intentionally does not specify the radices of the qudits x_1 through x_n. This is because the radices of the qudits are irrelevant: none of the analyses and arguments presented in this chapter rely on any assumptions about the radices of the qudits. It is even possible for the radix of each x_i to be different.

There is one notable difference between Definition 20 and the corresponding definition of an uncontrolled distance gate in the binary case, Definition 14: $a_i \neq b_i$ is only required for at least one i instead of for all i. As we will see, this modification allows both the uncontrolled

and controlled distance gates introduced in Chapter 5 to be seen as just different instances of the same fundamental structure.

Figure 6.1 illustrates a circuit that implements a distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$. The similarity with Figures 5.1 and 5.2 in the binary case is immediately clear. In this figure, the circuit contains n qudits, x_1 through x_n. The index p must be chosen so that $a_p \neq b_p$; property 3 in the definition of a distance gate ensures that at least one such p always exists. Note that for some choices of p, some of the qudits and gates depicted in the figure may not actually be present. For instance, if $p = n$, then x_p would be the bottommost qudit in the circuit; the qudits shown below x_p in the figure and the controlled-transposition gates targeting them would not be present. In addition, if $a_i = b_i$ for some $i \neq p$, then the corresponding controlled-transposition gates that target qudit x_i would disappear since their target gates would reduce to $T_{a_i a_i}$. For instance, if a_1 were equal to b_1, then the target gates of the first and last gates in Figure 6.1 would become $T_{a_1 a_1}$, which is a no-op.

The following are some noteworthy features of the circuit shown in Figure 6.1:

- It contains a single, central multiple-control transpositional gate, which is surrounded by controlled-transposition gates in a symmetrical manner.
- Every controlled-transposition gate shares the same control qudit and control value.
- This common control qudit also functions as the target qudit for the multiple-control transpositional gate.
- One of the two values affected by the target gate of the multiple-control transpositional gate is also the common control value for the controlled-transposition gates.
- For each qudit that functions as the target qudit of a controlled-transposition gate, one of the two values affected by that controlled-transposition gate's target gate is

also the control value for that same qudit when it functions as a control qudit for the multiple-control transpositional gate.

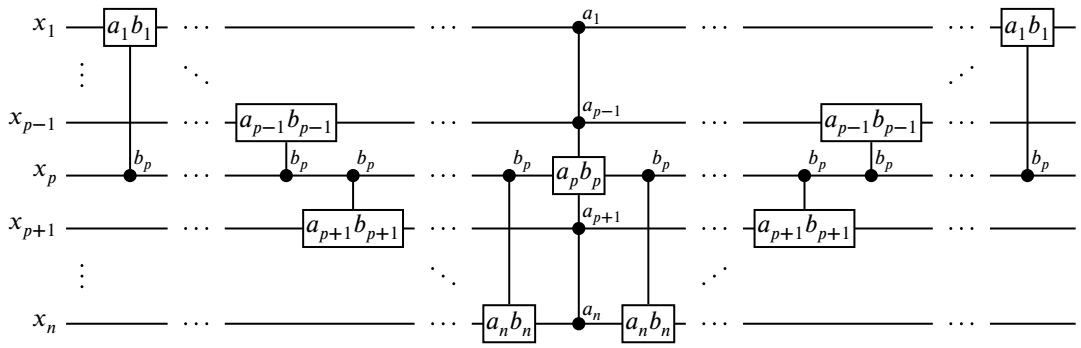


Figure 6.1: A circuit that implements a multiple-valued distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$.

To demonstrate that the circuit depicted in Figure 6.1 correctly implements a distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$, we analyze its operation as follows. All of the controlled-transposition on the left side of Figure 6.1 are active if and only if the initial state of qudit x_p is $|b_p\rangle$; therefore, if the initial states of the qudits x_1 through x_n are $|a_1\rangle$ through $|a_n\rangle$, respectively, then they will remain unaltered by the controlled-transposition gates on the left side. The central multiple-control transpositional gate will then induce a transition of the qudit x_p from the state $|a_p\rangle$ to the state $|b_p\rangle$. Following the controlled-transposition gates on the right side, the final states of the qudits are then $|b_1\rangle$ through $|b_p\rangle$. A similar analysis demonstrates that when the initial state is $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$, the final state will be $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$.

If the initial state is any basis state other than the two already considered, the circuit from Figure 6.1 will leave it unchanged. To see this, first observe that if the action of the central multiple-control transpositional gate does not alter the state of the circuit just prior to that gate, then the circuit as a whole will leave the initial state (prior to the circuit) unchanged,

because the symmetrical arrangement of controlled-transposition gates will cause them to cancel themselves out. Next, note that the state of the circuit just prior to the multiple-control transpositional gate will be unchanged by that gate if that state is not one of

$$|a_1\rangle \otimes \cdots \otimes |a_{p-1}\rangle \otimes |a_p\rangle \otimes |a_{p+1}\rangle \otimes \cdots \otimes |a_n\rangle \tag{6.1}$$

or

$$|a_1\rangle \otimes \cdots \otimes |a_{p-1}\rangle \otimes |b_p\rangle \otimes |a_{p+1}\rangle \otimes \cdots \otimes |a_n\rangle. \tag{6.2}$$

Since the circuit is reversible, both of these cases correspond to exactly one initial state: the state just prior to the Toffoli gate will be as given in (6.1) if and only if the initial state is $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$, and it will be as given in (6.2) if and only if the initial state is $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$. These two initial states are precisely the ones that we considered previously. It follows that for any other initial state, the state just prior to the multiple-control transpositional gate will be unchanged following that gate, and therefore the circuit as a whole will leave the initial state unchanged.

Observe that the preceding argument applies even if $a_i = b_i$ for one or more i, as long as $a_p \neq b_p$. If so, then, as previously observed, the corresponding controlled-transposition gates in Figure 6.1 become no-ops and can be removed. This does not affect the validity of the previous reasoning, since it was not assumed anywhere that a_i and b_i must be distinct for any i other than p.

To formalize the preceding discussion, we introduce the following notation for representing controlled-transposition and multiple-control transpositional gates. Every such gate

will be denoted by an expression of the form

$$CT_{ab}({x_1}^{c_1}, {x_2}^{c_2}, ..., {x_{n-1}}^{c_{n-1}}; x_n), \tag{6.3}$$

where:

1. "*CT*" stands for "controlled transposition";
2. x_1 through x_{n-1} are the gate's control qudits;
3. c_1 through c_{n-1} are its control values, with c_i being the control value associated with x_i;
4. x_n is its target qudit; and
5. a and b are the labels of the states exchanged by the target gate, *i.e.*, the target gate is T_{ab}.

Observe that a controlled-transposition gate is represented by a special case of this notation with only one control qudit,

$$CT_{ab}({x_c}^{c}; x_t), \tag{6.4}$$

where x_c and x_t are the control and target qudits (respectively) of the controlled-transposition gate, c is its control value, and T_{ab} is its target gate.

Using the notation of (6.3) and (6.4), we may then represent the circuit shown in Fig-

ure 6.1 as the sequence of gates

$$\left(\prod_{\substack{1 \le i \le m \\ i \neq p}} CT_{a_i b_i}(x_p^{b_p}; x_i) \right) \cdot CT_{a_p b_p}({x_1}^{a_1}, \ldots, {x_{p-1}}^{a_{p-1}}, {x_{p+1}}^{a_{p+1}}, \ldots, {x_n}^{a_n}; x_p) \cdot \left(\prod_{\substack{1 \le i \le m \\ i \neq p}} CT_{a_i b_i}({x_p}^{b_p}; x_i) \right), \tag{6.5}$$

where the products denote series concatenation of gates.

The previous analysis of the circuit from Figure 6.1 is now summarized by the following proposition.

Proposition 21. *The sequence of gates given by* (6.5) *implements a distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$; that is, this sequence of gates acting on the states $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$ and $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$ produces the states $|b_1\rangle \otimes \cdots \otimes |b_n\rangle$ and $|a_1\rangle \otimes \cdots \otimes |a_n\rangle$, respectively, and the same sequence acting on any other basis state leaves that state unchanged.*

We are thus able to implement a distance gate with any desired pair of active values by using the circuit depicted in Figure 6.1 and represented by (6.5). As this is the key result of this chapter, I restate it in a manner that explicitly and constructively shows how, given a desired pair of active values, one can generate a circuit that implements a distance gate with those active values.

Corollary 22. *A distance gate with active values $a_1 \cdots a_n$ and $b_1 \cdots b_n$ can be implemented*

by the circuit specified in (6.5) *or by the circuit*

$$\left(\prod_{\substack{1\le i\le m\\ i\neq p}} CT_{a_i b_i}(x_p^{\,a_p}; x_i)\right) \cdot CT_{a_p b_p}(x_1{}^{b_1}, \ldots, x_{p-1}{}^{b_{p-1}}, x_{p+1}{}^{b_{p+1}}, \ldots, x_n{}^{b_n}; x_p) \cdot \left(\prod_{\substack{1\le i\le m\\ i\neq p}} CT_{a_i b_i}(x_p{}^{a_p}; x_i)\right), \tag{6.6}$$

where p is any index in the range $1 \le p \le n$ *such that* $a_p \neq b_p$.

The difference between (6.6) and (6.5) is that an a_i appears in (6.6) wherever a b_i appears in (6.5), and vice versa. Corollary 22 reflects the fact that, just as in the binary case, every distance gate has two implementations for each possible value of p because the active values may be specified in either order. Figure 6.2 shows the alternative implementation described by (6.6).

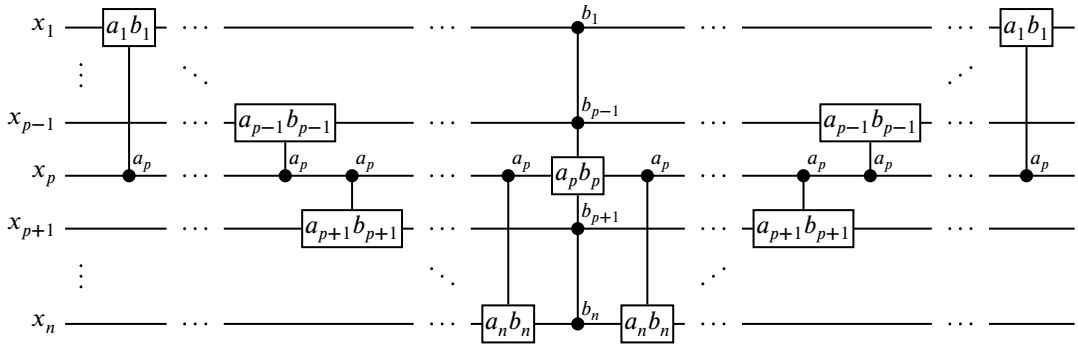


Figure 6.2: Alternative implementation of a distance gate corresponding to (6.6), obtained by replacing every a_i with a b_i and vice versa in Figure 6.1.

Analogously to the binary case, when implementing a distance gate using the circuit from Figure 6.1 or 6.2 to implement a distance gate, we also have the freedom to choose

any value of p for which $a_p \neq b_p$. I will adopt the same terminology used in Chapter 5 and call x_p the *pivot qudit*. Given two active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$, define the *Hamming distance* between them to be the number of indices i such that $a_i \neq b_i$; this is analogous to the definition of Hamming distance for binary values. If the Hamming distance between $a_1 \ldots a_n$ and $b_1 \ldots b_n$ is d, then one has d choices for the pivot qudit when implementing a distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$. There are therefore $2d$ possible implementations of this distance gate, when $d \geq 2$. When $d = 1$, the circuits shown in Figures 6.1 and 6.2 become identical and reduce to just a single multiple-control transpositional gate, so in that case there is only one possible implementation of the distance gate with active values $a_1 \ldots a_n$ and $b_1 \ldots b_n$.

6.2 Examples of multiple-valued distance gates and their operation

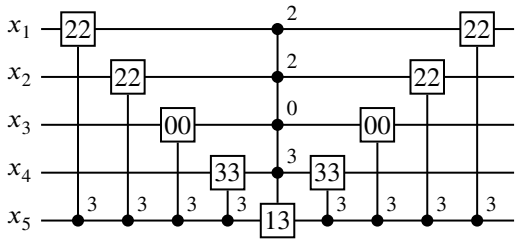
I now provide several examples to illustrate the theory introduced in Section 6.1. Figure 6.3 provides several concrete examples of distance gates implemented using the circuit structure from Figure 6.1. These examples demonstrate the appearance of the resulting circuit for a variety of Hamming distances between the active values, and for different choices of pivot qudit. In Figure 6.3a, the Hamming distance between the active values $a_1 \ldots a_5 = 22031$ and $b_1 \ldots b_5 = 22033$ is 1, so there is only one p such that $a_p \neq b_p$, namely $p = 5$. Therefore, the pivot qudit must be x_5. Since $a_i = b_i$ for $i = 1, 2, 3, 4$, the controlled-transposition gates in Figure 6.3a have trivial target gates that are no-ops and can therefore be removed, giving the circuit shown in Figure 6.3b. We see that the implementation of a distance gate reduces to just a single multiple-control transpositional gate when the Hamming distance between the active values is one. Figures 6.3c and 6.3d are analogous to 6.3a and 6.3b, but for a distance gate where the Hamming distance between the active values is 3. In this case, some but not

all of the controlled-transposition gates turn out to be trivial. Finally, Figure 6.3e shows the implementation of a distance gate with a Hamming distance 5 between its active values, the maximum possible for 5-qudit values. In this case, all of the controlled-transposition gates from Figure 6.1 are nontrivial and remain in the circuit. Figures 6.3f and 6.3g demonstrate, respectively, that the same distance gate can also be implemented with the active values swapped[1] and with a different choice of pivot qudit, in accordance with Corollary 22.

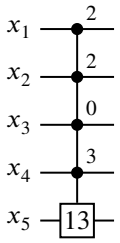
Figure 6.3 also helps demonstrate how the implementation of a distance gate is completely independent of the radices of the underlying qudits. In Figure 6.3b, the radix of the first qudit may be any integer greater than or equal to 3, since a control value of 2 requires a minimum radix of 3. Similarly, x_2 through x_5 have minimum radices of 3, 2, 4, and 4, respectively, and may otherwise be of any radix. Furthermore, regardless of the radices of the underlying qudits, the circuit depicted in Figure 6.3b realizes the transposition (22031 22033); note that this transposition itself specifies a permutative function in a radix-ignorant manner. Similarly, the circuit shown in Figure 6.3d always realizes the transposition (33014 30020) regardless of the radices of the qudits. There is also nothing to stop the qudits from having different radices; as long as the underlying controlled-transposition and multiple-control transpositional gates can be physically realized for qudits of the appropriate radices, distance gate implementations such as the ones in Figure 6.3 can be used.

The circuit from Figure 6.3d provides excellent opportunity to illustrate the proof of Theorem 21 by example. Conceptually, the circuit may be divided into three groups of gates: the set of controlled-transposition gates on the left, the central multiple-control transpositional gate, and the set of controlled-transposition gates on the right. For brevity, I will refer to the controlled-transposition gates on the left and right as the left and right

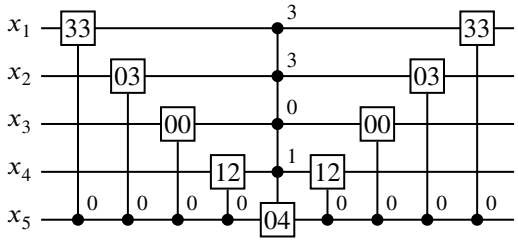
[1] Or equivalently, using the circuit structure from Figure 6.2 instead of Figure 6.1.



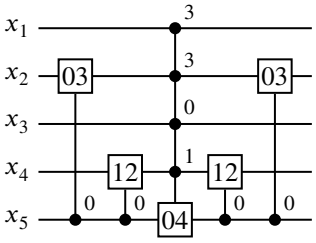
(a) A distance gate with active values 22031 and 22033 implemented according to Figure 6.1, showing controlled-transposition gates with trivial target gates.



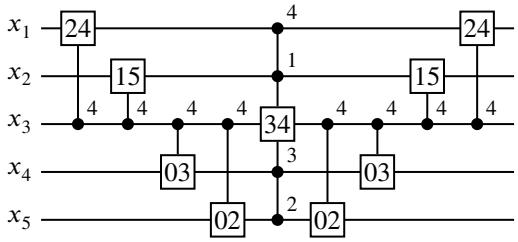
(b) Result of removing the trivial controlled-transposition gates in (a).



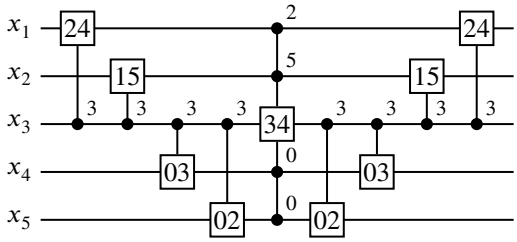
(c) The same as (a) but for active values 33014 and 30020.



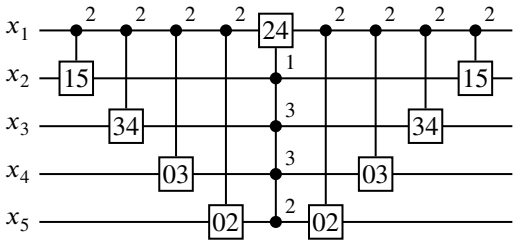
(d) Result of removing the trivial controlled-transposition gates in (c).



(e) Active values 41332 and 25400 with pivot qudit x_3.



(f) The same as (e), but implemented with active values swapped.



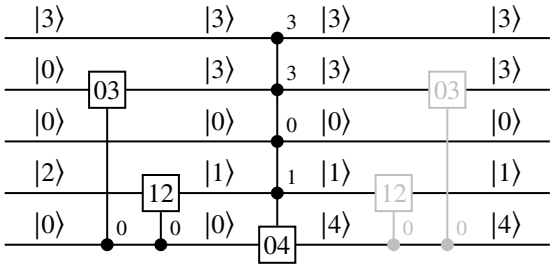
(g) The same as (e), but with pivot qudit x_1.

Figure 6.3: Assorted implementations of multiple-valued distance gates.

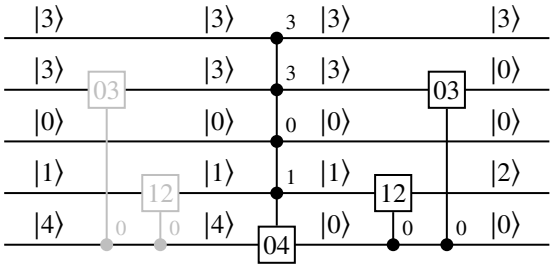
groups, respectively. In informal terms, the central multiple-control transpositional gate acts as a "filter" which selectively allows the circuit to alter the initial state. Without this multiple-control transpositional gate, the effects of the left and right groups would always cancel so as to have no overall effect on the state. The action of the multiple-control transpositional gate can prevent this cancellation if it alters the state of the bottommost qudit in Figure 6.3d. If the bottommost qudit's state changes from $|0\rangle$ to $|4\rangle$, then the left group is active but the right group is not; conversely, if the bottommost qudit's state changes from $|4\rangle$ to $|0\rangle$, then the right group is active but the left group is not.

The preceding discussion implies that the circuit from Figure 6.3d can operate in three distinct modes, which are illustrated in Figure 6.4. For each mode of operation, the inactive portion of the circuit is drawn in gray while the active portion is drawn normally. In the first mode, as shown in Figure 6.4a, the bottommost qudit's state changes from $|0\rangle$ to $|4\rangle$ across the multiple-control transpositional gate, and hence the left group is active but the right group is not. In the second mode, as shown in Figure 6.4b, the bottommost qudit's state changes from $|4\rangle$ to $|0\rangle$; the right group is active but the left group is not. In the third mode, as shown in Figure 6.4c, the bottommost qudit's state does not change at all because the multiple-control transpositional gate is inactive, and therefore the effects of left and right groups cancel. In this mode, if the initial state is $|x_1 x_2 x_3 x_4 x_5\rangle$ and the state following the left stage is $|x'_1 x'_2 x'_3 x'_4 x'_5\rangle$, then the state following the multiple-control transpositional gate is still $|x'_1 x'_2 x'_3 x'_4 x'_5\rangle$ and the right stage reverts the state back to $|x_1 x_2 x_3 x_4 x_5\rangle$. Of these three modes, the first and second correspond to the two active values of the distance gate, while the third corresponds to any other value, which the distance gate leaves unchanged.

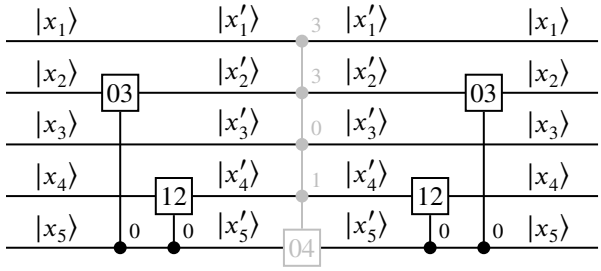
Figure 6.5 offers an alternative visualization of the same circuit's operation using the state space. The state space is the set of all possible basis states of the qudits on which the



(a) Left group active, right group inactive.



(b) Right group active, left group inactive.



(c) Central multiple-control transpositional gate inactive.

Figure 6.4: Modes of operation of the circuit shown in Figure 6.3d.

circuit operates. Although the circuit itself does not depend on the qudits having a specific radix, the state space does, since the number of possible basis states increases with the radices of the qudits. For the sake of illustration, suppose that every qudit in the circuit from Figure 6.3d has radix 5, which is the minimum radix for the bottommost qudit. Then, since there are five qudits in the circuit, the state space consists of 5^5 states running from $|00000\rangle$ to $|44444\rangle$. In Figure 6.5, the state space is plotted on the vertical axis against time on the horizontal axis, showing how several choices of initial state evolve as they pass through the circuit. Once again, we see that the effects of the left and right stages cancel unless the multiple-control transpositional gate is active, which only occurs for the two active values as shown by the bolded lines.

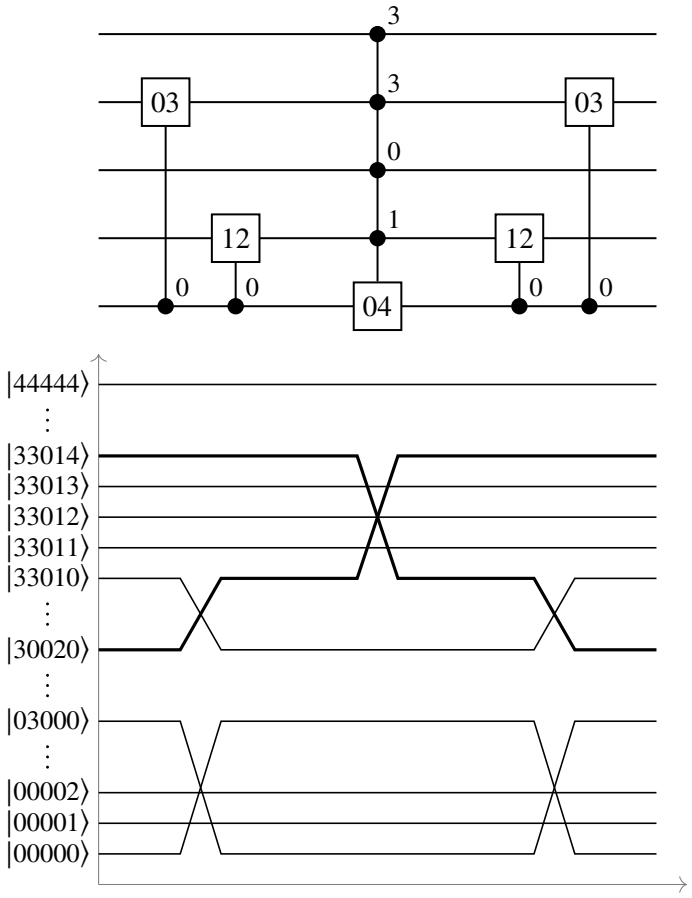


Figure 6.5: State-space visualization of the operation of circuit from Figure 6.3d.

6.3 Binary distance gates as a special case of multiple-valued distance gates

Both binary uncontrolled and controlled distance gates can be seen to be special cases of multiple-valued distance gates where the radix of all qudits happens to be 2. Specifically, comparing Definition 14 with Definition 20, it is apparent that an uncontrolled binary distance gate is just a multiple-valued distance gate acting on qudits of radix 2 (*i.e.*, qubits) in which the active values differ in every bit. Similarly, from Proposition 16, a controlled binary distance gate also satisfies Definition 20, but with active values that agree in at least one bit.

The implementations of binary distance gates (both uncontrolled and controlled) can also be derived from the implementations of multiple-valued distance gates shown in Figures 5.1 and 5.2. In particular, if we apply Proposition 15 combined with Definition 14 to Figure 5.2, then we see that $c_p = b_p$ and $c_i = a_i$ for $i \neq p$, where $a_1 \ldots a_n$ and $b_1 \ldots b_n$ are the active values of the distance gate realized by an instance of Figure 5.2. Keeping in mind that a CNOT gate is the same as a controlled-T_{01} gate applied to qubits, this mapping makes Figure 5.2 identical to Figure 6.1 when the qudits of the latter have radix 2. Similarly, Figure 5.4 is seen to be identical in structure[2] to Figure 6.1 when $a_i = b_i$ for one or more i in the latter, since in that case the corresponding controlled-transposition gates disappear, like in the examples of Figure 6.3. For instance, letting $n = 5$, $a_1a_2a_3a_4a_5 = 01101$, and $b_1b_2b_3b_4b_5 = 11000$ in Figure 6.1 produces the circuit of Figure 6.6 after trivial controlled-transposition gates have been removed. This circuit matches the one from Figure 5.6d, which does indeed realize the transposition (01101 11000).

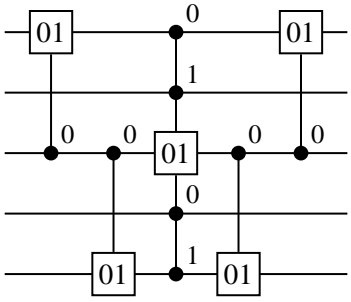


Figure 6.6: The binary controlled distance gate from Figure 5.6d obtained as a multiple-valued controlled distance gate where all qudits have radix 2.

As a result, both binary and multiple-valued distance gates can be unified under the single concept of distance gates, whose applicability to a quantum computing system is totally insensitive to the radices of the qubits and/or qudits used by that system. In particular,

[2] I say "identical in structure" because the naming and indexing of qubits differs between the two figures, although this of course has no bearing on the circuits' behavior.

distance gates can be used with purely binary systems, purely multiple-valued systems with the same radix for all qudits, and mixed systems combining qubits and qudits of any radices.

6.4 Distance-gate-based realization of permutative functions using multiple-valued circuits

In order to realize a reversible function using distance gates, the function must first be expressed in terms of cycles. This is done in the same manner already described in Chapter 5, Section 5.1. Once the function to be realized is in the form of a collection of cycles, each cycles can then be decomposed into a product of transpositions, and the transpositions realized using distance gates.

As an example, let us consider how to realize the function represented by the permutation (121 131 032)(003 133) using a quarternary circuit. The 3-cycle (121 131 032) is equivalent to the sequence of transpositions (121 131)(121 032). Each of these transpositions can be realized using a distance gate. To implement the distance gates using controlled-transposition and multiple-control transpositional gates, we must choose a pivot qudit to be used for each transposition. One possible choice is to use the second qudit as the pivot for all three distance gate, which produces the circuit shown in Figure 6.7.

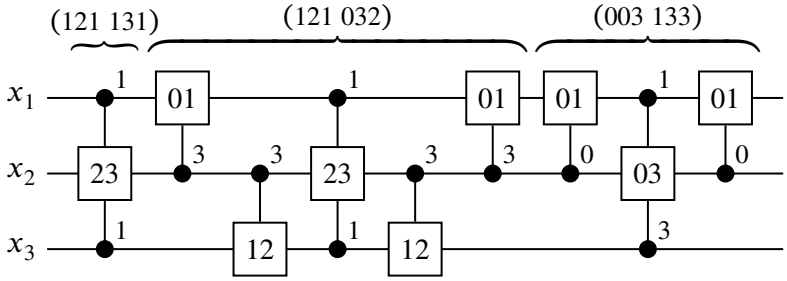
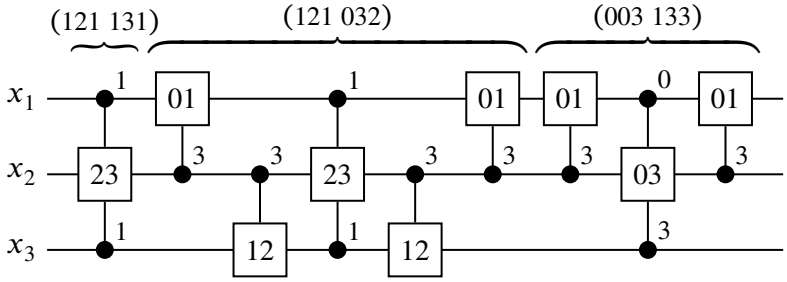
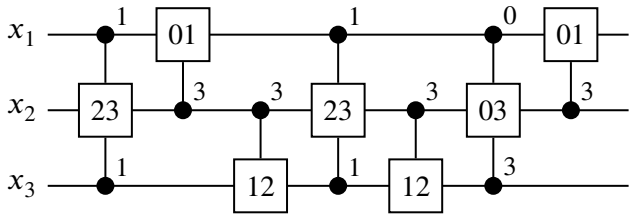


Figure 6.7: Distance gate-based realization of the reversible function represented by the permutation (121 131 032)(003 133).

As previously noted in the discussion following Corollary 22 and demonstrated by Figures 6.3e, 6.3f, and 6.3g, a distance gate has more than one possible implementation if the Hamming distance between its active values is greater than 1. Therefore, the circuit shown Figure 6.7 is not the only possible distance-gate-based realization of the permutation (121 131 032)(003 133). For instance, we may choose to implement the third distance gate, whose active values are $a_1a_2a_3 = 003$ and $b_1b_2b_3 = 133$, with the a_i's and b_i's swapped, in other words switching from the implementation of Figure 6.2 to that of Figure 6.1. Figure 6.8 shows the realization obtained in this fashion. We can see that the final distance gate in Figure 6.8a is implemented with different control values compared to the circuit from Figure 6.7. This alternative realization has the advantage that it contains a pair of adjacent and identical controlled-transposition gates, which can therefore be canceled, resulting in the circuit shown in Figure 6.8b.



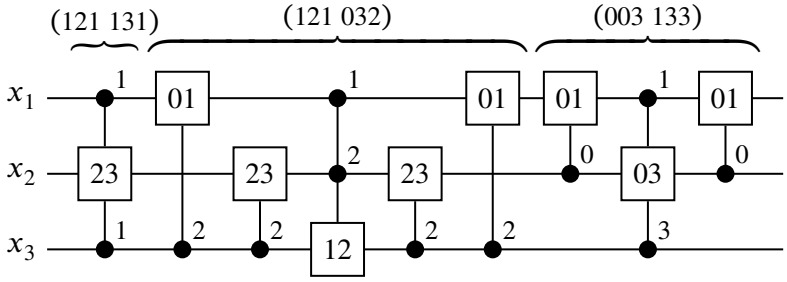
(a) Circuit with the active values of the last distance gate swapped relative to Figure 6.7.



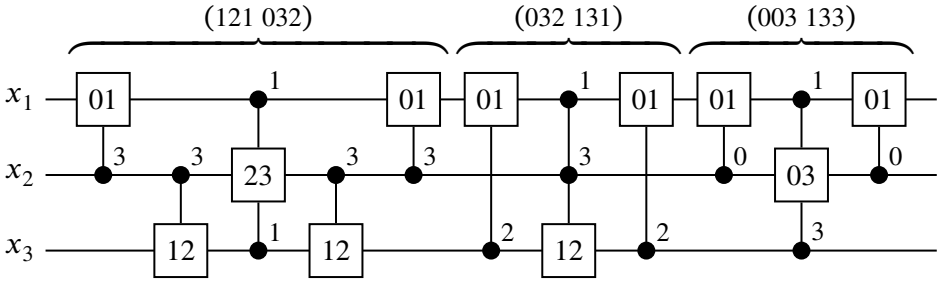
(b) Simplified version of Figure 6.8a obtained by canceling two controlled-transposition gates.

Figure 6.8: An alternative realization of (121 131 032)(003 133).

Yet other realizations are possible as well. We may choose to implement one of the distance gates using a different pivot bit, as shown in Figure 6.9a, or we may use a different breakdown of the 3-cycle (121 131 032) into a sequence of transpositions, as shown in Figure 6.9b. In the latter case, the circuit actually becomes more costly—requiring a greater number of controlled-transposition gates and the same number of multiple-control transpositional gates—than the original realization from Figure 6.7.



(a) Result of using a different pivot bit to implement the second distance gate.



(b) Result of breaking down the 3-cycle (121 131 032) as (121 032)(032 131).

Figure 6.9: Two more alternative realizations of (121 131 032)(003 133).

Collectively, the realizations shown in Figures 6.7, 6.8, and 6.9 illustrate that many different distance gate-based realizations of a given function are possible. For each cycle with three or more elements, one may choose between many possible decompositions of the cycle into a sequence of transpositions. Similarly, for each transposition, one may choose the pivot bit and order of active values (again as per the discussion following Corollary 22)

to be used in implementing the corresponding distance gate. Furthermore, depending on these choices, additional simplifications may become possible between adjacent distance gates in the final circuit. We observed an example of such a simplification in Figure 6.8. Thus, the choices made in realizing a particular permutation using distance gates can have a significant impact on the optimality of the resulting circuit.

6.5 Conclusion

In this chapter, I generalized the distance gates first introduced in Chapter 5 to the setting of multiple-valued quantum circuits and showed how they can be implemented using controlled-transposition and multiple-control transpositional gates. I showed that in this setting, the implementation of a binary distance gate is simply a special case of the implementation of a multiple-valued distance gate where all of the qudits have radix 2 and are therefore just qubits. In fact, the method used to implement distance gates is completely insensitive to the radix of the qudits being used, allowing distance gates to be easily implemented using circuits containing qudits of any radix or mixture of radices. Since distance gates realize individual transpositions, any permutative function can then be realized by representing it in terms of cycles, breaking the cycles down into transpositions, and realizing the transpositions using distance gates.

One notable issue that was not considered in this chapter is the question of how the controlled-transposition and multiple-control transpositional gates themselves are implemented. This issue was previously addressed by Muthukrishnan and Stroud [61], whose $\mathbf{\Gamma}_n$ gates are physically realizable using ion traps and are essentially equivalent to the controlled-transposition and multiple-control transpositional gates used here. The question of implementing multiple-control transpositional gates using two-qudit interactions has also been

addressed more recently by others [72, 73, 74, 75]. Although Muthukrishnan and Stroud originally only demonstrated the implementation of their proposed gates using ion traps, it is reasonable to expect that they should also be physically realizable in any other system of practical interest, since such gates or other gates with similar function are required for universality. This assumption is not unprecedented; in the case of topological quantum computation, Bocharov *et al.* [76] make use of a controlled-T_{01} gate, from which other controlled-transposition gates are also easily obtained. In conjunction with these known results, distance gates can function as an intermediate step in the realization of reversible functions using physical multiple-valued quantum systems.

Chapter 7

A quantum algorithm for automata encoding

Note: This chapter was previously published as the following journal article:

E. Tsai and M. A. Perkowski, "A quantum algorithm for automata encoding," *Facta Universitatis, Series: Electronics and Energetics*, vol. 33, no. 2, pp. 169–215, 2020.

As stated in Chapter 1, one of the objectives of this dissertation is to demonstrate in detail the design of quantum oracles that allow Grover's algorithm to be applied to problems of practical interest. In this chapter, I consider one such problem, namely that of state encoding for finite state machines (FSMs). A well-known problem in digital logic design, which has been recognized for over 50 years [77, 78] and today remains important and relevant to the design of virtually all very-large-scale integrated (VLSI) circuits and systems including microprocessors, state encoding (a.k.a. state assignment) is the assignment of binary states in a digital logic circuit to represent the internal states of an FSM. Distinct encodings for the same FSM produce distinct implementations in digital logic, which may differ considerably in complexity [77]. Usually, one wishes to find a state encoding which is minimal with respect to some metric, *e.g.*, power [79, 80] or silicon area [81, 82] of the resulting digital circuit.

Our goal is to find the exact minimum (with respect to one particular metric) solution for state and input encoding of finite state machines. So far, only one previous work [83]

has attempted to find exact minimum solutions to this problem. Furthermore, the methods in [83] find solutions for only state and not input encoding. There is currently no published result that finds the exact minimum solution for concurrent state and input assignment. The methods in [83] rely on finding prime implicants and solving a covering problem on the set of all prime implicants. This would be difficult to implement using Grover's algorithm on a quantum computer because at least one qubit would be required for each prime implicant and the total number of prime implicants can be extremely large. Therefore, we do not attempt to directly adapt the approach presented in [83] for a quantum computer. Instead, we use a simplified cost metric from [77, 78], in which the cost of an encoding is defined as the total number of dependencies of the next-state functions on current state and input variables. The use of this metric makes it easier to construct the quantum circuits necessary to use Grover's algorithm to search for encodings. Since Grover's algorithm effectively performs an exhaustive search, our method is always able to find an encoding with exact minimum cost. The techniques that we use to adapt Grover's algorithm for the purpose of encoding finite state machines may prove useful for other purposes as well.

7.1 Finite state machines and state encodings

7.1.1 Review of finite state machines

We assume that the reader is familiar with the concept of finite state machines (FSMs) and how they are realized using digital logic, as well as with the state encoding (a.k.a. state assignment or secondary state assignment) problem. For the sake of self-containment we briefly review these subjects here. An FSM consists of a set of *internal states* (call it **S**) together with sets of *inputs* and *outputs* (call them **I** and **O**, respectively); at all times, it maintains a single internal state which is an element of **S**, is presented with an input value

which is an element of **I**, and produces an output value which is an element of **O**. The machine's operation is idealized as a discrete-time process; with each successive unit of time, it updates its internal state and output in a deterministic fashion based on its current internal state and the input value that is being presented. Thus, the internal state of the machine at time $t + 1$ is a function of the internal state at time t and the input at time t:

$$\mathrm{S}_{t+1} = \delta(\mathrm{S}_t, \mathrm{I}_t), \tag{7.1}$$

where $\mathrm{S}_t \in \mathbf{S}$ is the internal state at time t, $\mathrm{I}_t \in \mathbf{I}$ is the input value at time t, and $\mathrm{S}_{t+1} \in \mathbf{S}$ is the internal state at time $t + 1$. We refer to the function δ as the *transition function* for the FSM. This function is also commonly called the *excitation function* or *next-state function*.

The output of an FSM can either directly depend on only its internal state, or it can directly depend on both the internal state and the input. The former scenario corresponds to a so-called Moore machine [84, 85] whereas the latter corresponds to a so-called Mealy machine [86, 85]. Here, as will be discussed in more detail below, we only consider the problem of encoding internal states; thus, the type of the machine is irrelevant and our work is equally applicable to both.

From now on, for the sake of brevity, we will simply use "states" to refer to the internal states of an FSM. The phrase "internal states" avoids confusion with other objects also referred to as "states", in particular the input and output values which are sometimes called input and output states, respectively. We will always use the phrases "input (output)" or "input (output) values" here, so that there is no risk of confusion in using simply "states" to refer to internal states.

FSMs are commonly implemented using digital logic circuits consisting of an array of flip-flops, which stores the current state, together with combinational logic (often referred

to as the next-state logic), which computes the next state from the current state and current input. To design this combinational logic, one must select a *state encoding*, a mapping which associates each state of the FSM with a state of the flip-flop array. More precisely, since the state of a flip-flop array is simply a binary string, a state encoding is a function $c_{\mathbf{S}} : \mathbf{S} \to \{0,1\}^n$ that maps each state of the FSM to a vector of flip-flop states that represents the FSM state in a digital logic circuit. Here, n denotes the number of flip-flops in the circuit.

Similarly, in order to implement an FSM with a digital logic circuit, one must also select an *input encoding*, which maps each input value of the FSM to an array of Boolean values, where each Boolean value represents the value supplied on an input wire to the digital logic circuit. In other words, an input encoding is a function $c_I : \mathbf{I} \to \{0,1\}^m$ where m is the number of input wires.

Finally, in addition to the state and input encoding, one must select an *output encoding* to fully implement an FSM with digital logic. However, we do not consider the problem of encoding outputs in this paper and consequently, we will ignore the outputs of an FSM from now on. We will use "encoding" without further qualification to mean the combination of a state and input encoding. We may also use the phrase "encoding of [a state or input value]" to mean the combination of state or input variable values corresponding to that state or input value under a given encoding. In other words, given a state encoding as a function $c_{\mathbf{S}}$ as described above, the encoding of a state S is simply $c_{\mathbf{S}}(\mathrm{S})$; the situation is analogous for inputs. Finally, we also use "encoding" as a verb to mean the process of selecting or generating an encoding. Thus, we describe the main objective of this paper as solving an FSM encoding problem.

From now on, when considering a digital logic circuit implementation of an FSM, we will follow the common convention of using Q_i to denote the state of the i-th flip-flop at

a given point in time. We will also use x_j to represent the value on the j-th input wire at a given point in time. We refer to the variables Q_1 through Q_n as *state variables* and the variables x_1 through x_m as *input variables*. Figure 7.1 illustrates this notation in the context of a digital logic circuit that implements an FSM.

We distinguish carefully between input *values*, which are the symbolic inputs of an FSM and elements of the set **I**, and input *variables*, which are the Boolean variables x_1 through x_m used in a digital logic circuit to represent input values. Similarly, we also distinguish between states and state *variables*—states are the symbolic internal states of an FSM and elements of the set **S**, while state variables are the variables Q_1 through Q_n that correspond to the states of individual flip-flops and together represent the symbolic state. To help avoid confusion, we will always follow a consistent notational convention where, in addition to using Q_i and x_j for states and input values respectively, we also use S or S_1, S_2, etc. to represent states and I, I_1, I_2, etc. to represent input values. In the context of an FSM, the word "input" without further qualification will always refer to an input value and not an input variable.

A state encoding may be bijective, that is, each flip-flop array state represents exactly one FSM state, or it may not. Non-bijective encodings might arise (for instance) if the number of possible flip-flop array states is greater than the number of FSM states, in which case some of the possible flip-flop states will not represent any FSM state at all. Similarly, input encodings may also be bijective, or not. For a bijective input encoding, each possible combination of assignments to input variables corresponds to exactly one input value. We say that an encoding (the combination of both state and input encodings) is bijective if both the state and input encodings are bijective.

If an encoding is bijective, then any combination of assignments to the variables Q_1

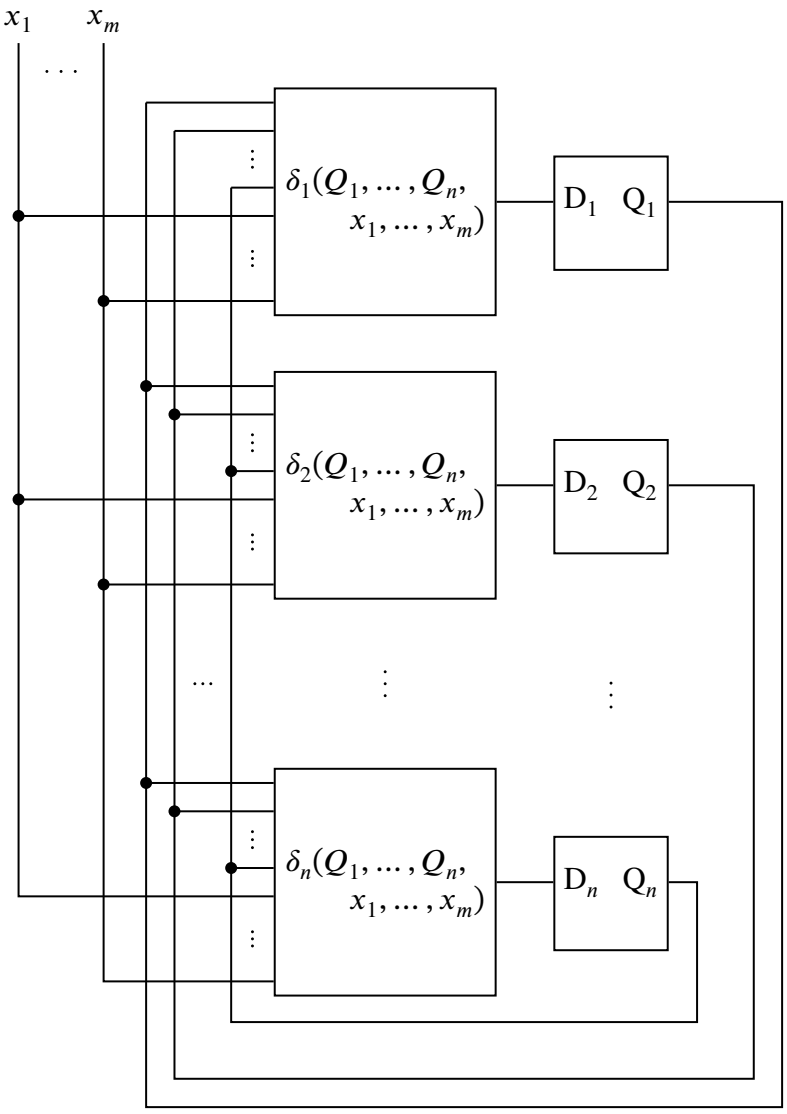


Figure 7.1: General structure of an FSM implemented as a digital logic circuit; output logic not shown.

through Q_n and x_1 through x_m corresponds to a unique state and input value of the FSM. Therefore, the next state is also uniquely determined by the transition function δ. In this case, we define a collection of *encoded transition functions* δ_1 through δ_n, where δ_i represents the next state of the i-th flip-flop in terms of the variables Q_1 through Q_n and x_1 through x_m. In other words, the values of Q_1 through Q_n and x_1 through x_m uniquely determine a state S and input value I. If these are the current state of and input to the FSM, then the next state of the FSM will be $\delta(\mathrm{S}, \mathrm{I})$ and the encoding of this next state is $c_{\mathbf{S}}(\delta(\mathrm{S}, \mathrm{I}))$ where $c_{\mathbf{S}}$ is the functional representation of a state encoding as previously described. We then define $\delta_i(Q_1, \ldots, Q_n, x_1, \ldots, x_m)$ to be the i-th component of $c_{\mathbf{S}}(\delta(\mathrm{S}, \mathrm{I}))$.

Encoded transition functions represent the computations to be performed by the next-state logic in a digital logic circuit implementation of an FSM. In other words, given a particular state encoding for an FSM, each encoded transition function gives the next state of a single flip-flop in terms of the current states of all flip-flops and the current input. Thus, digital logic design for an FSM involves realizing the encoded transition functions as digital logic circuits. Figure 7.1 graphically demonstrates this relationship between encoded transition functions and the digital logic implementation of an FSM. In the remainder of this paper, we will concentrate on evaluating the cost of realizing encoded transition functions and how this cost can be minimized.

If a state encoding is not bijective, one can still define a set of encoded transition functions using the same concept—each encoded transition function represents the next state of a single flip-flop in terms of the current states of all flip-flops and the current values of all input variables. However, the encoded transition "functions" defined in this way are no longer functions in the mathematical sense; they are relations instead. If the flip-flops' current state does not correspond to any FSM state or the input variables' values do not correspond to any

FSM input value, then the next state is indeterminate (a.k.a. a "don't-care"). In practice, digital logic design for FSMs commonly involves non-bijective encodings. Nevertheless, for reasons to be discussed later, we will only consider bijective encodings, for which all encoded transition functions are actual functions in the mathematical sense. Observe that this restriction implies that we are only considering FSMs where the number of states is a power of two, since no bijective encodings exist otherwise. It also implies the assumption that the machine is state-minimized, meaning that there are no equivalent states in the machine, so that no two distinct states may have the same encoding.

It is common to use the notation Q_i^+ to represent the next state of the i-th flip-flop, where Q_1 through Q_n represent the current states of all flip-flops. In other words, $Q_i^+ = \delta_i(Q_1, \ldots, Q_n, x_1, \ldots, x_m)$ for all i. This means that Q_i^+ is simply a more compact notation for the function δ_i that can be used when it is not necessary to explicitly show that δ_i is a function of Q_1 through Q_n and x_1 through x_m. From now on, we will use both the Q_i^+ and δ_i notations interchangeably, with the choice of notation being simply a matter of convenience.

7.1.2 Metric for evaluating cost of state encodings

The reader may observe that, by considering the implementation of FSMs using digital logic circuits, we have created a distinction between a symbolic FSM itself and its implementation using flip-flops and combinational logic. The former is simply an abstract mathematical concept, while the latter is a physical realization of that abstract concept. From now on, when the meaning is clear from the context, we will use "finite state machine" to refer to both an FSM in the abstract conceptual sense and physical implementations of that FSM. When the need to avoid confusion arises, we will use "FSM specification" to refer specifically to the abstract mathematical concept of an FSM.

There always exist many possible implementations of any single FSM specification, since an implementation of an FSM is always associated with some encoding and there are many possible encodings for any given set of states or inputs. The differences between implementations obtained with different encodings are of great interest to the digital logic designer. In particular, the combinational circuit complexity (as measured, for instance, by the number of logic gates or the maximum delay) of implementations may greatly vary for different encodings. Consider the FSM whose transition function is as given in Figure 7.2a. Figure 7.2b depicts a possible encoding for the FSM where the four possible two-bit strings are simply allocated in the usual base-two counting order (00 first, then 01, 10, and 11). Figure 7.2c then shows the resulting encoded transition functions. These functions may be represented by the following logical expressions:

$$Q_1^+ = (Q_1 \wedge x_2) \vee (Q_1 \wedge \neg Q_2 \wedge x_1) \vee (\neg Q_1 \wedge Q_2 \wedge x_1) \\ \vee (Q_2 \wedge \neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2), \tag{7.2}$$

$$Q_2^+ = (Q_1 \wedge Q_2 \wedge \neg x_1) \vee (Q_1 \wedge Q_2 \wedge \neg x_2) \vee (\neg Q_1 \wedge \neg Q_2 \wedge \neg x_1) \\ \vee(\neg Q_1 \wedge \neg Q_2 \wedge \neg x_2) \vee (\neg x_1 \wedge \neg x_2), \tag{7.3}$$

which show that both Q_1^+ and Q_2^+ depend on all four state/input variables (Q_1, Q_2, x_1, and x_2). In comparison, if the encoding shown in Figure 7.2d is used instead, then the encoded

transition functions are as shown in Figure 7.2e and may be represented by the expressions

$$Q_1^+ = Q_2 \vee \neg x_1, \tag{7.4}$$

$$Q_2^+ = (Q_1 \wedge \neg x_1) \vee (Q_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2), \tag{7.5}$$

where we observe that Q_1^+ depends on only two variables (Q_2 and x_1) and Q_2^+ depends on three (Q_1, x_1, and x_2).

	I_1	I_2	I_3	I_4
S_1	S_2	S_2	S_2	S_3
S_2	S_4	S_1	S_3	S_3
S_3	S_2	S_3	S_3	S_3
S_4	S_4	S_4	S_2	S_3

(a) Transition table for an FSM.

S	Q_1Q_2
S_1	0 0
S_2	0 1
S_3	1 0
S_4	1 1

I	x_1x_2
I_1	0 0
I_2	0 1
I_3	1 0
I_4	1 1

(b) An encoding for the FSM.

Q_1Q_2 \ x_1x_2	00	01	11	10
00	0	0	1	0
01	1	0	1	1
11	1	1	1	0
10	0	1	1	1

Q_1^+

Q_1Q_2 \ x_1x_2	00	01	11	10
00	1	1	0	1
01	1	0	0	0
11	1	1	0	1
10	1	0	0	0

Q_2^+

(c) Encoded transition functions resulting from the encoding in (b).

S	Q_1Q_2
S_1	0 1
S_2	1 0
S_3	1 1
S_4	0 0

I	x_1x_2
I_1	1 0
I_2	1 1
I_3	0 1
I_4	0 0

(d) Another encoding for the FSM.

Q_1Q_2 \ x_1x_2	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

Q_1^+

Q_1Q_2 \ x_1x_2	00	01	11	10
00	1	0	0	0
01	1	0	0	0
11	1	1	1	0
10	1	1	1	0

Q_2^+

(e) Encoded transition functions resulting from the encoding in (d).

Figure 7.2: Illustration of different encodings for an FSM resulting in different costs.

We therefore see that between these two encodings, the encoding from Figure 7.2d results

in encoded transition functions that depend on less variables. If we make the reasonable assumption that the complexity of a digital logic circuit is correlated with its number of inputs, then we would expect the encoding from Figure 7.2d to ultimately result in a less complex digital logic circuit, since the next-state logic for both flip-flops would involve fewer variables. Of course, this correlation between complexity and number of inputs depends on the precise definition of complexity used, and is not perfect in any case. There is no guarantee that the encoding from Figure 7.2d would actually produce a digital logic circuit that better suits the design goals (whatever they may be) of a digital logic designer. Nevertheless, in the remainder of this paper we will use the simple metric of defining cost as the total number of variables on which a Boolean function depends, for two reasons. First, this cost metric simplifies the problem of finding the optimal encoding for an FSM enough that we can always find the exact minimum solution. The only published work so far that achieves exact minimum results for FSM encoding is [83]. However, the authors in [83] use a different cost metric, which is based on the number of product terms when digital logic is implemented using a programmable logic array (PLA). This brings us to our second reason for using a cost metric based on number of dependencies: minimizing the number of product terms in a PLA has little to no relevance if an FSM is not implemented using PLAs, as they are not (for instance) in most modern VLSI chips. Minimizing the number of variable dependencies of a Boolean function, on the other hand, is a reasonable goal for virtually any digital logic technology, and will likely remain reasonable even for future technologies.

Based on the preceding discussion, we therefore formulate as follows the problem to be solved in the remainder of this paper—given an FSM that satisfies the following conditions:

- the number of states in the machine is a power of 2;
- the number of possible input values to the machine is a power of 2;

- the machine is state-minimal, meaning that no two distinct states are equivalent and therefore no two distinct states may be assigned the same encoding;
- no two input values are equivalent and therefore no two distinct input values may be assigned the same encoding;

find an encoding for the FSM with the lowest possible cost, where the cost of an encoding is defined as the sum of the costs of the encoded transition functions resulting from that encoding, and the cost of a single function is the number of variables on which the function depends. A function is considered to depend on a variable if and only if the function cannot be computed without knowledge of the value of that variable. Our cost model thus defined is the same as that used in [77] and [78].

7.2 Use of Grover's algorithm to solve optimization problems

At this point, before turning to the design of a quantum oracle to solve the state encoding problem, we first consider an interesting issue regarding the usage of Grover's algorithm for this problem, as it will affect the design of the quantum oracle. Specifically, some difficulty arises from the fact that, while state encoding cost minimization is an optimization problem, Grover's algorithm directly solves only satisfaction problems (a.k.a. decision problems). In other words, we wish to find the minimum value of a certain function (the cost function) while Grover's algorithm can only find a point at which a function evaluates to 1. In order to use Grover's algorithm for optimization, we reformulate optimization problems in terms of a sequence of satisfaction problems of the form: "find a point at which the value of the function f is less than r", where r is an arbitrary threshold. By executing Grover's algorithm for different values of the threshold r, one can conduct a search to find the minimum value of f. For example, such a search might proceed according to the following procedure:

1. Choose an initial value for the threshold a. Ideally, this initial value should be close to the minimum value of f, if an estimate of the minimum is available; if not, the search procedure will still function correctly with any initial value.

2. Execute Grover's algorithm for the chosen threshold as previously described.

3. If Grover's algorithm finds a solution, proceed to step 4. Otherwise, double the threshold a and repeat from step 2.

4. The preceding steps give a lower and upper bound for the minimum of f. In other words, one obtains threshold values $a_{\min}$ and $a_{\max}$ such that the function f never takes on any value less than $a_{\min}$ (as determined using Grover's algorithm) but takes on a value less than $a_{\max}$ at one or more points.

5. Execute Grover's algorithm for a threshold value of $(a_{\min} + a_{\max})/2$.

6. If Grover's algorithm finds a solution, then $(a_{\min} + a_{\max})/2$ gives a new upper bound for the minimum of f; otherwise, it gives a new lower bound. Repeat from step 4 until the minimum value of f is determined. The final output of Grover's algorithm gives the input to f at which the minimum occurs.

The above sequence of steps essentially describes the well-known binary search strategy. We give this strategy merely as an example showing that such a search is indeed possible. In particular, we do not claim that this strategy is optimal with respect to expected runtime or any other measure. The question of evaluating different search strategies, as well as what standard should be used to evaluate them in the first place, falls outside the scope of this paper. We leave this avenue of exploration open for future work.

We make the crucial observation that the above procedure involves executing Grover's algorithm using not a single, static quantum oracle, but a *sequence* of quantum oracles that are dynamically created on-the-fly for different thresholds. In particular, the threshold value r is not itself an input to the oracle, meaning that Grover's algorithm does *not* directly search for the threshold. Grover's algorithm is in fact incapable of directly searching for the threshold since, as previously discussed, that constitutes an optimization, not satisfaction, problem. Instead, the threshold value is built into the oracle, meaning that a different oracle is created for each threshold value. Consequently, to successfully use the above procedure, one requires not just a single quantum oracle but a method for generating quantum oracles for arbitrary threshold values.

We additionally observe that repeated execution of Grover's algorithm using a sequence of dynamically generated quantum oracles makes use of the freely reconfigurable nature of quantum circuits. More specifically, a quantum circuit is not a hardware circuit in the same sense as a classical digital logic circuit—in a quantum circuit, information is not transmitted through physical wires from gate to gate. Instead, the "wires" in a quantum circuit represent individual qubits that are stored on some physical medium, and the gates are not physical components but rather are manipulations performed on the physical medium using implementation-dependent hardware (*e.g.*, lasers, electromagnets, superconducting circuits). This means that a quantum "circuit" is in fact a sequence of software operations stored on a classical computer that controls the quantum hardware, and this sequence of operations can easily be modified at will. Thus, executing Grover's algorithm using a newly-generated quantum oracle simply involves having the classical computer perform the appropriate, newly-generated sequence of operations on the quantum hardware.

Based on the preceding observations, we introduce a distinction between *compile time*

and *run time* for quantum circuits. In classical computing, compile time is the time at which instructions for the computer are generated, while run time is the time at which those instructions are actually executed. By analogy, compile time of a quantum circuit will refer to the generation process of the quantum circuit (which, as previously noted, takes place on a classical computer). Run time of a quantum circuit, in contrast, will refer to the process of actually executing the quantum circuit on quantum hardware (which, as also noted, involves a classical computer controlling the quantum hardware with an appropriate sequence of commands).

Therefore, our objective in the subsequent sections of this paper becomes: given an FSM specification, demonstrate a procedure that can generate a quantum oracle for an arbitrary threshold value r, where the quantum oracle accepts as input an encoding for the FSM and answers the question "is the cost of the encoding (as defined in Section 7.1.2) less than r?" The process described in this section then constitutes a complete algorithm for finding an encoding for the given FSM with exact minimum cost.

7.3 Procedure to calculate the cost of a given encoding

7.3.1 Computing cost by considering pairs of states

We now consider a systematic procedure for computing the cost, as defined in Section 7.1.2, of a given encoding for a given FSM. This procedure will form the basis for the design of a quantum oracle that determines whether the cost of a given encoding is less than a predetermined threshold, therefore allowing the use of Grover's algorithm to find the exact minimum-cost encoding for an FSM.

To calculate the cost of a given encoding, we require a method to determine whether the value of a given state variable, say Q_i, at time $t + 1$ depends on the value of any (possibly

the same or different) state variable, say Q_j, at time t. By definition, the value of Q_i at time $t + 1$ is given by the function δ_i; thus,

$$Q_i^+ = \delta_i(Q_1, \ldots, Q_n, x_1, \ldots, x_m) \tag{7.6}$$

where n and m are the number of state and input variables, respectively. Now, Q_i^+ only depends on Q_j if, in at least one case, a change in Q_j and in no other variables causes a change in Q_i^+. In other words, if there exist distinct states S, S′ and an input value I such that Q_j is the only state variable assigned different values for S and S′, and Q_i is assigned different values for $\delta(\mathrm{S}, \mathrm{I})$ and $\delta(\mathrm{S}', \mathrm{I})$, then Q_i^+ depends on Q_j.

As an example, consider the state machine from Figure 7.2 and the encoding in Figure 7.2b. The encoded transition function $Q_1^+ = \delta_1(Q_1, Q_2, x_1, x_2)$ resulting from this encoding, shown in Figure 7.2c, depends on all four variables. The dependency on Q_1 can be seen from the fact that $\delta_1(0, 0, 0, 1) = 0$ but $\delta_1(1, 0, 0, 1) = 1$; *i.e.*, a change in only Q_1 causes a change in δ_1. In terms of states and input values, $Q_1Q_2 = 00$ corresponds to a current state of S_1 and $Q_1Q_2 = 10$ corresponds to S_3. We then see that given the pair of states $(\mathrm{S}_1, \mathrm{S}_3)$, whose encodings differ only in Q_1, for at least one input value—in this case, I_2, which corresponds to $x_1x_2 = 01$—the corresponding pair of next states, $(\mathrm{S}_2, \mathrm{S}_3)$, is such that the value of Q_1 differs between the two states in the pair.

The result of the preceding discussion may be more formally expressed as follows. For any two distinct states S and S′, let $D_j(\mathrm{S}, \mathrm{S}')$ mean "the encodings of S and S′ differ only in the value of Q_j" and let $A_i(\mathrm{S}, \mathrm{S}')$ mean "the encodings of S and S′ agree in the value of Q_i". Then, check whether

$$D_j(\mathrm{S}, \mathrm{S}') \Rightarrow \forall \mathrm{I} \in \mathbf{I}\ A_i\big(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}', \mathrm{I})\big) \tag{7.7}$$

for all pairs of distinct states $(\mathrm{S}, \mathrm{S}')$. If so, then Q_i^+ does *not* depend on Q_j; otherwise, it does.

We determine the dependency of a given Q_i^+ on an input variable x_j in a similar manner. Specifically, Q_i^+ depends on x_j if and only if, in at least one case, a change in x_j and in no other state or input variables causes a change in Q_i^+. Therefore, we consider all pairs of distinct input values and determine whether, for those pairs whose encodings differ only in x_j, the encodings of the corresponding pair of next states can ever differ in Q_i. In other words, for any two distinct input values I and I′, let $D_j(\mathrm{I}, \mathrm{I}')$ mean "the encodings of I and I′ differ only in the value of x_j", and for any two states S and S′, let $A_i(\mathrm{S}, \mathrm{S}')$ mean (as before) "the encodings of S and S′ agree in the value of Q_i". Then, we wish to check whether the condition

$$D_j(\mathrm{I}, \mathrm{I}') \Rightarrow \forall \mathrm{S} \in \mathbf{S}\ A_i\big(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}')\big) \tag{7.8}$$

holds for every pair of distinct input values $(\mathrm{I}, \mathrm{I}')$. If so, Q_i^+ does *not* depend on x_j; otherwise, it does.

Equipped with a procedure to determine the dependency of a given Q_i^+ on any single state or input variable, it is now straightforward to compute the total cost of an encoding. We calculate the cost of each Q_i^+ in accordance with our cost model—the total number of state and input variables on which Q_i^+ depends gives the cost of Q_i^+. Then, the sum of costs of Q_i^+ for $1 \leq i \leq n$ gives the total cost of a given encoding.

7.3.2 Necessity for transition functions to be completely specified

The just-described procedure only gives the correct cost if the encoding is bijective. If this condition is not satisfied, problems can arise from the fact that the "functions" δ_i are no longer functions in the mathematical sense, but rather relations or so-called incompletely

	I_1	I_2	I_3
S_1	S_2	S_4	S_5
S_2	S_1	S_1	S_1
S_3	S_3	S_3	S_1
S_4	S_3	S_3	S_3
S_5	S_1	S_3	S_1

(a) Transition table of the FSM.

S	$Q_1Q_2Q_3$
S_1	0 0 0
S_2	0 1 1
S_3	1 0 1
S_4	1 1 0
S_5	1 1 1

I	x_1x_2
I_1	0 0
I_2	0 1
I_3	1 0

(b) A possible encoding for the FSM.

$Q_1Q_2Q_3$ \ x_1x_2	00	01	11	10
000	1	1	–	1
001	–	–	–	–
011	0	0	–	0
010	–	–	–	–
110	0	0	–	0
111	0	0	–	0
101	0	0	–	0
100	–	–	–	–

(c) Resulting form of Q_2^+.

Figure 7.3: Don't-cares in a transition function for an FSM with numbers of states and inputs not powers of 2.

specified functions. In general, when δ_i is incompletely specified, the question of whether or not δ_i depends on a given variable cannot be answered by the simple procedure described above.

Figure 7.3 illustrates an instance where the procedure from the previous section fails to correctly determine the dependencies of a function δ_i on Q_1 through Q_n and x_1 through x_m. In this figure we have a state machine where the number of states (5) is not a power of two. Consequently, we require at least three flip-flops to implement this state machine. However, since eight distinct states exist for an array of three flip-flops, three flip-flop array states remain unused. In other words, three of the eight possible flip-flop array states do not represent any FSM state at all. Similarly, the number of input values (3) is also not a power of two, and hence the input is encoded by two bits with one of the four possible combinations remaining unused. The effects of these unused states and input combinations can be seen in the encoded transition function δ_2, where the value of δ_2 for the unused states is recorded as a "–", indicating a "don't-care". A "don't-care" output indicates that the digital logic implementing the FSM may output a value of either 0 or 1 for that combination of flip-flop states and inputs, since the combination should never occur during normal operation.

We now observe that at least three possible implementations exist for δ_2:

$$Q_2^+ = \neg Q_1 \wedge \neg Q_2, \tag{7.9}$$

$$Q_2^+ = \neg Q_1 \wedge \neg Q_3, \tag{7.10}$$

$$Q_2^+ = \neg Q_2 \wedge \neg Q_3. \tag{7.11}$$

Thus, it is clearly possible to implement δ_2 with a dependency on only two variables. On the other hand, one can see from inspection that any one of Q_1, Q_2, Q_3, x_1, or x_2 alone is not enough to determine the value of δ_2. Hence, our cost model assigns δ_2 a cost of 2.

However, the procedure from the previous section finds the cost of δ_2 to be 0. For instance, when considering whether δ_2 depends on Q_1, the procedure considers all pairs of states $(\mathrm{S}, \mathrm{S}')$ whose encodings differ only in Q_1, and then examines whether the encodings of $\delta(\mathrm{S}, \mathrm{I})$ and $\delta(\mathrm{S}', \mathrm{I})$ differ in Q_2 for any input value I. In this case, the only such pair of states is $(\mathrm{S}_2, \mathrm{S}_5)$, and under δ with $\mathrm{I} = \mathrm{I}_1$, $\mathrm{I} = \mathrm{I}_2$, and $\mathrm{I} = \mathrm{I}_3$, the images of this pair are $(\mathrm{S}_1, \mathrm{S}_1)$, $(\mathrm{S}_1, \mathrm{S}_3)$, and $(\mathrm{S}_1, \mathrm{S}_1)$, respectively. For all three image pairs, Q_2 does not differ between the encoded values of the two states in the pair. Thus, the procedure determines that δ_2 does not depend on Q_1. In a similar fashion, the procedure also determines that δ_2 does not depend on Q_2, Q_3, x_1, or x_2, and consequently calculates the cost of δ_2 as 0. This conclusion is clearly incorrect since δ_2 is not constant and must therefore depend on at least one variable. Indeed, we previously determined the minimum number of dependencies to be 2.

Our procedure's failure to correctly determine cost in this case ultimately arises from the fact that, considered individually, δ_2 need not depend on any particular one of Q_1, Q_2, and Q_3. For instance, (7.11) implements δ_2 without a dependency on Q_1. Similarly, (7.10) and (7.9)

implement δ_2 without dependencies on Q_2 and Q_3, respectively. It is however not possible to *simultaneously* avoid two of these dependencies. In other words, any implementation of δ_2 that lacks a dependency on Q_1 must then depend on both Q_2 and Q_3. So, although the implementation given by (7.11) does not depend on Q_1, it depends on both Q_2 and Q_3. The procedure from the previous section assumes (in this case incorrectly) that any two dependencies can always be simultaneously avoided if they can be individually avoided, which results in an incorrect computed cost.

The scenario described here can never occur if all encoded transition functions are actual functions (not relations) in the mathematical sense. To see this, observe that any mathematical function lacking a dependency on each of two variables individually must also lack a dependency on those variables simultaneously. In other words, suppose that a function f of variables[1] x_1 through x_n lacks a dependency on x_1, meaning that a change in only x_1 cannot affect the value of f and f can be computed without knowledge of the value of x_1. Furthermore suppose that f similarly lacks a dependency on x_2. Then if the value of x_1 changes to a different value x_1' and the value of x_2 simultaneously changes to x_2', we have[2]

$$f(x_1, x_2, \ldots, x_n) = f(x_1, x_2', \ldots, x_n) = f(x_1', x_2', \ldots, x_n), \tag{7.12}$$

showing that simultaneous changes in x_1 and x_2 cannot affect the value of f. Hence, f can be computed without knowledge of the values of either x_1 or x_2. This argument breaks down if f is not actually a function but a relation, because then the value of f may not be uniquely defined at a given point.

[1]The variables x_1 through x_n here are unrelated to the input variables of an FSM; here, they are simply variables in an arbitrary function f.

[2]Here we use the notation x_1' simply to indicate that the value of x_1 has changed. In particular, we do not use the prime ($'$) symbol to indicate logical negation, as is sometimes done.

We therefore see that the procedure from the previous section, which only evaluates dependency on a single variable at a time, correctly computes cost when all encoded transition functions are actual functions (*i.e.*, there are no "don't-cares") but may fail if those "functions" are actually relations. No "don't-cares" will exist if the state and input encodings are bijective. Bijectivity of an encoding is equivalent to the condition that the numbers of states and inputs are both powers of two, the minimum possible number of state and input variables are used, and no two distinct states or inputs may be assigned the same encoding.

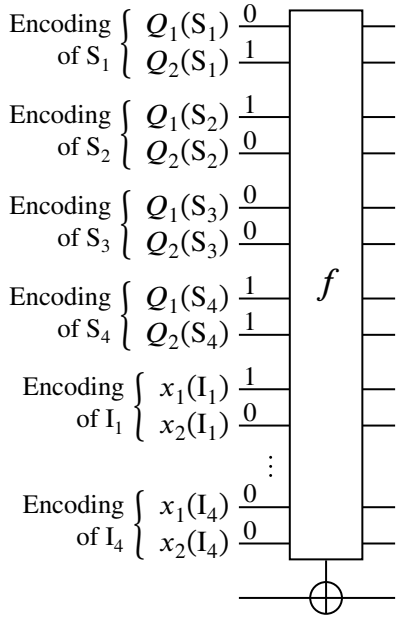
7.4 Design of a quantum oracle to find optimal encodings

7.4.1 Binary representation of encodings in the quantum oracle

We now demonstrate how to, given an FSM and a threshold value r, construct a quantum oracle that, when given an encoding for that FSM as input, determines whether the cost of the encoding is less than r. Using this quantum oracle, the procedure described in Section 7.2 then constitutes a complete algorithm for finding the exact minimum solution to the FSM encoding problem under the assumptions and conditions described before. As the first design step, we must agree on the manner in which a candidate encoding is to be supplied as input to the oracle. We will use the following scheme to represent an encoding as binary data: letting n be the number of bits used by the encoding, we allocate an array of n qubits for each element of the set being encoded (either the state or input set of an FSM), and assign to each such array the encoded value of the corresponding set element. For instance, suppose that S_1 through S_4 are the internal states of an FSM. Since we require all encodings to be bijective, only 2-bit state encodings will be considered for this FSM. We therefore create four arrays of two qubits each, where the input supplied to each array is the encoded value of the corresponding state. Thus, if S_1, S_2, S_3, and S_4 are encoded by 01, 10, 00, and 11,

S	$Q_1 Q_2$	I	$x_1 x_2$
S_1	0 1	I_1	1 0
S_2	1 0	I_2	1 1
S_3	0 0	I_3	0 1
S_4	1 1	I_4	0 0

(a) State and input encodings for an FSM.



(b) Their representations as supplied to the oracle.

Figure 7.4: The quantum oracle's representation of an encoding as binary data.

respectively, then we represent this encoding by supplying 01 on the first array of two qubits, 10 on the second array, 00 on the third array, and 11 on the fourth array. Input encodings are also represented in a similar manner.

Figure 7.4 provides an example of an encoding represented as binary data that can be input to a quantum oracle. In this figure as well as others in this section, the notation $Q_i(S_j)$ denotes the value assigned to Q_i in the encoding of S_j. Similarly, $x_i(I_j)$ denotes the value assigned to x_i in the encoding of I_j.

The number of input qubits to a quantum oracle is of great importance as it determines the run time of Grover's algorithm. If an FSM has 2^n internal states, each state will be encoded using n bits, and therefore, our scheme for representing encodings requires 2^n arrays of n qubits each, for a total of $n \cdot 2^n$ qubits. Similarly, for an FSM with 2^m input values, our

scheme requires $m \cdot 2^m$ qubits. Thus, an FSM with 2^n internal states and 2^m input values requires a grand total of $n \cdot 2^n + m \cdot 2^m$ qubits.

7.4.2 Quantum circuit to detect dependencies

We next demonstrate a quantum circuit design that uses the procedure from Section 7.3.1 to determine whether the next state of a single flip-flop (*i.e.*, one of Q_1^+ through Q_n^+) depends on the current state of another flip-flop (*i.e.*, one of Q_1 through Q_n) or the current value of a single input bit (*i.e.*, one of x_1 through x_m). Recall that this procedure involves checking the conditions given by (7.7) and (7.8) for each pair of states or input values, respectively. In turn, (7.7) and (7.8) involve checking the conditions $D_j(\mathrm{S}, \mathrm{S}')$, $A_i(\mathrm{S}, \mathrm{S}')$, and $D_j(\mathrm{I}, \mathrm{I}')$ for pairs of states or input values, where D_j and A_i are as defined in Section 7.3.1.

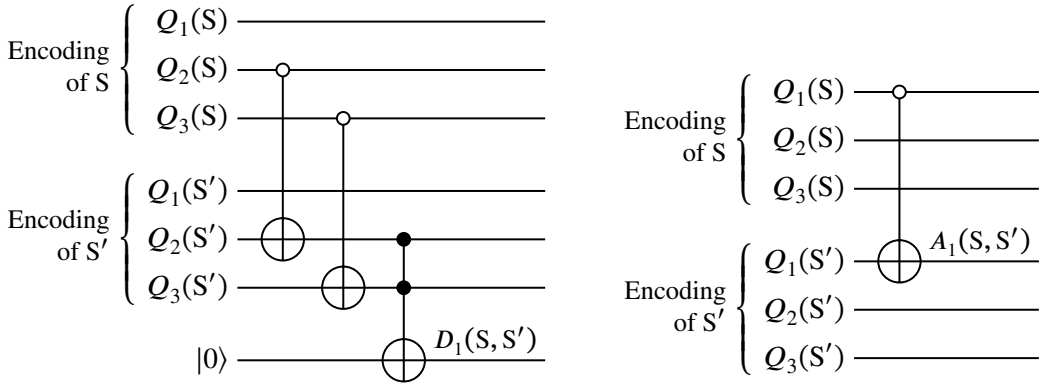
Using the binary representation described in Section 7.4.1, one can easily construct quantum circuits to check $D_j(\mathrm{S}, \mathrm{S}')$, $A_i(\mathrm{S}, \mathrm{S}')$, and $D_j(\mathrm{I}, \mathrm{I}')$ for any pair of states or inputs. For example, Figure 7.5a shows a quantum circuit that evaluates $D_1(\mathrm{S}, \mathrm{S}')$ for any two distinct states S and S′.[3] This circuit operates on just a subset of the complete binary representation of an encoding; specifically, it uses the qubits carrying information about the encoded values of S and S′. It uses CNOT gates to perform comparisons and a Toffoli gate to evaluate the logical AND of two comparison results. The final output of the circuit is given by the logical expression $\neg\big(Q_2(\mathrm{S}) \oplus Q_2(\mathrm{S}')\big) \wedge \neg\big(Q_3(\mathrm{S}) \oplus Q_3(\mathrm{S}')\big)$, which evaluates to 1 if and only if

[3] Technically, this circuit only works correctly if the states S and S′ have distinct encodings, as required by the conditions stated at the end of Section 7.1.2. If S and S′ happen to have the same encoding, the circuit will erroneously output 1. However, later on, in Section 7.4.5, we describe a circuit that enforces the condition that distinct states have distinct encodings. This circuit is incorporated into the final oracle, as described in Section 7.4.6, so that it forces the oracle's output to 0 whenever this condition is violated. The operation of the circuit described in the present section therefore becomes irrelevant in such a case, since Grover's algorithm will never find such a distinctness-violating encoding as a solution. From now on, we therefore proceed with the implicit assumption that distinct states have distinct encodings. The same applies to input values as well—the operation of the circuit described here is similarly irrelevant if distinct input values do not have distinct encodings.

the encodings of S and S′ agree in all state variables except Q_1, exactly the definition of $D_1(\mathrm{S}, \mathrm{S}')$.

Although we assumed for the purpose of this particular illustration that each state is encoded by three bits, the general circuit structure applies to encodings of any size. Likewise, although this particular circuit evaluates $D_1(\mathrm{S}, \mathrm{S}')$, similar circuits suffice to evaluate $D_j(\mathrm{S}, \mathrm{S}')$ for any j, as well as $D_j(\mathrm{I}, \mathrm{I}')$ for two input values I and I′ instead of two states.

In a similar vein, Figure 7.5b shows a quantum circuit that evaluates $A_1(\mathrm{S}, \mathrm{S}')$. This circuit is extremely simple, as it simply evaluates $\neg\big(Q_1(\mathrm{S}) \oplus Q_1(\mathrm{S}')\big)$, which amounts to just a one-bit comparison. As before, the same circuit structure is usable for encodings of any size, not just three bits.



(a) Circuit to check $D_1(\mathrm{S}, \mathrm{S}')$. **(b)** Circuit to check $A_1(\mathrm{S}, \mathrm{S}')$.

Figure 7.5: Quantum circuits to check $D_1(\mathrm{S}, \mathrm{S}')$ and $A_1(\mathrm{S}, \mathrm{S}')$.

Next, to check the conditions specified by (7.7) and (7.8), our quantum circuit must be able to evaluate a logical implication. Recalling that the expression $a \Rightarrow b$ is defined as $\neg a \vee b$, we observe that a single Toffoli gate suffices to evaluate a logical implication, as shown in Figure 7.6.

Finally, we are ready to turn to the design of the full quantum circuit for checking (7.7)

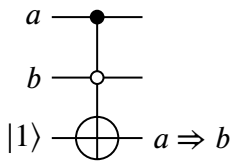


Figure 7.6: Logical implication evaluated using a Toffoli gate.

or (7.8). This task is complicated by the fact that checking (7.7) or (7.8) involves evaluating $A_i(\mathrm{S}, \mathrm{S}')$ simultaneously for many pairs of states. For instance, consider the state machine from Figure 7.2, and suppose that we wish to design a quantum circuit to evaluate (7.8) for $i = 1$, $j = 2$, and the pair of input values $(\mathrm{I}, \mathrm{I}') = (\mathrm{I}_1, \mathrm{I}_2)$. The circuit must then check whether $A_1\big(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}')\big)$ holds for all states S. In this case, given $\mathrm{I} = \mathrm{I}_1$ and $\mathrm{I}' = \mathrm{I}_2$, the corresponding pairs $\big(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}')\big)$ of next states are $(\mathrm{S}_2, \mathrm{S}_2)$, $(\mathrm{S}_4, \mathrm{S}_1)$, $(\mathrm{S}_2, \mathrm{S}_3)$, and $(\mathrm{S}_4, \mathrm{S}_4)$ for $\mathrm{S} = \mathrm{S}_1$, S_2, S_3, and S_4, respectively. Since $A_1(\mathrm{S}_2, \mathrm{S}_2)$ and $A_1(\mathrm{S}_4, \mathrm{S}_4)$ are trivially true, we can ignore them. Thus, the quantum circuit must check whether $A_1(\mathrm{S}_4, \mathrm{S}_1)$ and $A_1(\mathrm{S}_2, \mathrm{S}_3)$ simultaneously hold. The upper portion of Figure 7.7 demonstrates a subcircuit that accomplishes this task. The lower portion of Figure 7.7 then combines this subcircuit with another subcircuit (from Figure 7.5b) to evaluate the full (7.8). In other words, the final output of this circuit, on the bottommost qubit, will be 1 if (7.8) is satisfied, and 0 if it is not. We reiterate that for an FSM with more than four input values (so that each input value is encoded by at least three bits), one would replace the simplified circuit for evaluating $D_2(\mathrm{I}_1, \mathrm{I}_2)$ with the full one from Figure 7.5a.

In some cases, checking eq. (7.7) or (7.8) may involve evaluating $A_i(\mathrm{I}, \mathrm{I}')$ simultaneously for two or more overlapping pairs of inputs (or states). In these scenarios, one must construct the quantum circuit for checking eq. (7.7) or (7.8) using a slightly different design. For example, suppose that we now wish to design a quantum circuit to evaluate (7.8) for the same state machine and $i = 1$, $j = 2$ as before, but a different pair of input values $(\mathrm{I}_1, \mathrm{I}_3)$.

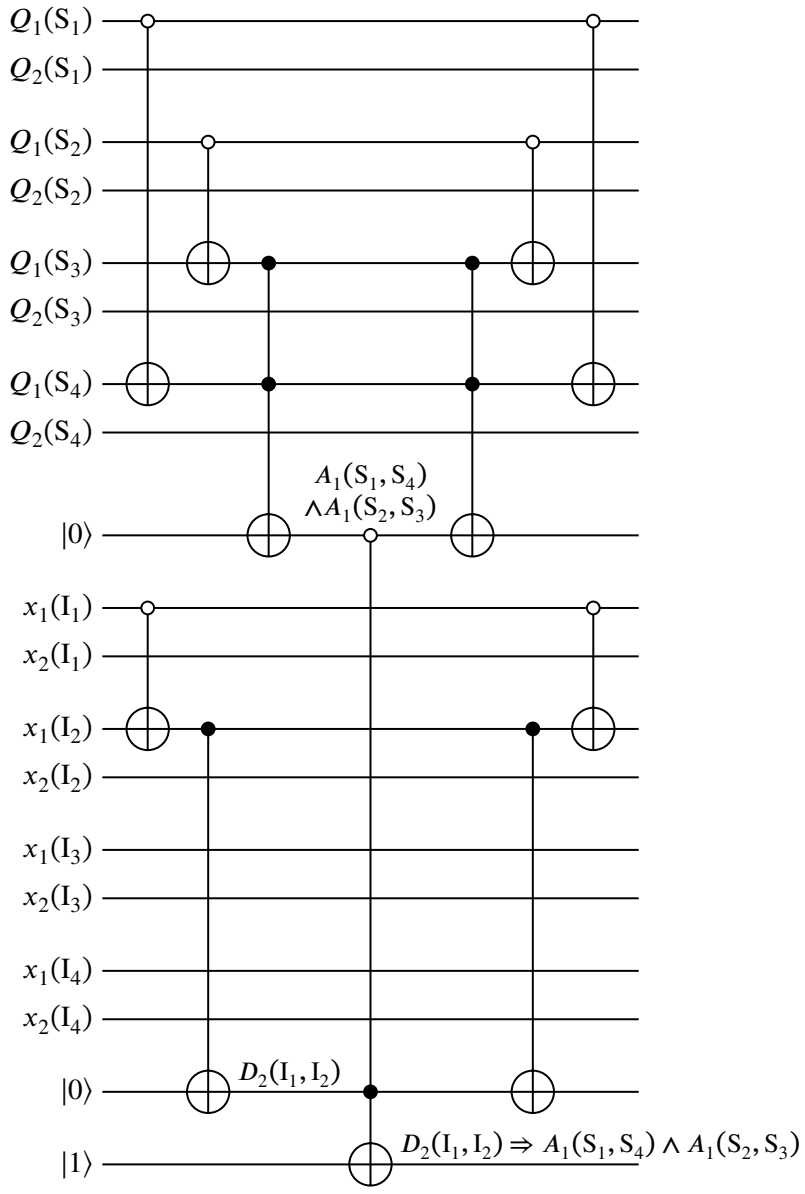


Figure 7.7: Quantum circuit to evaluate (7.8) for input pair (I_1, I_2).

The pairs of next states corresponding to current states of S_1, S_2, S_3, and S_4 are (S_2, S_2), (S_4, S_3), (S_2, S_3), and (S_4, S_2), respectively. Therefore, the quantum circuit must evaluate

$$D_2(I_1, I_3) \Rightarrow A_1(S_4, S_3) \wedge A_1(S_2, S_3) \wedge A_1(S_4, S_2). \tag{7.13}$$

We then observe that, since the pairs of states on the right-hand side overlap, the entire right-hand side is equivalent to the condition that the encodings of S_2, S_3, S_4 must *all* agree in the value of Q_1.[4] We denote this condition by $A_1(S_2, S_3, S_4)$, a natural generalization of our earlier $A_i(S, S')$ notation for pairs of states. Figure 7.8 illustrates the quantum circuit for evaluating (7.8) in this case. The critical difference between Figures 7.7 and 7.8 is that the CNOT gates in the upper portion of Figure 7.8 operate on the encodings of overlapping pairs of states, and must therefore be applied in the correct order. Additionally, although the right-hand side of (7.13) contains the term $A_1(S_4, S_2)$, the circuit in Figure 7.8 does not contain a corresponding CNOT gate to evaluate this term. Such a gate is unnecessary because of transitivity—it is enough to check both $A_1(S_2, S_3)$ and $A_1(S_3, S_4)$ since they are together equivalent to $A_1(S_2, S_3, S_4)$, which implies $A_1(S_4, S_2)$.

More generally, one can construct similar circuits to evaluate $A_i(S, S')$ for any number of overlapping pairs of states. One simply takes the union of all such pairs to obtain an arbitrarily-sized set of states and then constructs a quantum circuit according to the pattern shown in Figure 7.9a. Figure 7.9b then shows a circuit that checks whether A_i is simultaneously satisfied for multiple sets of states. These circuit structures are sufficient to algorithmically construct a quantum circuit for evaluating (7.7) or (7.8) in any case. Specifically, given any FSM with any number of states and input values, the following procedure

[4] For this particular example, this condition is impossible because, since we require encodings to be bijective, the encodings of at most two states can agree in the value of Q_1. However, this condition could be satisfied in an FSM with more states.

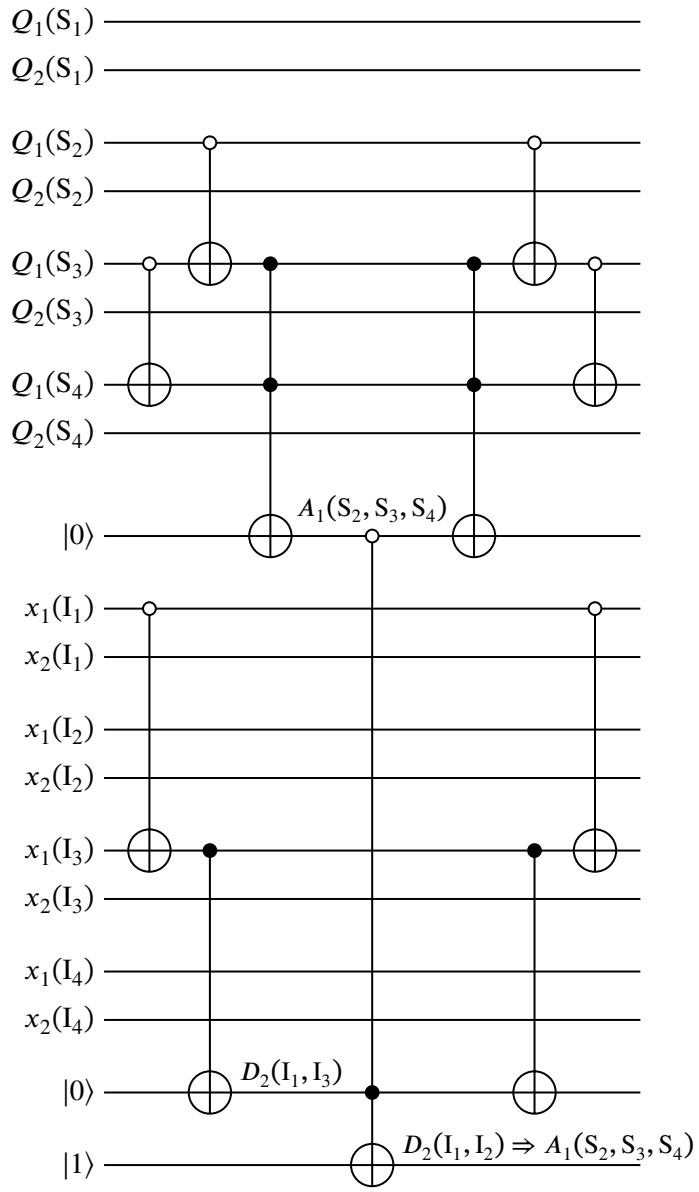


Figure 7.8: Quantum circuit to evaluate (7.8) for input pair (I_1, I_3).

constructs a quantum circuit to evaluate (7.7) for any i, j, and pair of states[5] $(\mathrm{S}, \mathrm{S}')$:

1. List all of the pairs of states appearing on the right-hand side of (7.7) when expanded. In other words, create a list containing the pair of states $\big(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}', \mathrm{I})\big)$ for every possible input value I.

2. In this list, merge any overlapping pairs of states together to obtain a collection of mutually disjoint sets of states.

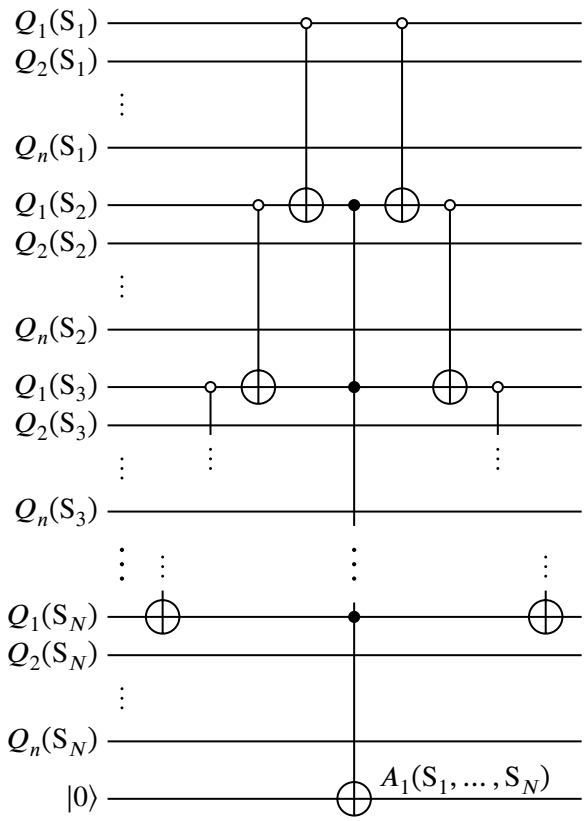
3. Using the circuit structures from Figure 7.9, construct a circuit that checks whether A_i is simultaneously satisfied for every set of states produced by the previous step.

4. Using the circuit created in step 3 as a subcircuit, construct a circuit that checks the full (7.7), following the general pattern illustrated in Figures 7.7 and 7.8.

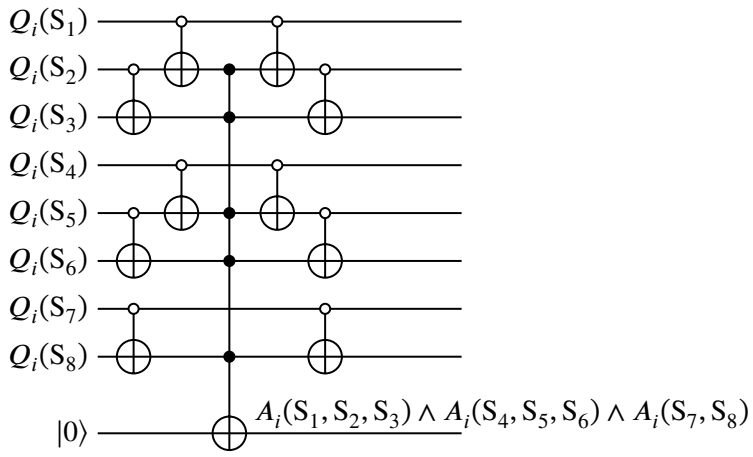
From now on, we will refer to any circuit generated by this procedure as a *partial dependency checker* for the given pair of states, because it checks (7.7) for a single pair of states while the existence of a dependency is determined by checking (7.7) for all possible pairs of states.

We observe that crucially, the quantum circuit design procedure described in this section requires at compile time only knowledge of the transition function of the FSM being encoded. In particular, the quantum circuit cannot be modified depending on the encoding whose cost is being evaluated, because individual encodings are not considered at compile time. Individual encodings are only considered at run time, when Grover's algorithm is used to simultaneously evaluate the cost of every possible encoding in the search space. Thus, in a single run of Grover's algorithm, the same quantum circuit must be able to evaluate any encoding in the search space without any modifications.

[5] We describe the procedure here using (7.7) with a pair of states, but it functions equally well for (7.8) with a pair of input values.



(a) Circuit to evaluate $A_1(S_1, \ldots, S_n)$ for an arbitrary number of states.



(b) Circuit for multiple sets of states.

Figure 7.9: Construction of quantum circuits to evaluate any number of A_i terms.

7.4.3 Quantum circuit to calculate total cost of an encoding

With the ability to generate a quantum circuit to evaluate eqs. (7.7) or (7.8) for any i, j, and pair of states or inputs, we now consider the task of designing a quantum circuit to calculate the full cost of a given encoding. Following the procedure from Section 7.3.1, the quantum circuit determines whether a given Q_i^+ depends on Q_j by checking if (7.7) is satisfied for *all* possible pairs of states.[6] Figure 7.10 shows a circuit structure that accomplishes this task. From now on, we will refer to this circuit as a *dependency checker*.

In Figure 7.10, the labeled "input qubits" represent the entire collection of qubits storing the binary representation of an encoding, as described in Section 7.4.1. Each block represents a partial dependency checker that checks (7.7) for the given i, j, and the pair of states with which it is labeled. "Ancilla qubits" represents the ancilla qubits that are used by these partial dependency checkers, as shown in Figures 7.7 and 7.8. For illustrative purposes, we have assumed an FSM with four states, resulting in six possible pairs of states; partial dependency checkers for three of them are shown. The same circuit structure, scaled up to accommodate the correspondingly larger number of subcircuits, would be used for an FSM with more states. The output from each partial dependency checker is stored on an ancilla qubit so that the final result can be obtained by taking their logical AND. Each partial dependency checker must be applied again after the result is computed to restore the ancilla qubits to their original states, which, as discussed in Chapter 1, Section 1.4, is required for Grover's algorithm to work correctly. The final output from the dependency checker is 1 if and only if Q_i^+ depends on Q_j.

Next, we require another quantum circuit to calculate the total cost of the encoding by

[6] Once again, although we use dependencies on state variables for the sake of explanation, the whole discussion applies to dependencies on input variables as well.

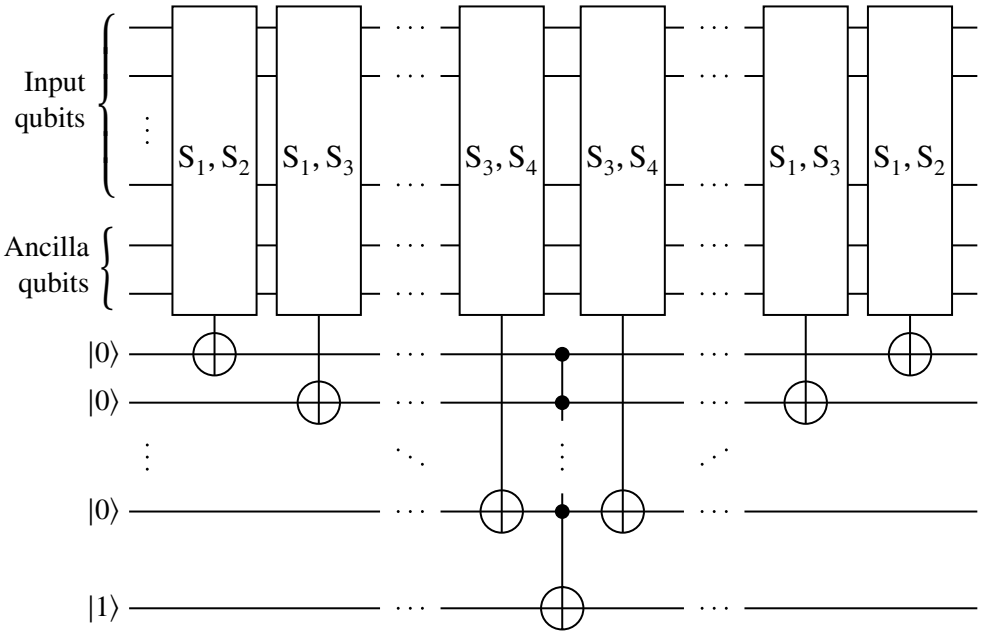
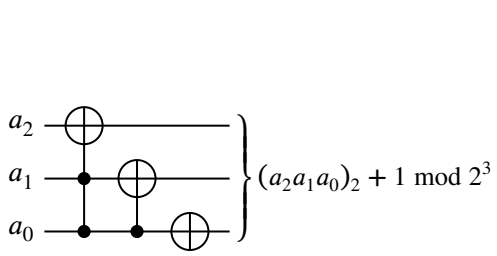


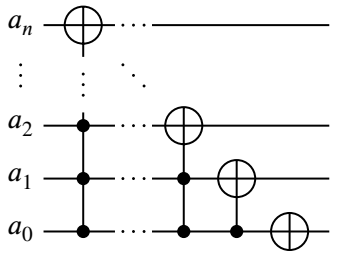
Figure 7.10: Quantum circuit to check dependency of Q_i^+ on a single state or input variable.

counting the total number of dependencies over all Q_i^+. We introduce a *quantum counter* for this purpose. The quantum counter consists of a register of qubits, which stores an integer in base-two representation, together with *incrementer* circuits that add one to this stored integer when a control qubit is in the state $|1\rangle$.

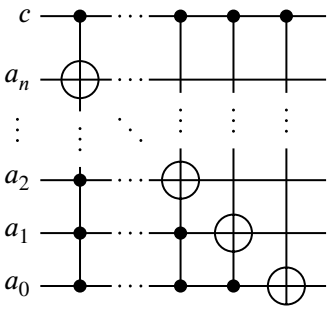
Figure 7.11a shows a three-qubit incrementer. If the input qubits represent an integer $a_2a_1a_0$, with a_2 being the most significant and a_0 being the least significant bit, then the output of the incrementer is (the base-two representation of) $a_2a_1a_0 + 1 \bmod 2^3$. The result is taken modulo 2^3 because due to reversibility, the maximum value of $2^3 - 1$ must wrap around to 0 upon increment. Figure 7.11b illustrates how the incrementer is extended to any number of qubits. Figure 7.11c then demonstrates how, by adding an additional control qubit to each individual gate making up the incrementer, a *controlled* incrementer is produced. As the name suggests, the controlled incrementer only increments the register $a_n \ldots a_2a_1a_0$ if



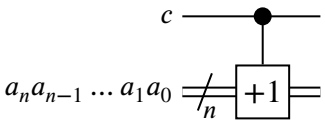
(a) A three-qubit incrementer.



(b) General n-qubit incrementer.



(c) Controlled three-qubit incrementer.



(d) Schematic symbol for the controlled incrementer.

Figure 7.11: Incrementer circuits.

the control qubit c is in the state $|1\rangle$. Figure 7.11d depicts the schematic symbol that we will use from now on to compactly represent a controlled incrementer. Observe that the lower line in this schematic represents not a single qubit but an entire register or array of n qubits.

Using controlled incrementers, Figure 7.12 illustrates how a quantum counter is formed by a sequence of controlled incrementers all acting on the same target register. This register stores a running count which will be incremented once for each control qubit in the state $|1\rangle$. Since the register is initialized to $|000\rangle$, the final value on the register is simply the total number of control qubits in the state $|1\rangle$, represented as a base-two integer as before. We observe that the quantum counter depicted here is limited to a maximum of seven control qubits because the maximum value of the three-qubit target register is $|111\rangle$. Attempting to count further than this would simply result in the counter wrapping around back to $|000\rangle$, as previously mentioned. This limit can be raised by increasing the size of the target register; a

target register consisting of n qubits will allow the quantum counter to count up a maximum value of $2^n - 1$.

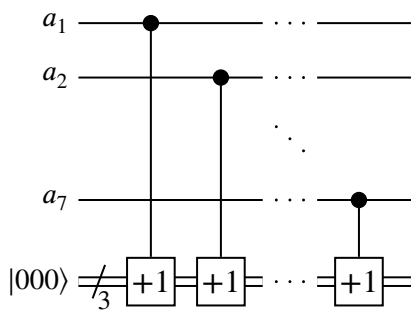


Figure 7.12: An example of a quantum counter.

With quantum counters, it is easy to construct a quantum circuit that calculates the total cost of an encoding. We recall that cost is equal to the total number of dependencies of each Q_i^+ on state and input variables Q_j and x_k, summed over all i. Therefore, we create a quantum circuit consisting of many dependency checkers, one to evaluate each possible dependency, where the outputs of the dependency checkers are fed to a quantum counter that counts the total number of dependencies. The final value of the quantum counter then gives the cost of the encoding represented by the input qubits. Figure 7.13 depicts the structure of the resulting circuit.

In Figure 7.13, each block labeled "D.C. Q_i^+, Q_j" or "D.C. Q_i^+, x_j" represents a dependency checker that checks whether Q_i^+ depends on Q_j or x_j, respectively. Due to space constraints, only a few checkers are shown in Figure 7.13, but the full circuit requires dependency checkers for every possible combination of a Q_i^+ and a Q_j or x_j. In other words, the circuit contains dependency checkers for Q_1^+ depending on each of Q_1 through Q_n and x_1 through x_m, Q_2^+ depending on each of Q_1 through Q_n and x_1 through x_m, and so on up to Q_n^+ depending on each of Q_1 through Q_n and x_1 through x_m, where n is the number of state variables and m is the number of input variables. Thus, the total number of dependency checkers

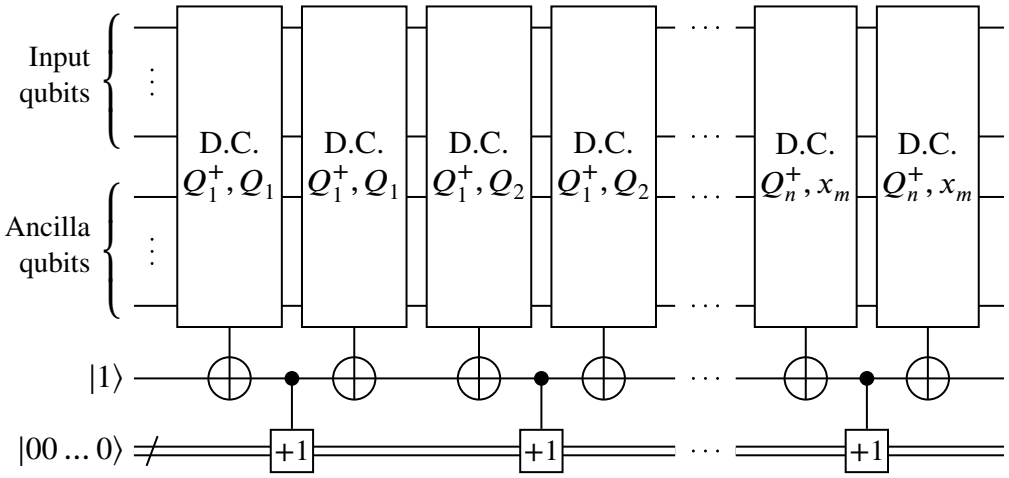


Figure 7.13: Quantum circuit to calculate total cost using a quantum counter.

is $n(n + m)$. This number is actually quite small relative to the size of the state machine being encoded, since the number of state and input variables grow only logarithmically with the number of states and input values, respectively, in the FSM.

We additionally observe that the quantum counter used in Figure 7.13 is constructed slightly differently from the example shown in Figure 7.12. Specifically, the quantum counter in Figure 7.13 is able to count using only one control qubit, while the one in Figure 7.12 counts the number of ones present in a whole collection of control qubits. The reason for this difference is that the circuit in Figure 7.13 counts the number of ones not in a collection of qubits, but appearing on a single qubit at different times. In other words, the circuit places the result of a given dependency checker onto the single counter control qubit and uses it to update the counter, but crucially, the circuit then applies the dependency checker again to restore that control qubit to its original state so that the next dependency checker can also use it to update the counter as well. In this way, the circuit is able to count dependencies without using a separate ancilla qubit to store the result of each dependency checker.

As previously mentioned, the maximum value that a quantum counter can reach before wrapping around to zero is determined by the number of qubits in the counter's target register. In Figure 7.13, the size of the quantum counter's target register is not explicitly indicated because it depends on the maximum possible cost. The maximum possible cost is equal to the total number of possible dependencies, which as we previously saw is $n(n+m)$ where n and m are the number of states and input values, respectively. Therefore, the quantum counter's target register must contain at least $\lceil log_2(n(n+m)+1) \rceil$ qubits to guarantee that no wrap-around can occur, which would cause the circuit to produce an incorrect result.

7.4.4 Quantum threshold circuit

With a quantum circuit for calculating the cost of an encoding, we now add a *threshold circuit* to create a quantum oracle that determines whether the cost of the encoding is less than r, where r is the threshold for which the oracle is being generated. The threshold circuit accepts a set of qubits representing a base-two integer as input and produces an output that depends on whether the input is less than or equal to r. Designing such threshold circuits is a well-known and solved problem in classical digital logic. For instance, the following recursive procedure allows one to generate a logical expression that determines whether a given base-two integer $(a_n a_{n-1} \ldots a_1 a_0)_2$ is less than or equal to a threshold $(r_n r_{n-1} \ldots r_1 r_0)_2$:

1. If the threshold and value to be compared against it consist of only a single bit, the expression is $\neg a_0$ if $r_0 = 0$ and a constant 1 if $r_0 = 1$. In this case, stop immediately as we are finished.

2. Otherwise, recursively use this procedure to generate a logical expression that determines whether $(a_{n-1} \ldots a_0)_2 \leq (r_{n-1} \ldots r_0)_2$.

3. If $r_n = 1$, take the logical OR of $\neg a_n$ with the expression generated in step 2. Otherwise, if $r_n = 0$, take the logical AND. Return the resulting expression as the output of this procedure.

4. At the very end, when all recursive steps are complete, it may be possible to simplify the expression using the identities $x \vee 1 = 1$ and $x \wedge 1 = x$.

For example, suppose we wish to generate a logical expression that determines whether $a_2 a_1 a_0$ is less than or equal to 5, whose base-two representation is 101. Then, since $r_2 = 1$, we take the logical OR of $\neg a_2$ with an expression recursively generated to determine whether $a_1 a_0$ is less than or equal to $(01)_2 = 1$. Since $r_1 = 0$, we take the logical AND of $\neg a_1$ with the expression that determines whether a_0 is less than 1. This is the base case for which the procedure above returns a constant 1. Therefore, the complete generated expression is $\neg a_2 \vee (\neg a_1 \wedge 1)$, which simplifies to $\neg a_2 \vee \neg a_1$.

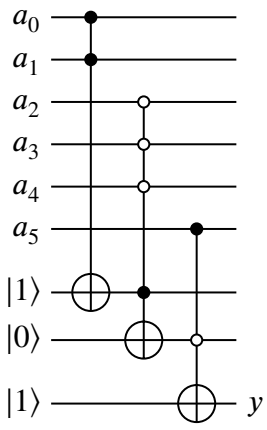


Figure 7.14: Quantum circuit implementing the expression $y = \neg a_5 \vee \big(\neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge (\neg a_1 \vee \neg a_0)\big)$.

Once a logical expression is obtained, constructing a quantum threshold circuit is simply a matter of implementing that logical expression with quantum gates. This can be achieved using a cascade of Toffoli gates as shown in Figure 7.14, where consecutive logical operations

of the same type (either AND or OR) can be combined into a single Toffoli gate to reduce the number of ancilla qubits required. We observe that the threshold is "hard-coded" into the circuit (*i.e.*, it is built into the circuit structure itself and can only be changed by changing the circuit) and therefore, generating a quantum oracle for a different threshold involves generating a new threshold circuit, as discussed in Section 7.2.

7.4.5 Quantum circuit to enforce bijectivity of encodings

In Section 7.3.2, we saw that it is necessary for all encodings to be bijective, which implies that no two states or inputs of an FSM may be encoded by the same value. Therefore, the quantum oracle must include a circuit to check for and rule out encodings where the same encoded value is used more than once. This can be achieved by comparing the encoded values for every possible pair of states/inputs and verifying that the encoded values are different for every such pair. We therefore construct the circuit shown in Figure 7.15. Since there are four states in this figure, six pairs of states must be checked, of which three are shown. Similarly, checks for three out of the six pairs of inputs are shown, with the understanding that the full circuit requires checks on all six. A mirror circuit (not shown) is also required to restore the ancilla qubits to their original states. Observe that the exact same circuit applies for checking both state and input encodings, and that although Figure 7.15 assumes four states, the same structure may of course be scaled up for FSMs with any number of states or inputs.

7.4.6 The complete quantum oracle

Finally, we consider how the quantum circuits we have demonstrated thus far are assembled to form a complete quantum oracle. Recall that the cost calculation circuit (Section 7.4.3),

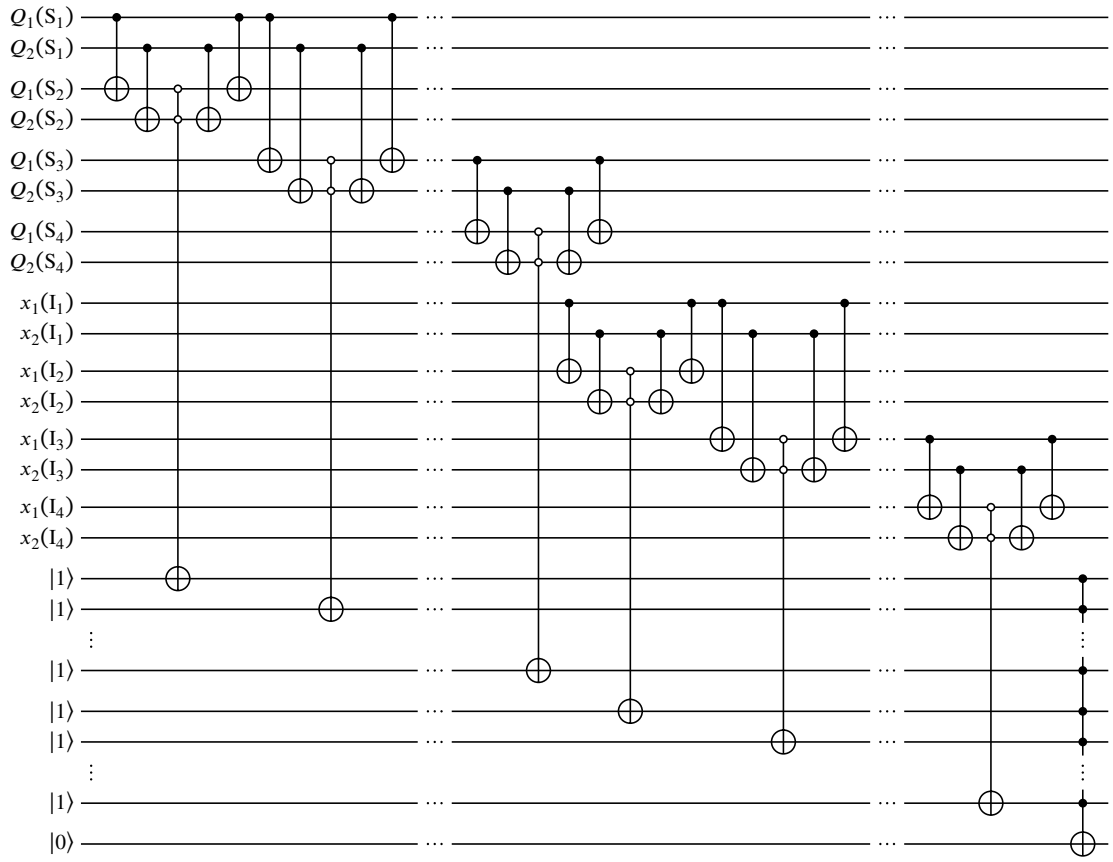


Figure 7.15: Quantum circuit to verify uniqueness of encoding for each state/input.

which is itself assembled using a quantum counter and dependency checkers outputs the cost of an encoding expressed as a base-two integer, which is then passed to the threshold circuit (Section 7.4.4). The threshold circuit produces a single-qubit answer indicating whether the calculated cost is below the threshold or not. At the same time, an encoding must be bijective in order to be considered at all. This condition is enforced by a circuit that checks uniqueness of each state or input's encoding (Section 7.4.5). We therefore see that our quantum oracle should only output 1 (true) if *both* the threshold and uniqueness checking circuits output 1. The complete quantum oracle therefore requires one additional Toffoli gate to produce its final output, as shown in Figure 7.16.

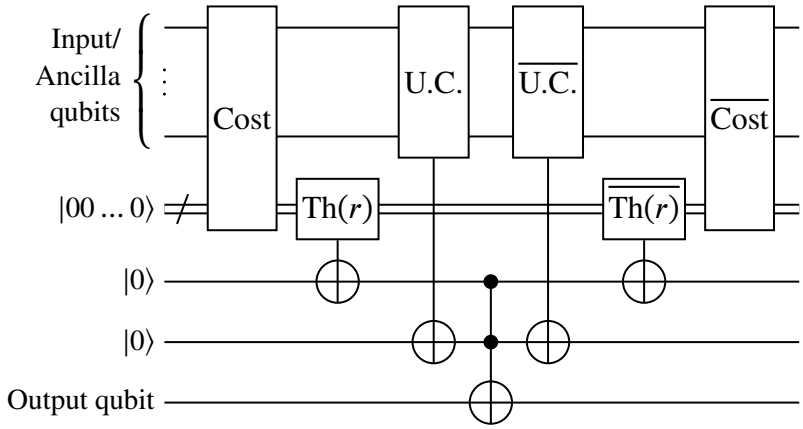


Figure 7.16: High-level view of the quantum oracle.

In Figure 7.16, the block labeled "Cost" denotes the cost calculation circuit, "Th(r)" denotes a threshold circuit with threshold r, and "U.C." ("uniqueness checker") denotes the uniqueness checking circuit. The oracle also requires additional mirror circuits to restore the ancilla qubits to their original values. These are denoted by overbars; *e.g.*, "$\overline{\text{U.C}}$" denotes the mirror circuit for the uniqueness checker. In particular, the mirror of the cost calculation circuit contains decrementer instead of incrementer circuits; these decrementer circuits naturally arise from reversing the order of the gates in the incrementer circuits from Figure 7.11.

Observe that the design of the oracle allows us to omit additional mirror circuits that would be necessary if these subcircuits were being used as stand-alone circuits. For instance, as a stand-alone circuit, the threshold circuit shown in Figure 7.14 would require additional mirror gates (as shown in Figure 7.17) if one wished to preserve the states of the two ancilla qubits for later reuse. However, when used in the quantum oracle, these additional mirror gates become unnecessary because the Th(r) and $\overline{\text{Th}(r)}$ subcircuits already act as mirrors to each other. Hence, the circuit structure shown in Figure 7.14, without additional mirror

gates, can be used for both of these subcircuits (except, of course, that the $\overline{\text{Th}(r)}$ subcircuit is reversed compared to its counterpart). This simplification effectively halves the size of both of these subcircuits, as compared with their stand-alone form as seen in Figure 7.17. The exact same observation also applies to the uniqueness checking subcircuits U.C. and $\overline{\text{U.C.}}$—their stand-alone forms would require additional mirror gates on top of the circuit structure shown in Figure 7.15, which become unnecessary when the subcircuits are used as components of the oracle in Figure 7.16.

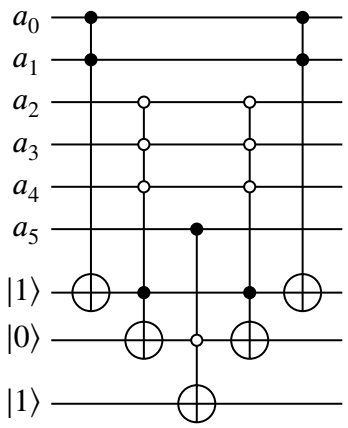


Figure 7.17: Threshold circuit from Figure 7.14 with mirror gates. The mirror gates turn out to be unnecessary as explained in the main text.

We observe that the threshold circuit is the only part of the oracle that must be regenerated for each run of Grover's algorithm as described in Section 7.2. The other parts of the oracle are generated depending on the FSM being encoded, but are independent of the chosen threshold; thus, they remain unchanged throughout the entire search procedure described in Section 7.2. With this oracle,[7] we have met the objective stated at the end of Section 7.2 and therefore, the procedure described in Section 7.2 gives a complete algorithm for finding

[7]To be more precise, we have described not a single oracle but a design according to which a whole sequence of oracles (for different thresholds) can be generated for any given FSM. This ability to generate a sequence of oracles is exactly what we need, as discussed in Section 7.2.

an exact optimal encoding of an FSM with the help of Grover's algorithm.

7.5 Run-time complexity analysis

7.5.1 Run-time complexity of the proposed quantum algorithm

We now wish to determine the run time complexity of our proposed quantum algorithm, in order to compare its performance against that of an analogous exhaustive search-based classical algorithm for the same problem. In this determination, we make certain simplifying assumptions, as described below. These simplifications are justified because our goal is not to perform the most detailed and nuanced analysis possible, but rather to show that our proposed algorithm can reasonably be expected to outperform the classical algorithm.

When computing the run time complexity of a quantum circuit, we must take into account the differing *quantum cost* [19] of the various quantum gates used in the circuit—in our case, these are CNOT, Toffoli, and multiple-control Toffoli gates. We recall that quantum gates are not physical hardware components like classical digital logic gates; instead, they are manipulations of the physical qubits, consisting of sequences of fundamental physical operations on those qubits. The quantum cost of a quantum gate is then the number of physical operations required to implement that gate, which roughly corresponds to the run time complexity of the gate if we assume that each fundamental physical operation requires approximately the same amount of time. Authors in the quantum computing literature have proposed a variety of quantum cost models, based on differing assumptions about the physical implementation of a quantum computing system and the fundamental operations available therein. For instance, a well-known result due to Barenco *et al.* [20] suggests[8]

[8]We say "suggests" because, although the results of Barenco *et al.* are widely used throughout the literature as a standard quantum cost function for multiple-control Toffoli gates, we do not know of a conclusive proof that it is impossible to implement a multiple-control Toffoli gate with lower cost complexity (assuming that no

that the quantum cost of a multiple-control Toffoli may be up to $\mathcal{O}(2^n)$, where n is the size of the gate. However, this result assumes that no ancilla qubits are used. With the use of ancilla qubits, it is easy to see that multiple-control Toffoli gates of arbitrary size may be implemented with a quantum cost that is linear with respect to the size of the gate. A cascade of Toffoli gates, as shown in Figure 7.18, accomplishes this task.

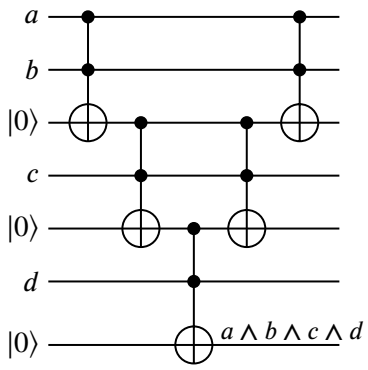


Figure 7.18: Implementation of a multiple-control Toffoli gate with quantum cost scaling linearly with gate size.

Throughout the following analysis, we will assume that an arbitrary number of ancilla qubits are available, and therefore the quantum cost of a multiple-control Toffoli gate is linear with respect to the gate's size. We make this assumption as it simplifies our calculations and is most conducive to our goal of obtaining a rough estimate of the run time complexity of our proposed quantum algorithm. The question of how the run time complexity varies in other quantum cost models is an interesting one, but outside the scope of the present paper. We leave further investigation of this question open for future work.

Our algorithm operates under the condition that the numbers of states and input values of the FSM must both be powers of two. Let us denote the number of states by N_S and the number of input values by N_I. Then, $N_\mathrm{S} = 2^{n_\mathrm{S}}$ and $N_\mathrm{I} = 2^{n_\mathrm{I}}$ for some nonnegative

ancilla qubits are used). In any case, our results are unaffected since we assume instead that ancilla qubits are used to implement multiple-control Toffoli gates with $\mathcal{O}(n)$ quantum cost, as illustrated in the main text.

integers n_S and n_I, where n_S and n_I are respectively the number of state and input variables. Conversely, $n_S = \log_2 N_S = \mathcal{O}(\log N_S)$ and $N_I = \log_2 N_I = \mathcal{O}(\log N_I)$. We first consider the run time complexity for the dependency checking quantum circuits described in Sections 7.4.2 and 7.4.3. Calculation of this complexity is complicated by the fact that the partial dependency checkers described in Section 7.4.2 do not have a fixed complexity; rather, as discussed in that section, their internal structure depends on the particular details of the state machine under consideration. However, we may still calculate an upper bound for the complexity of each partial dependency checker.

In the following paragraph, we use the example circuits shown in Figures 7.5, 7.7, and 7.8 as a reference. Evaluating $D_j(S, S')$ for a pair of states requires $\mathcal{O}(n_S) = \mathcal{O}(\log N_S)$ time, since $n_S - 1$ CNOT gates are required together with a multiple-control Toffoli gate of size n_S ($n_S - 1$ control qubits and one target qubit). By analogous reasoning, evaluating $D_j(I, I')$ requires $\mathcal{O}(n_I) = \mathcal{O}(\log N_I)$ time. Meanwhile, evaluating $A_i(S, S')$ is an $\mathcal{O}(1)$ operation, since only a single CNOT gate is needed. Observe that the upper portion of every partial dependency checker (as in Figures 7.7 and 7.8) must evaluate $A_i(S, S')$ for up to $N_S - 1$ state pairs, with this maximum being obtained when all state pairs are overlapping, as in (S_1, S_2), (S_2, S_3), (S_3, S_4), etc. The lower portion of each partial dependency checker always evaluates $D_j(S, S')$ or $D_j(I, I')$ for only one pair of inputs/states. The total complexity of a partial dependency checker is therefore $(N_S - 1) \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(\log N_S) = \mathcal{O}(N_S + \log N_S) = \mathcal{O}(N_S)$, for a partial dependency checker that evaluates $D_j(S, S')$ for a pair of states and checks (7.7), or $(N_S - 1) \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(\log N_S) = \mathcal{O}(N_S + \log N_I)$, for a partial dependency checker that evaluates $D_j(I, I')$ for a pair of inputs and checks (7.8).

From Figure 7.10 we can see that checking the dependencies of the next state of a *single* flip-flop on a *single* state or input variable requires either $N_S(N_S - 1)/2$ (for a state variable)

or $N_I(N_I - 1)/2$ (for an input variable) partial dependency checkers. Combining with the previously derived complexity for a single partial dependency checker gives $\mathcal{O}({N_S}^3)$ (for a state variable) or $\mathcal{O}\big({N_I}^2(N_S + \log N_I)\big)$ (for an input variable). The central Toffoli gate in Figure 7.10 has either $N_S(N_S - 1)/2$ or $N_I(N_I - 1)/2$ control qubits, so its time complexity is dominated by that of the partial dependency checkers.

In Figure 7.13, we can see that there are ${n_S}^2 = (\log N_S)^2$ checks for dependency of Q_i^+ (for any i) on a state variable, and $n_S n_I = \log N_S \log N_I$ checks for dependency on an input variable. Therefore, the total complexity of all the dependency checkers in the circuit from Figure 7.13 is

$$\begin{aligned}(\log N_S)^2 \cdot \mathcal{O}({N_S}^3) + \log N_S \log N_I \cdot \mathcal{O}\big({N_I}^2(N_S + \log N_I)\big) \\ = \mathcal{O}\big({N_S}^3(\log N_S)^2 + {N_I}^2(N_S + \log N_I)(\log N_S \log N_I)\big).\end{aligned} \tag{7.14}$$

The complexity of the incrementer circuits in Figure 7.13 is clearly dominated by that of the dependency checkers, so the preceding expression gives the time complexity for the entire cost-calculating portion of the quantum oracle.

Finally, we must consider the complexity of the two other components of the oracle, as shown in Figure 7.16—the threshold circuit and uniqueness-checking circuit. The complexity of the cost-calculating circuit clearly dominates that of the threshold circuit, but the situation with respect to the uniqueness-checking circuit is less clear. As seen in Figure 7.15, the uniqueness-checking circuit performs a comparison for every possible pair of states, of which there are $N_S(N_S - 1)/2$, and every possible pair of input values, of which there are $N_I(N_I - 1)/2$. Furthermore, a comparison of a pair of states has complexity $\mathcal{O}(n_S) = \mathcal{O}(\log N_S)$, since each individual state variable assignment must be compared between the two states, and a comparison of a pair of inputs similarly has complexity $\mathcal{O}(n_I) = \mathcal{O}(\log N_I)$.

The final Toffoli gate in Figure 7.15 has $N_S(N_S - 1)/2 + N_I(N_I - 1)/2$ control qubits, giving the entire uniqueness-checking circuit a complexity of

$$\begin{aligned} &\frac{N_S(N_S-1)}{2} \cdot \mathcal{O}(\log N_S) + \frac{N_I(N_I-1)}{2} \cdot \mathcal{O}(\log N_I) \\ &\qquad + \mathcal{O}\left(\frac{N_S(N_S-1)}{2} + \frac{N_I(N_I-1)}{2}\right) \\ &= \mathcal{O}({N_S}^2 \log N_S + {N_I}^2 \log N_I). \end{aligned} \tag{7.15}$$

Comparison with (7.14) shows that the cost-calculating circuit dominates the uniqueness-checking circuit in terms of complexity. Therefore, the run time complexity of the entire quantum oracle is still given by (7.14).

As we recall from Section 7.2, our proposed quantum algorithm involves executing Grover's algorithm multiple times to find an optimal encoding for an FSM. It is uncertain exactly how many executions of Grover's algorithm are required, as we do not consider the details of the search procedure used to find the minimum cost. However, we may consider the time complexity of just a single execution of Grover's algorithm. This does not affect our ability to compare the performance of quantum and classical algorithms because they both must perform a search procedure as described in Section 7.2 in order to find the minimum possible cost.

We discussed in Chapter 1, Section 1.4 how Grover's algorithm (together with its extensions for the case of multiple solutions) is able to find a solution to a satisfaction problem, or detect the non-existence of a solution, with $\mathcal{O}\left(\sqrt{M}\right)$ executions of the quantum oracle, where M is the size of the search space. (We have used M to avoid confusion with the numbers of states and input values.) As discussed in Section 7.4.1 and shown in Figure 7.4b, our proposed oracle requires $N_S n_S + N_I n_I = \mathcal{O}(N_S \log N_S + N_I + \log N_I)$ input qubits,

corresponding to a search space of size $\mathcal{O}\big(2^{N_\mathrm{S}\log N_\mathrm{S}+N_\mathrm{I}+\log N_\mathrm{I}}\big) = \mathcal{O}({N_\mathrm{S}}^{N_\mathrm{S}}{N_\mathrm{I}}^{N_\mathrm{I}})$. Hence, a single execution of Grover's algorithm requires $\mathcal{O}\Big(\sqrt{{N_\mathrm{S}}^{N_\mathrm{S}}{N_\mathrm{I}}^{N_\mathrm{I}}}\Big)$ executions of the quantum oracle. The total complexity of one execution of Grover's algorithm, taking into account the previously determined complexity of the oracle, is then

$$\mathcal{O}\Bigg(\sqrt{{N_\mathrm{S}}^{N_\mathrm{S}}{N_\mathrm{I}}^{N_\mathrm{I}}}\Big[{N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{I}}^2(N_\mathrm{S}+\log N_\mathrm{I})(\log N_\mathrm{S}\log N_\mathrm{I})\Big]\Bigg). \tag{7.16}$$

Eq. (7.16) is rather unwieldy for a simple, rough comparison between the performance of quantum and classical algorithms. If we assume that $N_\mathrm{S} = N_\mathrm{I}$, that is, the FSM has the same number of states as input values, then (7.16) simplifies to the much more manageable

$$\begin{aligned}
&\mathcal{O}\Bigg(\sqrt{{N_\mathrm{S}}^{N_\mathrm{S}}{N_\mathrm{I}}^{N_\mathrm{I}}}\Big[{N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{I}}^2(N_\mathrm{S}+\log N_\mathrm{I})(\log N_\mathrm{S}\log N_\mathrm{I})\Big]\Bigg) \\
&\qquad = \mathcal{O}\Big({N_\mathrm{S}}^{N_\mathrm{S}}\Big[{N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{S}}^2(N_\mathrm{S}+\log N_\mathrm{S})(\log N_\mathrm{S}\log N_\mathrm{S})\Big]\Big) \\
&\qquad = \mathcal{O}\Big({N_\mathrm{S}}^{N_\mathrm{S}}\Big[{N_\mathrm{S}}^3(\log N_\mathrm{S})^2\Big]\Big) \\
&\qquad = \mathcal{O}\big({N_\mathrm{S}}^{N_\mathrm{S}+3}(\log N_\mathrm{S})^2\big).
\end{aligned} \tag{7.17}$$

7.5.2 Comparison with a classical algorithm

A comparable classical algorithm to solve the FSM encoding problem would operate in much the same way as our proposed quantum algorithm, with the main difference being that a classical computer of course cannot use Grover's algorithm and must instead use a straightforward exhaustive search. In particular, a classical computer must also use eqs. (7.7) and (7.8) to calculate the cost of a given encoding. Most, if not all, modern digital computers are capable of performing so-called bitwise logical operations on words of significant length (at least 32 or 64 bits), which allows them to evaluate $D_j(\mathrm{S},\mathrm{S}')$ and $A_i(\mathrm{S},\mathrm{S}')$ for any pair of

states or $D_j(\mathrm{I}, \mathrm{I}')$ for any pair of inputs in $\mathcal{O}(1)$ time. (We need not consider state machines with more than 2^{32} states or input values, as such a machine would be far too large to be of practical interest.) It follows that a classical computer can evaluate (7.7) in $\mathcal{O}(N_\mathrm{I})$ time, since it requires iterating over all input values. Then, determining whether Q_i^+ depends on Q_j, for any values of i and j, requires evaluating (7.7) for all $N_\mathrm{S}(N_\mathrm{S}-1)/2$ pairs of states and therefore takes $\mathcal{O}({N_\mathrm{S}}^2 N_\mathrm{I})$ time. Similarly, a single evaluation of (7.8) requires $\mathcal{O}(N_\mathrm{S})$ time and determining whether Q_i^+ depends on x_k, for any values of i and k, requires $\mathcal{O}(N_\mathrm{S}{N_\mathrm{I}}^2)$ time.

Calculating the total cost of an encoding involves checking the dependency of Q_i^+ on Q_j for all combinations of i and j, of which there are $(\log_2 N_\mathrm{S})^2$, and the dependency of Q_i^+ on x_k for all combinations of i and k, of which there are $(\log_2 N_\mathrm{S})(\log_2 N_\mathrm{I})$. Therefore, the complete calculation of the cost of an encoding on a classical computer requires $\mathcal{O}\big({N_\mathrm{S}}^2 N_\mathrm{I}(\log N_\mathrm{S})^2 + N_\mathrm{S}{N_\mathrm{I}}^2(\log N_\mathrm{S})(\log N_\mathrm{I})\big)$ time. The number of possible encodings is $N_\mathrm{S}!N_\mathrm{I}!$, so a full exhaustive search on a classical computer has a total time complexity of

$$\mathcal{O}\Big(N_\mathrm{S}!N_\mathrm{I}!\big[{N_\mathrm{S}}^2 N_\mathrm{I}(\log N_\mathrm{S})^2 + N_\mathrm{S}{N_\mathrm{I}}^2(\log N_\mathrm{S})(\log N_\mathrm{I})\big]\Big). \tag{7.18}$$

Just as for the quantum algorithm, we can simplify this expression if we assume that $N_\mathrm{S} = N_\mathrm{I}$; in this case, (7.18) reduces to

$$\begin{aligned}&\mathcal{O}\Big(N_\mathrm{S}!N_\mathrm{S}!\big[{N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{S}}^3(\log N_\mathrm{S})(\log N_\mathrm{S})\big]\Big)\\ &\qquad = \mathcal{O}\big((N_\mathrm{S}!)^2{N_\mathrm{S}}^3(\log N_\mathrm{S})^2\big).\end{aligned} \tag{7.19}$$

Comparing eqs. (7.17) and (7.19), we see that the factor of ${N_\mathrm{S}}^3 \log N_\mathrm{S}$ is common to both. We may therefore compare the relative complexities of the quantum and classical

Table 7.1: Comparison of relative complexities of quantum and classical algorithms, assuming $N_S = N_I$.

N_S	${N_S}^{N_S}$ (Quantum)	$(N_S!)^2$ (Classical)	$\frac{(N_S!)^2}{{N_S}^{N_S}}$
4	2.56×10^2	5.76×10^2	2.25
8	1.68×10^7	1.63×10^9	96.9
16	1.84×10^{19}	4.38×10^{26}	2.37×10^7
32	1.46×10^{48}	6.92×10^{70}	4.74×10^{22}

algorithms, if $N_S = N_I$, by looking only at the terms ${N_S}^{N_S}$ (for the quantum algorithm) and $(N_S!)^2$ (for the classical algorithm). Table 7.1 shows the results for a few values of N_S. We immediately see that the quantum algorithm appears to be significantly faster than the classical algorithm. Care must be taken in this comparison because we are only comparing the relative complexities of the two algorithms and not their actual run times. In particular, the actual ratio of run times between the two algorithms is better approximated as $C(N_S!)^2/{N_S}^{N_S}$, where C is an unknown constant. For example, $C = 1/1000$ indicates, very roughly speaking, that the classical computer's "clock speed"—by which we mean not necessarily the hardware's physical clock speed, but rather the number of low-level instructions executed per second—is on the order of 1000 times faster than the quantum computer's. From Table 7.1, we can see that for $N_S = 16$, our proposed quantum algorithm is expected to be faster even if the quantum computer on which it is running has a clock speed a million times slower than the competing classical computer. For $N_S = 32$, our proposed quantum algorithm will, nearly unquestionably, run many orders of magnitude faster than the comparable classical algorithm using an exhaustive search. This gives us a high degree of confidence in our expectation that our proposed quantum algorithm will outperform the classical algorithm for state machines of reasonable size.

It is interesting to observe that the quantum algorithm actually contains a significant inefficiency in comparison to the classical algorithm. The inefficiency arises from the fact that Grover's algorithm can only search through a space consisting of all possible binary strings of a given length. Therefore, the quantum algorithm searches through all possible combinations of inputs to the oracle as illustrated in Figure 7.4b, even those that do not represent a valid encoding. This means that a large portion of the search space is effectively extraneous, and is excluded by the uniqueness checking circuit from Figure 7.15. The classical algorithm has no such difficulty as it can simply search through only the set of valid encodings, and is not forced to search through a space of all possible binary strings of a given length, as Grover's algorithm is. The fact that the quantum algorithm still outperforms the classical algorithm, with a lower run time complexity, shows that this disadvantage is outweighed by the quadratic speedup in searching obtained from using Grover's algorithm.

7.6 Conclusion

We presented a quantum algorithm for finding an exact solution to the problem of encoding a finite state machine with the lowest cost possible. Specifically, our algorithm finds an optimal encoding for any FSM with numbers of inputs and states that are powers of two, under the assumptions that the number of state and input variables must be the smallest possible and that the cost of an encoding is given by the total number of variables on which the encoded transition functions resulting from that encoding depend. Our algorithm contains the following notable features:

1. It uses a quantum computer with Grover's algorithm as a subroutine to perform exhaustive searches with lower time complexity than that which is achievable using a classical computer alone, thus making those exhaustive searches more practical. Little

to no published work exists on the subject of applying Grover's algorithm to directly solve a practically useful problem, and the present work is the first to apply Grover's algorithm to the problem of finding optimal encodings, based on the simple metric of dependencies for completely specified finite state machines where both the number of states and the number of input values are a power of 2.

2. It simultaneously optimizes both state and input encodings for an FSM. Currently, [83] is the only other published method that finds exact minimum solutions; however, [83] only solves the problem of state, and not input, encoding. Additionally, the method presented in [83] is specialized for FSMs implemented using PLAs because it minimizes the total number of PLA product terms. In comparison, we use the cruder but more generally applicable cost metric of the total number of variables on which a function depends.

3. It uses Grover's algorithm to solve an optimization, rather than satisfaction, problem. It achieves this by solving a *sequence* of satisfaction problems using Grover's algorithm; each such satisfaction problem is of the form "find an encoding with cost at most r" where r is a threshold that is varied. By repeatedly executing Grover's algorithm for different thresholds, where the threshold is varied according to an appropriate strategy (*e.g.*, a binary search strategy), the algorithm eventually finds the exact minimum possible cost and an encoding with that cost.

4. We introduced the concept of using a quantum counter in tandem with a threshold circuit as part of a quantum oracle. Such oracles check whether the value of a certain function (in this case, our cost function) lies below a given threshold and are exactly what is needed to solve an optimization problem using the procedure described in

Section 7.2. The use of quantum counters and threshold circuits in quantum oracles is not limited to solving the FSM encoding problem. It can be applied to many other optimization problems such as MAXSAT, in which the objective is to satisfy as many terms of a Boolean expression as possible.

We compared the run time complexity of the proposed quantum algorithm against that of the analogous exhaustive search-based classical algorithm. This analysis does not tell us the absolute run times of either algorithm, as calculating such would require much more detailed information regarding the precise specifications of the quantum and classical computers being used. Nevertheless, the comparison of run time complexities provides strong evidence that the quantum algorithm can significantly outperform the classical algorithm for FSMs of reasonable size that might be encountered in practice.

In addition, our work may serve as the basis for further investigation in a number of different directions. We leave these possibilities open to future exploration. Among them, the most promising include:

Incompletely specified transition functions—a significant limitation of our method is that it requires all encoded transition functions to be completely specified, which means that it is only applicable to FSMs with a number of states that is a power of two. Extending our method for encoded transition functions that are incompletely specified would allow it to apply to all FSMs.

Output encodings—in a realistic digital logic design scenario, the outputs of a state machine of course cannot be ignored. We believe that the methods presented here can, without too much difficulty, be extended to the problem of encoding outputs of FSMs as well. Such an extension would represent the first quantum algorithm to solve the FSM encoding problem simultaneously for states, inputs, and outputs.

A more detailed cost model—we used a simple cost model which only takes into account the number of dependencies of each encoded transition function on state and input variables. While this simple model possesses the advantage of not being closely tied to a single digital logic implementation technology, it is still clearly desirable to extend our method to more realistic cost models that take into account additional factors.

Comparison of threshold search strategies—one of the key elements of our method is the execution of Grover's algorithm multiple times with a sequence of quantum oracles generated for different thresholds as described in 7.2. We did not attempt to compare different strategies for varying the threshold. Such a comparison could significantly improve our algorithm, because the selected strategy affects the expected number of Grover runs needed to find the minimum possible threshold, which in turn affects the run time of our entire algorithm. An analysis of threshold search strategies would also be applicable to problems other than FSM encoding (see also the following paragraph).

Application to other problems—as mentioned before, the principle of solving an optimization problem by running Grover's algorithm multiple times using a sequence of quantum oracles can be applied to other problems. One such problem, for example, is MAXSAT, where the objective is to maximize the number of simultaneously satisfied clauses in a Boolean formula expressed in conjunctive normal form. Further investigation into this and other such problems would greatly increase the generality of the method presented here.

Chapter 8

A quantum algorithm for state-space path planning problems

Chapter 7 demonstrated the detailed design of a quantum oracle to be used with Grover's algorithm to solve a problem of practical interest, namely the problem of finding optimal encodings for finite state machines. In this chapter, I present a strategy or approach for the design of quantum oracles targeted at another class of problems with practical applications, which I call *state-space path planning* problems. These types of problems can be informally summarized as involving a system with a number of states, where the objective is to find one's way to a desired ending state from a specified starting state while only making certain allowed moves from state to state.

In Section 8.1, I first introduce the concept of a state-space path planning problem in more detail. I introduce two examples of such problems, which will be used to demonstrate aspects of the general oracle-designing strategy. The first such problem is to solve a tile- or block-sliding puzzle commonly known as the "8-puzzle" or "15-puzzle". In this puzzle, the player is presented with a 3×3 or 4×4 square grid containing tiles or blocks labeled from 1 to 8 or 1 to 15, depending on the size of the grid, with one empty space. The second problem I consider is a labyrinth-solving problem where the player must find a path between two designated locations. This problem may be complicated by the addition of obstacles such as a closed door that the player cannot pass unless they first activate a switch that opens

the door, located elsewhere in the labyrinth.

In Section 8.2, I first give a high-level overview of the structure of the quantum oracle that will be used with Grover's algorithm to solve state-space path planning problems. Specifically, I use a type of oracle based on a *quantum state machine*, which simulates a sequence of moves and checks if the desired ending state has been reached after those moves. This quantum state machine is based on a single subcircuit, the next-state circuit, which is simply repeated as many times as necessary to simulate a desired number of moves. In Sections 8.3 and 8.4, I then consider the design of quantum oracles in more detail for each of the two problems described above.

Finally, in Section 8.5, using the concepts and principles introduced in the previous examples, I define a general approach to designing a quantum oracle for any state-space path planning problem that meets a few requirements. Several problems with applications in robotics and machine learning are seen to meet these requirements and can therefore be solved using this approach.

8.1 State-space path planning problems

A state-space path planning problem conforms to the following description:

- One has a set of states together with a set of moves, which describe the allowed transitions between these states, and the objective is to reach a desired ending state from a specified starting state in as few moves as possible.

- The moves should have some degree of uniformity. Uniformity in this context is defined as having a single set of moves that remains the same regardless of the current state. In other words, the available moves should appear the same from state to state.

Some exceptions are permitted: certain moves may be prohibited for certain states only, so that for those states, the available moves are only a subset of the normal set of moves.

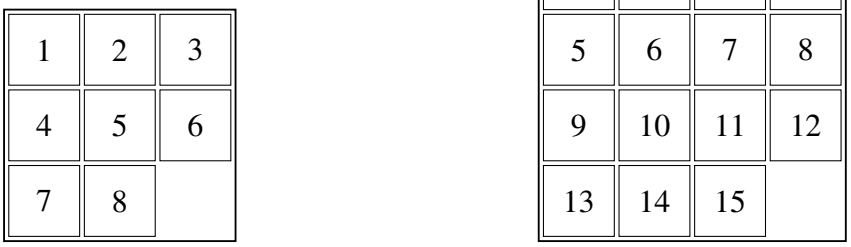
- The function that maps the current state and chosen move to the next state should be reversible. Knowledge of the current state and the previous move that was made to reach the current state should be sufficient to determine the previous state. Certain exceptions, discussed in Section 8.4.2, are permitted to this requirement as well.

This description is intentionally left slightly vague. In particular, the uniformity requirement is subject to interpretation as to the precise definition of moves "appearing the same". This is because the initial step of formulating a problem of interest in a manner that follows the above description requires the ingenuity of a human designer and cannot be done in a strictly formulaic way. It is ultimately only important that a problem has been formulated in a way that makes it clear how the approach described in this chapter can be applied. The following sections will provide context and motivation for the uniformity requirement and will also give an intuitive idea of what is needed to conform to this requirement. In Section 8.5, assisted by the examples presented in Sections 8.3 and 8.4, I give a more precise description of the types of problems that can be solved using the concepts introduced by those examples.

8.1.1 Example: the 8-puzzle and 15-puzzle

Figure 8.1 depicts a well-known puzzle involving tiles or blocks that can slide within a square grid. In theory, the square grid may be of any size, but I will consider the 3×3 and 4×4 variants, which are the most well known. The tiles are numbered sequentially starting from 1 and there is always one less of them than the number of squares in the grid, therefore

leaving one empty space. The objective of the puzzle is to slide the tiles to reach the "home position" depicted in either Figure 8.1a (for the 3×3 variant) or Figure 8.1b (for the 4×4 variant), where the tiles are in sequential order from left to right and top to bottom with the empty space in the lower right corner. The problem of finding the shortest solution to the $n \times n$ version of this puzzle is known to be NP-hard [87, 88], and it has therefore been used as a test case for heuristic search algorithms [89].



(a) 3×3 variant (8-puzzle). **(b)** 4×4 variant (15-puzzle).

Figure 8.1: Examples of the $(n^2 - 1)$-puzzle.

Both the 3×3 and 4×4 variants of the puzzle can be formulated in terms of the description given at the start of Section 8.1. The set of states consists of all possible arrangements of tiles within the grid. This is equivalent to the set of all permutations of 9 or 16 objects for the 3×3 and 4×4 variants, respectively. For instance, any possible position of the 3×3 puzzle can be described by a permutation of the integers 0 through 8, where 1 through 8 represent their respective tiles and 0 represents the empty space.

For any state of either puzzle, there are up to four possible moves, because one can choose to slide any of the four tiles adjacent to the empty space into that space. This provides an example of uniformity in possible moves: regardless of where the empty space is currently located, and how the tiles are currently arranged, these four moves are the only ones that are ever possible. However, in some positions, only two or three moves are possible. If the

empty space is currently at an edge or in a corner, then there is no tile that can move into the empty space from the direction or directions in which it is adjacent to the boundary. For instance, if the empty space is at the right edge, no tile is available on the right, so only three moves are possible. This restriction on available moves is an example of an exception to uniformity: some of the four usually-possible moves are prohibited in certain states.

Finally, we may note that this formulation of the puzzle in terms of states and moves also satisfies the reversibility condition stated at the start of Section 8.1. Given the current state of the puzzle and the most recent move, it is easy to reconstruct the state of the puzzle prior to that move by simply undoing it. For instance, if the most recent move was to slide the tile above the empty space into that space, then this move is undone by sliding that same tile (which is now below the empty space) back up into its previous position.

8.1.2 Example: labyrinth with a closed door

The second example of a state-space planning problem that I consider in this chapter is that of solving the labyrinth depicted in Figure 8.2. The objective is to navigate from the starting point, labeled B[1] in Figure 8.2, to the ending point, labeled E. The solid lines represent walls while the dashed line is a door that is initially closed, but may be opened by means of a switch located at the position labeled S. Once the switch has been activated, the door will remain open indefinitely and can freely be passed through.

To formulate the labyrinth as a state-space path planning problem, we define a state to consist of two components: the current position in the labyrinth and whether or not the door has been opened. In order to keep the problem simple for demonstrative purposes, position can be discretized to a 4×4 grid so that there are 16 possible positions. A position may then

[1] B stands for "begin" because S has been used for "switch".

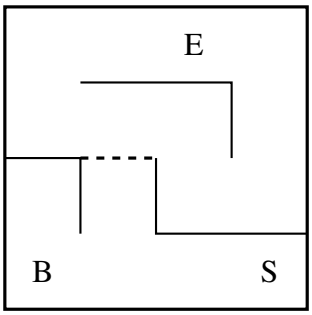


Figure 8.2: A labyrinth that must be solved by opening a closed door with a switch.

be defined by an x- and y-coordinate, each being an integer ranging from 0 to 3, as shown in Figure 8.3.

$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$
$(0,2)$	$(1,2)$	$(2,2)$	$(3,2)$
$(0,3)$	$(1,3)$	$(2,3)$	$(3,3)$

Figure 8.3: A coordinate system for the labyrinth from Figure 8.2.

With position being discretized, the possible moves then include moving one square in any of the four cardinal directions. In addition, there is a fifth move, which is to activate the switch and thus open the door. As with the 8-puzzle/15-puzzle, not all of the moves are available if one's current position is adjacent to a wall. In fact, in the particular labyrinth considered here, every square is adjacent to at least one wall, so there is no state in which it is possible to move in all four directions. The number of available directions may be as few as one if the current position is surrounded by three walls. The move of activating the switch is also only available under specific circumstances; namely, when one's current position is at the switch.

If we consider only movement moves (excluding the move of activating the switch), then the labyrinth problem satisfies the reversibility condition perfectly. Given a current position and the most recent move, that move can simply be undone to reconstruct the previous position. However, the switch-activating move is not reversible because the door is defined to remain open indefinitely once the switch has been activated. In other words, additional activations of the switch beyond the first have no effect. If the current state is position $(3, 0)$ (the location of the switch) and door open, and the most recent move was to activate the switch, we cannot determine whether the door was closed or open in the previous state, because this information does not tell us whether the switch was already activated earlier. If the switch was already activated earlier, then the door must be open in the previous state. If the switch was not activated earlier, then the activation in the most recent move was the first one and therefore the door must be closed in the previous state. This type of irreversible move receives special treatment when designing a quantum oracle to solve the problem using Grover's algorithm, as will be described in Section 8.4.2.

8.2 Outline of oracle design for state-space path planning problems

An outline of the circuit structure used to create quantum oracles for state-space path planning problems is shown in Figure 8.4. This type of oracle makes use of two subcircuits: a *next-state circuit*, labeled "NS", and an *ending-state-checking circuit*, labeled "= E". The oracle operates as a quantum state machine that computes the states visited by a sequence of moves, and uses the ending-state-checking circuit to determine whether the desired ending state has been reached. Specifically, one first chooses an encoding scheme for the states and moves of the problem to be solved. The inputs to the oracle—that is, the qubits whose possible states will be searched by Grover's algorithm—then represent a sequence of moves. These are

the groups of qubits labeled “Move 1” through “Move N” in Figure 8.4; each such group represents a single move according to the chosen encoding scheme.

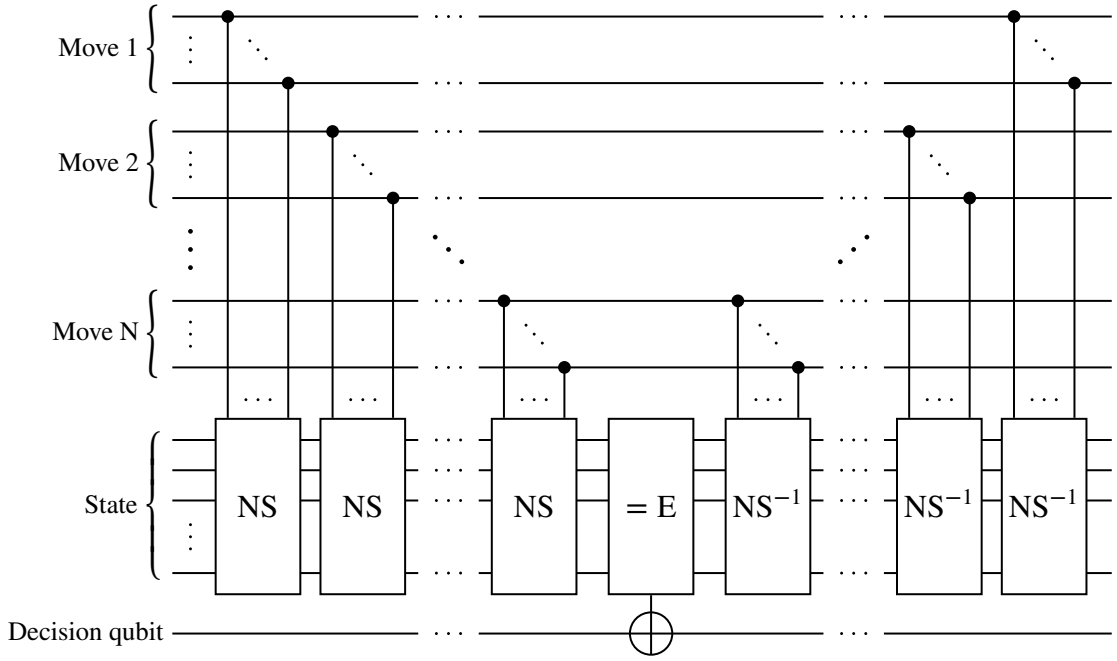


Figure 8.4: The basic circuit structure of quantum oracles for state-space path planning problems.

Given a sequence of moves, the oracle computes the states visited by that sequence of moves and determines whether the desired final state has been reached. It does this by using another group of qubits, labeled “State” in Figure 8.4, to represent and track the current state. These state-tracking qubits are initialized to values that represent the initial state of the problem, again according to the chosen encoding scheme. Each next-state circuit is identical, and a single such circuit is designed to take a current state together with a current move and compute the next state. In this way, the sequence of next-state circuits in Figure 8.4 computes the sequence of states visited by the specified moves. The next-state circuits are drawn using a special notation where the qubits representing the move look like control qubits, which however have separate connections to the “NS” block rather than a single

connection as in a Toffoli or other multiply-controlled gate. This is because the next-state circuit does not function as a controlled gate, with a single target gate that is either active or inactive. Instead, the circuit may perform multiple different actions depending on the current move, since each possible move may result in a different next state.

Following the sequence of next-state circuits, the state-tracking qubits contain a representation of the final state reached by the sequence of moves given to the oracle. The ending-state-checking circuit is designed to invert the oracle's decision qubit, indicating a correct solution, if and only if the desired ending state has been reached. Therefore, when used with Grover's algorithm, the oracle will find a move-sequence that successfully reaches the desired ending state, if such a sequence exists.

From the point of view of Grover's algorithm, the state-tracking qubits are simply ancillary or work qubits used by the oracle, so the oracle must ultimately restore them to their original state. The oracle does this by using a mirror circuit, which consists of the inverses of the next-state circuits applied in reverse order.

It is important to observe that an oracle of the type shown in Figure 8.4 does not directly solve a state-space path planning problem. Instead, when used with Grover's algorithm, the oracle only solves the more limited problem "find a path from the starting state to the ending state in N moves, if one exists". Grover's algorithm must then be executed multiple times, adjusting the value of N and creating a new oracle each time as described in Chapter 7, Section 7.2. The smallest value of N for which Grover's algorithm is able to find a path then gives the overall solution to the state-space path planning problem.

8.3 Design of quantum oracles for the 8- and 15-puzzles

8.3.1 Encoding scheme for states and moves

As discussed in Section 8.1.1, the possible states of the puzzle can be represented by permutations of the integers from 0 through 8 (for the 3×3 version) or 0 through 15 (for the 4×4 version). Such a permutation can in turn be represented with 9 or 16 groups of four qubits each, where each group of qubits is assigned to one of the cells in the grid and carries the base-two representation of the number of the tile in that cell. For instance, if the group of qubits assigned to the upper-left corner has values 0011, this indicates that tile 3 is currently in that corner. A value of 0 (base-two representation 0000) indicates the empty space. Each group consists of four qubits because that is the minimum number needed to represent all integers from 0 through 8 or 0 through 15. This encoding scheme can be extended to larger puzzles by using more qubits per group: a 5×5 puzzle would require 25 groups of five qubits each, five being the minimum number needed to represent all integers from 0 through 24. For future reference, I will refer to these qubits collectively as the *permutation qubits*. Each individual group of four qubits will be called a *tile register*, being a register of qubits that encodes the identity of the tile located at a particular cell.

In addition to the permutation qubits, an additional set of qubits is used to track the current position of the empty space. For this purpose we use the same coordinate system that was introduced for the labyrinth problem in Figure 8.3; in the 3×3 case, both the x- and y-coordinates have a maximum value of 2. Four qubits are then sufficient to represent the position of the empty space: one pair representing the x-coordinate and one pair representing the y-coordinate. I will refer to these qubits as the *position qubits*. The information carried by the position qubits is redundant because the state of the puzzle is already entirely determined

Table 8.1: Encoding of moves for the 8-puzzle/15-puzzle.

Move	Encoding $m_{i,1}m_{i,0}$	Action
Left	0 0	Decrement x-coordinate by 1
Right	0 1	Increment x-coordinate by 1
Up	1 0	Decrement y-coordinate by 1
Down	1 1	Increment y-coordinate by 1

by the permutation qubits, and in particular, the position of the empty space is determined since it is the one whose corresponding tile register has a value of 0000. However, the position qubits assist in the operation of the next-state circuit, as detailed in Section 8.3.2, because they allow for manipulation of the permutation qubits using a multiplexing/demultiplexing circuit design, which simulates a move of the puzzle where one tile is shifted.

The encoding of moves is substantially simpler since there are only four possible moves. Table 8.1 shows an encoding of the possible moves using two qubits, denoted $m_{i,1}$ and $m_{i,0}$. The subscript i indicates that the two qubits encode the i-th move, and it is included to emphasize that every move is encoded in an identical fashion with its own pair of qubits. In other words, for an N-move oracle, the encoding of the whole sequence of N moves consists of $2N$ qubits, labeled $m_{1,1}$, $m_{1,0}$, $m_{2,1}$, $m_{2,0}$, $m_{3,1}$, etc. up to $m_{N,0}$. The entries listed under the "Move" column of Table 8.1 represent the directions in which the empty space moves; the tile sliding into the position of the empty space then moves in the opposite direction. For instance, from the starting configuration of the 3×3 puzzle shown in Figure 8.1a, if tile 8 slides rightwards into the empty space, then the empty space moves leftwards to the original location of tile 8. In the scheme specified by Table 8.1, this move would be considered a leftwards move and would therefore be encoded as 00.

8.3.2 Design of the next-state circuit

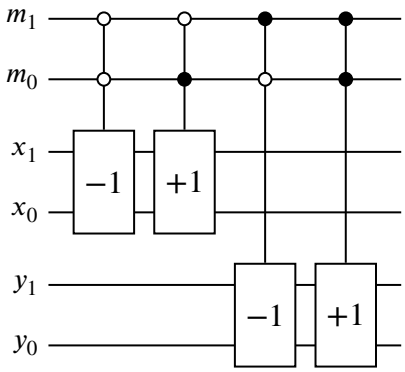
Based on the encoding scheme described in Section 8.3.1, we can see that the next-state circuit must accomplish two tasks. First, it must simulate the movement of a tile by swapping the contents of two tile registers: the one corresponding to the current location of the empty space and the one corresponding to the tile being moved. Second, the next-state circuit must also update the position of the empty space. We consider this second task first because it is simpler.

Since the empty space always moves to an adjacent cell with each move, exactly one of its two coordinates must change by either $+1$ or -1 in a single move. For instance, a movement of the empty space to the left is represented by decrementing its x-coordinate by 1. The "Action" column of Table 8.1 shows the action performed on the position qubits for all four possible moves. Based on this information, we can create the circuit shown in Figure 8.5a, which takes a move and performs the appropriate action on the position qubits. The position qubits are x_1, x_0, y_1, and y_0, where x_1x_0 and y_1y_0 are the base-two representations of the x- and y-coordinates, respectively. The circuit shown in Figure 8.5a uses controlled decrementers and incrementers to implement the actions specified in Table 8.1 in a case-by-case manner. For example, the first decrementer in Figure 8.5a acts on the x-coordinate and is active when both move qubits are 0, so it handles the case given by the first row of Table 8.1.

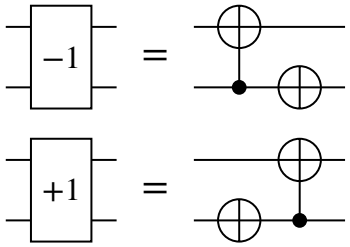
Using the implementations of a two-qubit decrementer and incrementer shown in Figure 8.5b, we can expand the circuit from Figure 8.5a in terms of CNOT and Toffoli gates, producing the circuit shown in Figure 8.5c. This circuit can in turn be simplified: the pairs of adjacent 3-control Toffoli gates share the same set of control qubits and differ only in the control polarity of a single qubit, so they collapse into 2-control Toffoli gates, giving

the circuit shown in Figure 8.5d. This is the *next-position circuit*, which computes the next position of the empty space given its current position and the current move.

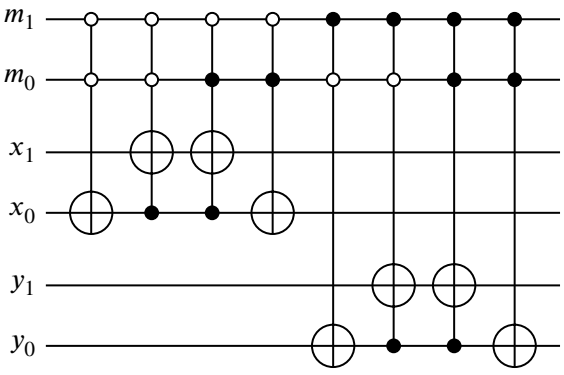
It should be noted that the circuit shown in Figure 8.5d is applicable to both the 3×3 and 4×4 variants of the puzzle. This follows from the fact that Table 8.1 applies to both variants, since the effect of each possible move on the x- and y-coordinates (and therefore on the position qubits) does not depend on the puzzle size.



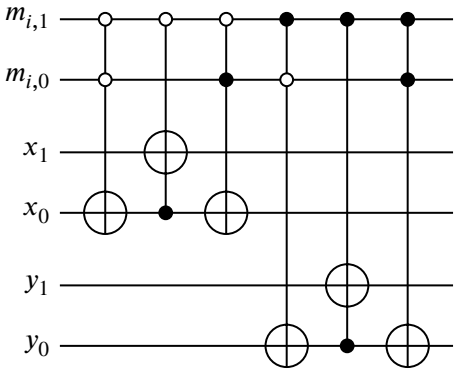
(a) Outline of the circuit using incrementers and decrementers.



(b) Implementation of an incrementer and decrementer.



(c) The circuit expanded in terms of Toffoli gates.



(d) The final circuit after simplification.

Figure 8.5: Design of a circuit for updating x- and y-coordinates according to a provided move.

The next-position circuit from Figure 8.5d is only one component of the next-state circuit, since the next-state circuit must also perform the first task stated at the beginning of this

section. Namely, the remaining part of the next-state circuit should update the tile registers to reflect the move that is being made. Figure 8.6 shows an example of how this operation on the tile registers is conceptually performed. In this figure, the *ad hoc* notation of two empty blocks connected by a double-ended arrow is used to represent a swap between two registers. Assume for the sake of argument that the tile to be moved is in cell $(0, 0)$ and that cell $(0, 1)$ is empty. Then a swap could simply be performed on the two corresponding tile registers to compute the next state. However, for reasons that will become clear in a moment, the circuit design presented here instead performs the swap in two steps using an ancillary register, which is simply a group of four ancillary qubits. The first tile register, corresponding to the tile to be moved, is swapped with the ancillary register, and then the ancillary register is swapped with the second tile register, corresponding to the empty cell. Figure 8.6 shows the values of the relevant registers at each step, where x is the number of the tile being moved. We can see that x moves to the register for cell $(0, 1)$, formerly empty, while the register it vacates, corresponding to cell $(0, 0)$, becomes empty instead.

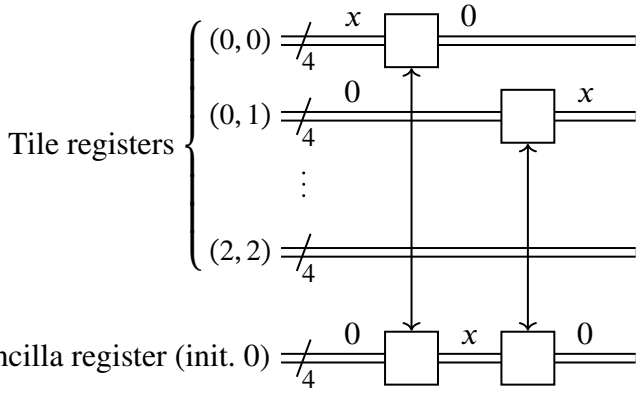


Figure 8.6: Representation of a move by swapping tile registers.

Figure 8.6 only represents a conceptual model of how the tile registers are updated and cannot actually be used as a circuit. This is because the registers to be swapped will

depend on the move and on the location of the empty space. In Figure 8.6, the empty space was assumed to be at the position $(0, 1)$, but this will of course not always be the case. The actual circuit for updating the tile registers therefore uses a multiplexer-demultiplexer design to achieve a swap between the appropriate tile registers depending on the move, as shown in Figure 8.7. In this figure, the multiplexer and demultiplexer are represented by *ad hoc* notations similar to the notation used for the swap operations in Figure 8.6. The multiplexer selects one of the tile registers and transfers it to the ancillary register at the bottom of the circuit, this selection being controlled by the contents of the position qubits. The demultiplexer is the inverse of the multiplexer, so it transfers the ancillary register back to one of the tile registers, again depending on the contents of the position qubits, and restores the ancillary register to its initial value of 0. "NP" and "NP^{-1}" denote, respectively, the next-position circuit from Figure 8.5d and its inverse. An inverse next-position circuit may be obtained by simply reversing the order of gates in Figure 8.5d.

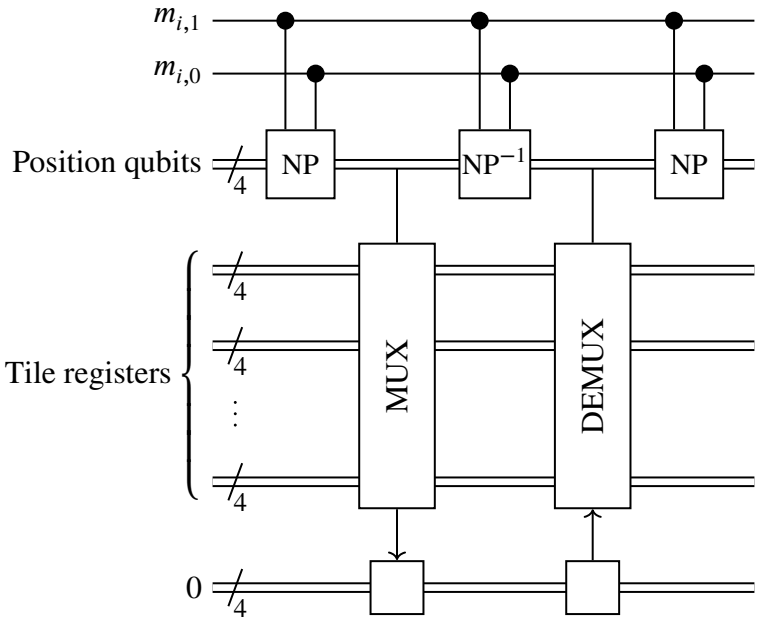
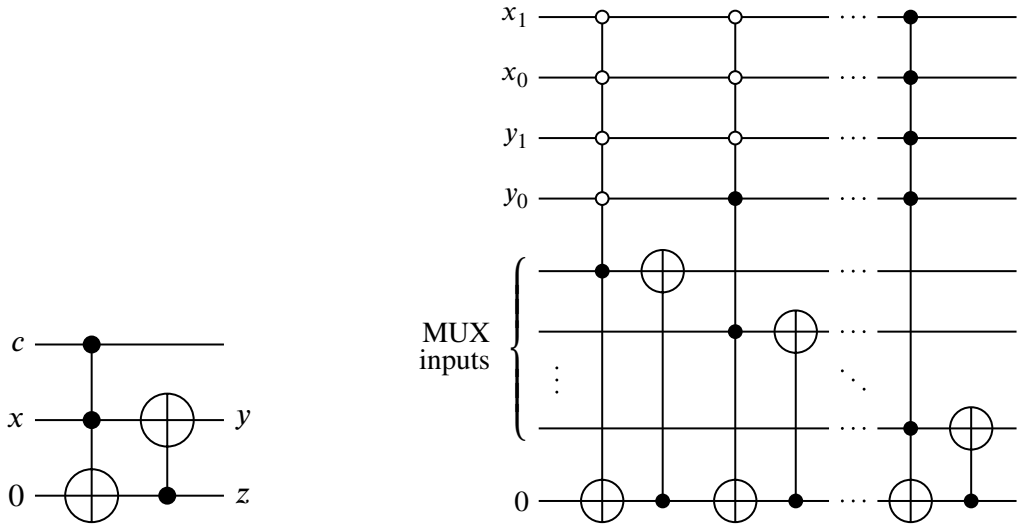


Figure 8.7: Multiplexer-demultiplexer design of the next-state circuit.

The circuit from Figure 8.7 operates in the following manner. Given a move and the current state of the puzzle, consisting of the position qubits together with the tile registers, a next-position circuit is first used to compute the next position of the empty space. This position is the same as the current position of the tile that is being moved. Then, the multiplexer transfers the corresponding tile register to the ancillary register, corresponding to the first swap operation in Figure 8.6. An inverse next-position circuit restores the position qubits to their original values in preparation for the demultiplexer, which then transfers the ancillary register to the tile register that is currently empty, corresponding to the second swap in Figure 8.6. Finally, a second next-position circuit is used to once again update the position qubits to their new values. From this circuit design, it is apparent why two swaps were used in Figure 8.6 instead of a single one: the multiplexer-demultiplexer design requires an ancillary register to act as temporary storage so that the value selected by the multiplexer can be fed into the demultiplexer.



(a) A controlled "half swap". **(b)** A single-qubit multiplexer using controlled half swaps.

Figure 8.8: Implementation of a quantum multiplexer.

It remains to consider how the multiplexer and demultiplexer are implemented. Figure 8.8a shows a circuit that I will refer to as a controlled "half swap". Assuming that the bottommost qubit in this figure is initialized to 0, the circuit swaps the states of the bottom two qubits when the control input c is 1; otherwise, the qubits are unaffected. In other words, the behavior of the controlled half swap is described by the following equations:

$$y = \begin{cases} x & \text{if } c = 0, \\ 0 & \text{if } c = 1 \end{cases} \qquad \text{and} \qquad z = \begin{cases} 0 & \text{if } c = 0, \\ x & \text{if } c = 1, \end{cases} \tag{8.1}$$

where c, x, y, and z are as labeled in Figure 8.8a. This behavior is easily verified by simply checking both cases in (8.1). When $c = 0$, the Toffoli gate in Figure 8.8a is inactive, so the bottommost qubit retains its value of 0, causing the following CNOT gate to be inactive as well. When $c = 1$, the Toffoli gate produces a value of $c \wedge x = x$ on the bottommost qubit, so that the bottom two qubits are both set to x. The following CNOT gate then produces outputs of $y = 0$ and $z = x$. The name "half swap" reflects the fact that the circuit does not swap two arbitrary values, but requires the bottommost qubit to be initialized to 0 in order to function as intended.

Figure 8.8b shows a multiplexer implemented using controlled half swaps. The controlled half swaps have been extended with additional control qubits. Each controlled half swap uses a different combination of control polarities, with the result that exactly one of them is active for each possible combination of values of x_1, x_0, y_1, and y_0. For instance, if $x_1 = x_0 = y_1 = 0$ and $y_0 = 1$, then the second controlled half swap is active and it swaps the corresponding input onto the ancillary qubit. In this way, the multiplexer selects one of the qubits labeled "MUX inputs" in Figure 8.8b and transfers its value onto the ancillary qubit, with the selection being controlled by x_1, x_0, y_1, and y_0. A demultiplexer is the inverse of a multiplexer, so it can be implemented by the circuit from Figure 8.8b with the gates applied

in reverse order.

The multiplexer implementation from Figure 8.8b only selects a single qubit at a time. In order to create a multiplexer that operates on registers consisting of multiple qubits, as used in Figure 8.7, multiple copies of the circuit from Figure 8.8b can be used, with each copy operating on a specific qubit from each register. In particular, four copies of the circuit from Figure 8.8b are sufficient to implement the multiplexer seen in Figure 8.7. In the first copy, the qubits labeled "MUX inputs" in Figure 8.8b consist of the first qubit from each tile register in Figure 8.7, and the ancillary qubit of Figure 8.8b is the first qubit in the ancillary register of Figure 8.7. In the second copy, the "MUX inputs" in Figure 8.8b consist of the second qubit from each tile register, and so on. The demultiplexer of Figure 8.7 is implemented analogously.

For the 3×3 puzzle, the multiplexer circuit from Figure 8.8b may be simplified slightly: since both coordinates are limited to values from 0 through 2, the controlled half swaps corresponding to an x- or y-coordinate of 3 can be omitted. This means that only 9 controlled half swaps are needed, instead of all 16 possible combinations of control polarities that are used for the 4×4 puzzle. Figure 8.7 represents a complete next-state circuit for either the 3×3 or 4×4 puzzle when the multiplexer is implemented as just described and the next-position circuit is implemented as shown in Figure 8.5.

8.3.3 Use of move-restriction circuits in the oracle

The quantum oracle design described in Section 8.2 is sufficient to solve any state-space path planning problem that strictly conforms to the description given at the start of Section 8.1, without any of the mentioned exceptions. However, both of the example problems considered in this chapter do have such exceptions: in both problems, not all moves are available for all

states, as previously discussed in Sections 8.1.1 and 8.1.2. In order to handle this scenario, I introduce another component to the oracle design in addition to the next-state circuit: a *move-restriction* circuit. A move-restriction circuit is one that takes as input the current state and current move, as the next-state circuit does, but instead of computing the next state, it determines whether the move is legal for the particular current state.

If one imagines programming a classical computer to compute the states visited by a given sequence of moves, as the oracle does, it is fairly easy to detect invalid moves. One can simply instruct the classical computer to check the validity of each move, and terminate the simulation immediately if an invalid move is found. This approach cannot be used for the quantum oracle—neither the current state nor the current move can be checked during the execution of the oracle, as they would both be in superposition states when running Grover's algorithm and attempting to measure their values would collapse the superpositions and cause the entire algorithm to fail. One can then instead imagine programming the classical computer with a variable that acts as a flag for invalid moves, where this variable is set to an initial value of 0, set to 1 when an invalid move is detected, and checked after all moves have been simulated. This approach is better suited to the quantum setting, because it does not require mid-computation measurement of the current position or move. However, it still has one problem: the process of setting the flag to 1 upon detection of an invalid move is not reversible. In particular, if a sequence of moves contains more than one invalid move, then it is not possible to reverse any step where an invalid move is detected, because there is no way of knowing whether the invalid-move flag should be set back to 0 or remain with a value of 1, which it would if the current move is not the first invalid move in the sequence.

In order to solve these problems with invalid move detection in the oracle, an *invalid-move counter* is introduced. This invalid-move counter performs the same function as

the "invalid-move flag" described in the previous paragraph, but it avoids the problem of irreversibility by being a counter, as its name suggests, rather than just a single-bit flag. The invalid-move counter uses an additional group of ancillary qubits, the *invalid-move counter register*, added to the quantum oracle and initialized to a value of 0. At every step, if an invalid move is detected, the oracle increments the invalid-move counter register by 1, and after simulation of all moves, the oracle only outputs 1 if, in addition to the desired ending state being reached, the register's value remains at 0. In the inverse stage of the oracle, when the state-tracking qubits are being returned to their original states, the invalid-move counter register is then decremented once each time an invalid move is detected. This guarantees that the invalid-move counter register is reset to its initial value of 0 following operation of the quantum oracle. The register then performs the same role as the hypothetical invalid-move flag discussed earlier, but is compatible with the reversibility requirement for quantum circuits.

To see this concept in practice, consider how a move-restriction circuit may be designed for the 3×3 puzzle. The criteria for determining whether a move is legal can be stated simply: any move that would cause the x- or y-coordinate of the empty space to become negative or exceed 2 is disallowed, while all other moves are legal. This means that the simplest way to check the legality of a move is to check whether at least one of the coordinates has value 3 after the move is made. By a fortunate coincidence, this check handles both underflow and overflow of the coordinates: if a coordinate has value 2 and the next move has the effect of incrementing this coordinate, then it becomes 3, while if a coordinate has value 0 and the next move decrements this coordinate, then the value wraps around and also becomes 3 (since 3 has the base-two representation 11). Therefore, the move-restriction circuit should be inserted into the oracle after each next-state circuit, and it should increment

the invalid-move counter register when either $x_1 = x_0 = 1$ or $y_1 = y_0 = 1$. Figure 8.9 shows a circuit that accomplishes this last task.

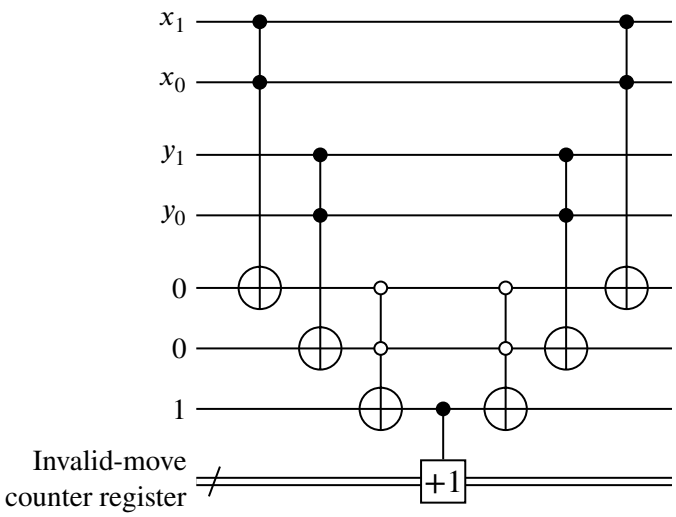


Figure 8.9: A move-restriction circuit for the $3x3$ puzzle.

There is no requirement that the invalid-move counter register must be incremented exactly once upon detection of an invalid move. The move-restriction circuit may also be designed to perform more than one increment for some invalid moves. This can be useful to avoid the need to compute the logical OR of several conditions, all of which render a move invalid. For instance, the circuit from Figure 8.9 uses two ancillary qubits to hold intermediate values: one is used for the result of checking the condition $x_1 = x_0 = 1$, and the other is used for the result of checking $y_1 = y_0 = 1$. A Toffoli gate is then used to compute the logical OR of these two conditions. These two ancillary qubits can be eliminated if the invalid-move counter register is allowed to be incremented twice: the circuit can simply increment the register once if $x_1 = x_0 = 1$ and increment it again if $y_1 = y_0 = 1$. Since any nonzero value of the invalid-move counter register signals an invalid move, the oracle will still correctly recognize a move as invalid if the register is incremented twice. Figure 8.10

shows a move-restriction circuit using two incrementers based on this concept.

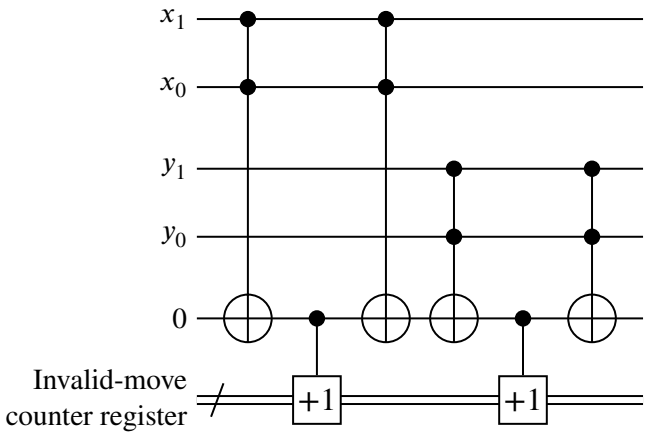


Figure 8.10: An alternative move-restriction circuit using two incrementers to eliminate two ancillary qubits.

As a further enhancement to the invalid-move counter concept, the DIPS-based quantum counter described in Chapter 3, Section 3.4 can be used in place of the incrementer circuits seen in Figures 8.9 and 8.10. Recall that the DIPS-based counter does not operate by incrementing the counter register in discrete steps; rather, controlled root-of-NOT gates are applied to the qubits of the counter register for every signal that is to be counted, and at the end, another sequence of controlled root-of-NOT gates is used to extract the final count from the counter register. Figure 8.11 shows the move-restriction circuit adapted for a DIPS-based counter. The aforementioned final stage of the counter that extracts the actual count is not included because it is only applied when the oracle is generating its final output, after all moves have been simulated.

So far we have only considered move-restriction circuits for the 3×3 puzzle. The design of a move-restriction circuit for the 4×4 puzzle is slightly different. In the 3×3 puzzle, as previously observed, any invalid move will necessarily cause one of the coordinates of the empty space to attain a value of 3, and the move-restriction circuits presented above rely on

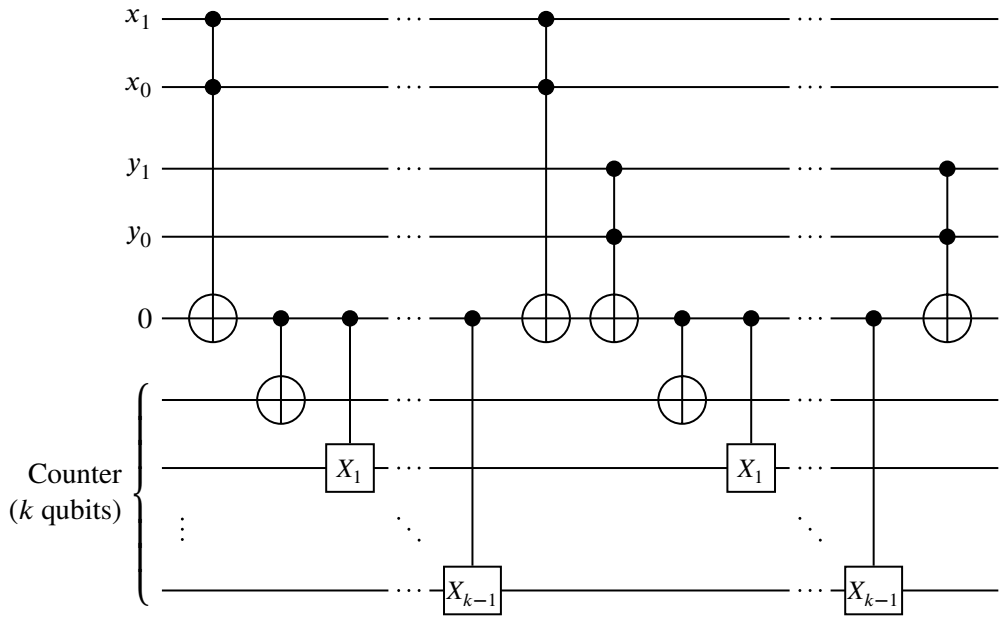


Figure 8.11: The move-restriction circuit adapted for a DIPS-based counter.

this fact. The 4×4 puzzle lacks this property because both coordinates may validly attain values from 0 through 3.

We may use the following reasoning to design a move-restriction circuit for the 4×4 puzzle. If the x-coordinate of the empty space is 3, then it cannot increase any further, so a rightwards move is invalid. Similarly, if the x-coordinate is 0, then a leftwards move is invalid, and the situation is analogous for the y-coordinate. All other moves are valid, since the only invalid moves are those that would move the empty space in an out-of-bounds direction. Referring to Table 8.1, we therefore determine that a move is invalid if and only if

$$(\neg m_{i,1} \wedge \neg m_{i,0} \wedge \neg x_1 \wedge \neg x_0) \vee (\neg m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0) \\ \vee (m_{i,1} \wedge \neg m_{i,0} \wedge \neg y_1 \wedge \neg y_0) \vee (m_{i,1} \wedge m_{i,0} \wedge y_1 \wedge y_0) = 1. \quad (8.2)$$

This is a Boolean function of six variables, which can be realized in several different ways.

The simplest method is to map the logical AND and OR operations of (8.2) directly to Toffoli gates, although this requires four ancillary qubits, one for each of the four terms that are OR-ed together. Another possibility is to use the DIPS-based realization method described in Chapter 4 or any other method for realizing Boolean functions using quantum circuits. Finally we can also map the terms of (8.2) to Toffoli gates but take advantage of the ability to increment the invalid move counter register more than once, as was done in Figure 8.10, which avoids the need for four ancillary qubits. If we suppose for demonstrative purposes that the last option is used, then the resulting circuit is shown in Figure 8.12. This circuit may also be adapted for a DIPS-based counter, analogously to Figures 8.11.

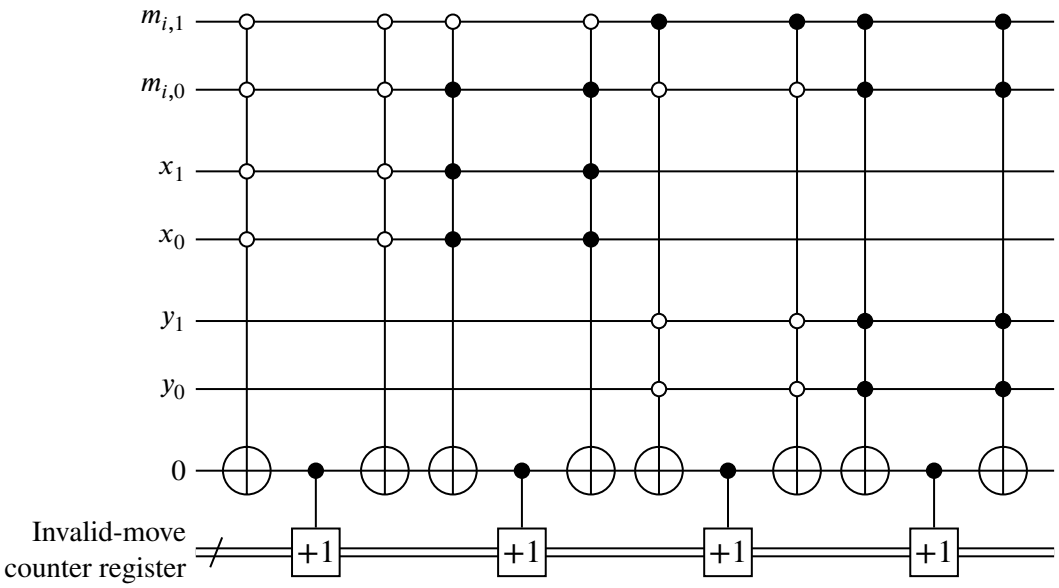
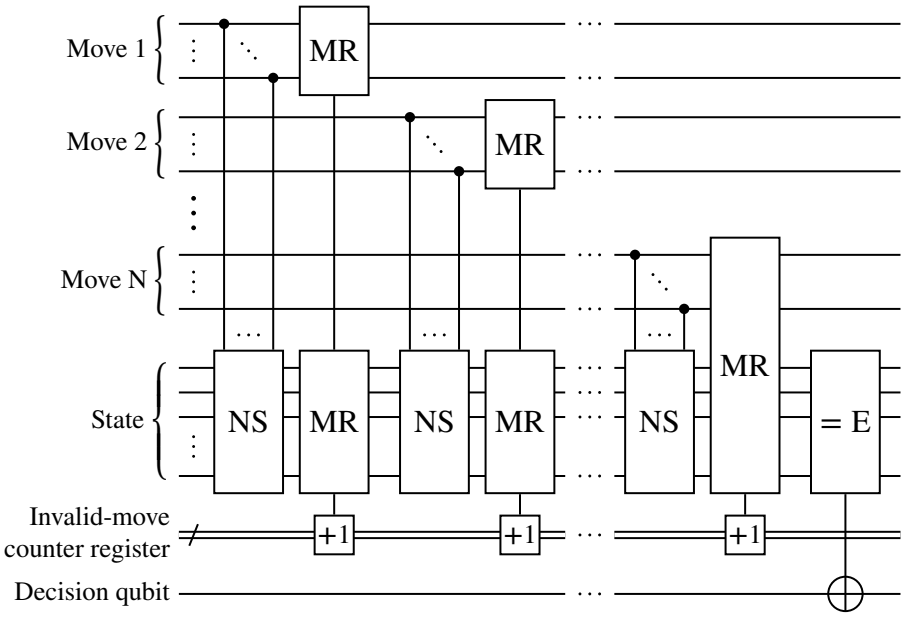


Figure 8.12: A move-restriction circuit for the 4×4 puzzle.

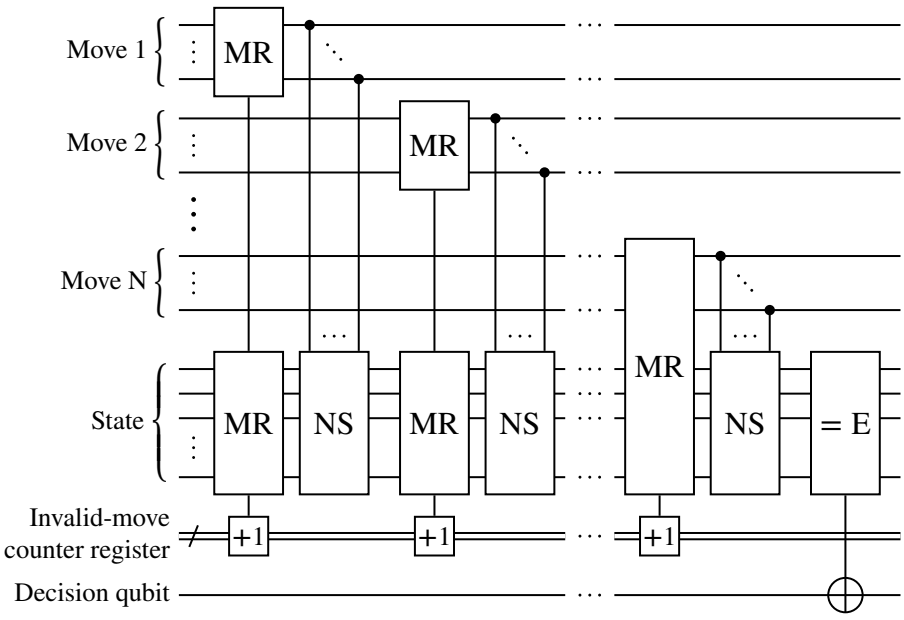
An important difference between the move-restriction circuit for the 3×3 and 4×4 puzzles is that in the 3×3 case, the move-restriction circuit for a given move is placed after the next-state circuit for that move, while in the 4×4 case, the move-restriction circuit is placed before the next-state circuit. For other state-space path planning problems, the choice

of whether to place the move-restriction circuit before or after the next-state circuit will depend on which option allows the circuit to be designed more easily, or which option results in a more efficient circuit. Here, "more efficient" may mean that the circuit has a lower quantum cost or that it uses fewer ancillary qubits, depending on the resources available in the quantum computing system that is being used. It may even be the case that a combined circuit that performs the functions of both the next-state and move-restriction circuits at the same time gives superior results over designing the two circuits separately. Figure 8.13 illustrates the resulting structure of the quantum oracle when the move-restriction circuit for a move is placed either after or before the next-state circuit for that same move. The move restriction circuits other than the last one are represented by two blocks connected by a line, both labeled "MR", because they do not operate on a set of qubits that happen to be contiguous in the circuit. For example, the first move-restriction circuit in Figure 8.13a uses the qubits representing move 1 and the state-tracking qubits to determine whether the first move is invalid, but does not use the qubits representing any of the other moves, so the notation used in this figure visually shows that the circuit "skips over" the qubits for moves 2 through N. The mirror circuits are not shown, but they can easily be derived as they simply consist of the same next-state and move-restriction circuits applied in reverse order.

In order to create a quantum oracle incorporating move-restriction circuits, we must know the number of qubits to be used for the invalid-move counter. The counter cannot be allowed to overflow, because if it does, then its value will wrap around to 0, creating the illusion that there were no invalid moves when in fact there were many. As previously discussed, the move-restriction circuit may be designed to increment the invalid-move counter more than once. If the move-restriction circuit increments the counter up to p times, and the oracle is to simulate N moves, then the highest value the counter can reach is pN. Since k bits



(a) Move-restriction circuit placed after the next-state circuit.



(b) Move-restriction circuit placed before the next-state circuit.

Figure 8.13: Possibilities for incorporating move-restriction circuits into the oracle design from Figure 8.4.

are sufficient to represent positive integers up to $2^k - 1$, it follows that the invalid-move counter register must consist of $\lceil \log_2(pN + 1) \rceil$ qubits to ensure that it does not overflow. This consideration shows that the use of multiple incrementers in Figures 8.10 and 8.12 to eliminate ancillary qubits is not entirely "free", since it increases the required size of the counter register, whose constituent qubits are after all also ancillary qubits for the oracle. Nevertheless, since the required counter register size only increases with the logarithm of the number of incrementers per move-restriction circuit, this increase is far outweighed by the number of ancillary qubits that can be saved by using those incrementers. For instance, the use of four incrementers in Figure 8.12 eliminates four ancillary qubits that would otherwise be needed to compute a logical OR of four terms, but only increases the size of the counter register by two qubits (as compared to the register size that would be required if the move-restriction circuit only used one incrementer). In a scenario where a logical OR of eight terms is needed, the use of eight incrementers would eliminate eight ancillary qubits while only increasing the size of the counter register by three qubits, and so on for higher numbers of terms.

8.3.4 Ending-state-checking circuit

The desired solved state of the puzzle, as depicted in Figure 8.1, has the empty space in the lower right-hand corner and all numbered tiles arranged sequentially in order. It is easy to formulate the solved state as a set of conditions that must be checked for the state-tracking qubits in the oracle. In particular, suppose we denote the position qubits by x_1, x_0, y_1, and y_0 as before; denote the qubits making up the i-th tile register as $t_{i,3}t_{i,2}t_{i,1}t_{i,0}$, where the registers are numbered starting from 1 and proceeding in order from left to right then top to bottom; and denote the qubits making up the invalid-move counter register as c_i. Then the

3×3 puzzle is solved when the following conditions are all met:

$$x_1 = 1 \qquad x_0 = 0 \qquad y_1 = 1 \qquad y_0 = 0 \tag{8.3}$$

$$(t_{i,3}t_{i,2}t_{i,1}t_{i,0})_2 = i \text{ for } 1 \leq i \leq 8 \tag{8.4}$$

$$t_{9,3} = t_{9,2} = t_{9,1} = t_{9,0} = 0 \tag{8.5}$$

$$c_i = 0 \text{ for all } i, \tag{8.6}$$

where $(t_{i,3}t_{i,2}t_{i,1}t_{i,0})_2$ denotes the integer represented in base-two by $t_{i,3}t_{i,2}t_{i,1}t_{i,0}$. We have (8.3) because the coordinates of the lower-right corner are $(2, 2)$ in the 3×3 puzzle, and an x-coordinate (resp. y-coordinate) of 2 is represented by $x_1x_0 = 10$ (resp. $y_1y_0 = 10$.) For the 4×4 puzzle, the coordinates of the lower-right corner are instead $(3, 3)$, so (8.3) must be amended so that $x_0 = y_0 = 1$ instead.

Eq. (8.4) represents the requirement that the numbered tiles must be in sequential order. Since the i-th tile register contains the base-two representation of the number of the tile in the i-th position (again, counting from left to right and top to bottom), it must contain the base-two representation of i in the solved state of the puzzle. We have $1 \leq i \leq 8$ for the 3×3 puzzle since there are 8 tiles; for the 4×4 puzzle, the range of i in (8.4) should be amended to $1 \leq i \leq 15$. In a similar vein, (8.5) represents the requirement for the empty space to be in the lower-right corner, since the empty space is represented by a tile register with a value of 0. For the 4×4 puzzle, the lower-right corner corresponds to the 16th tile register, so the first index of the t's in (8.5) should instead be 16.

Finally, (8.6) will be satisfied when no invalid moves have been made. A solution to the puzzle of course requires all moves to be valid. As described in Section 8.3.3, any non-zero value of the invalid-move counter register indicates that at least one invalid move has been

made, so we require that all qubits making up this register have a value of zero.

In practice, it is not necessary to check all of the above conditions. In particular, it is enough to check that (8.4) and (8.6) are satisfied. This is because under normal operation, when no invalid moves are made, the next-state circuit from Figure 8.7 only permutes the contents of the tile registers, so if the first 8 (or 15 for the 4×4 puzzle) registers have the correct values, then the last one must have the correct value as well. Similarly, the position qubits represent information (the location of the empty space) that is already contained in the tile registers, so they do not need to be checked either. If at least one invalid move is made, the state can become "corrupted"—that is, the values of the position qubits are no longer consistent with those of the tile registers, or the tile registers no longer represent a valid permutation of the tiles—since the the next-state circuit is not designed to handle invalid moves. However, in this case, the move-restriction circuit will recognize the move as invalid and increment the invalid-move counter, preventing (8.6) from being satisfied. Therefore, the ending state-checking circuit can be designed to only check (8.4) and (8.6). Since these conditions amount to checking that each qubit in some collection has a particular expected value, it can be done using a single Toffoli gate, which is shown for the 3×3 puzzle in Figure 8.14. The ending-state-checking circuit for the 4×4 puzzle is nearly identical, but uses 15 tile registers.

8.3.5 The full oracle and extension to larger puzzle sizes

Complete quantum oracles for either the 3×3 or 4×4 puzzles can now be created by inserting the next-state, move-restriction, and ending-state-checking circuits into one of the templates from Figure 8.13. As discussed in Section 8.3.3, the move-restriction circuit for the 3×3 puzzle is designed to be placed after the next-state circuit, so the template from

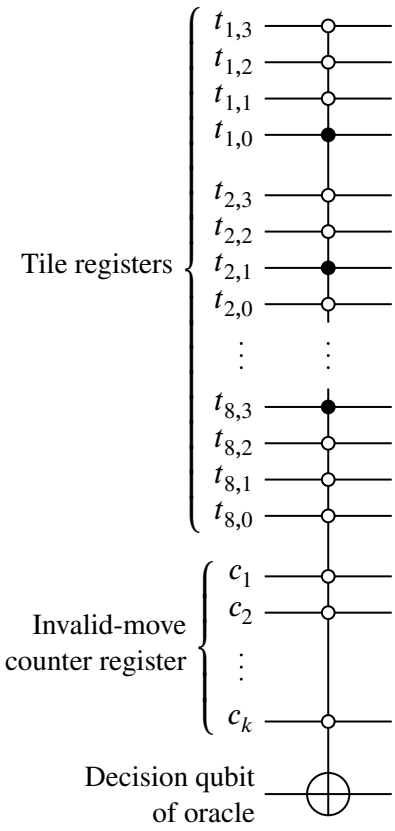


Figure 8.14: Ending-state-checking circuit for the 3×3 puzzle.

Figure 8.13a should be used in that case. For the 4×4 puzzle, the move-restriction circuit is designed to be placed before the next-state circuit, so the template from Figure 8.13b should then be used.

Most of the circuits described in this section are easily scaled up to handle an $(n^2 - 1)$-puzzle. Regardless of the size of the puzzle, there are always the same four possible moves, so each move is always represented by two move qubits. For an $n \times n$ puzzle, the position qubits must be able to handle coordinates of up to $n-1$, so $\log_2 n$ qubits are required for each coordinate, giving a total of $2 \log_2 n$ position qubits. The incrementers in the next-position circuit are then accordingly scaled up as well. The number of tile registers for an $n \times n$ puzzle is n^2, and each one of them must be able to represent the integers from 0 through $n^2 - 1$. Each tile register must therefore have a size of $\log_2(n^2) = 2 \log_2 n$ qubits. The multiplexers used in the next-state circuit are then also easily scaled up to handle these larger tile registers. As for the ending-state-checking circuit, it straightforward to extend eqs. (8.3) through (8.6) to larger puzzle sizes as well, so the reasoning used in Section 8.3.4 serves to create ending-state-checking circuits for these puzzles.

The one circuit whose scaling-up is not completely straightforward is the move-restriction circuit, because different design approaches were used for the 3×3 and 4×4 versions of this circuit. For an $n \times n$ puzzle, if n is not a power of 2, then the design used for the 3×3 puzzle can be generalized: the circuit should be placed after the next-state circuit, and it should check whether either the x- or y- coordinate is out of bounds, keeping in mind that the decrementers used in the next-state circuit to update the position qubits will cause a coordinate to wrap around if it is decremented when at 0. For instance, in a 5×5 puzzle, each coordinate is represented by three position qubits, and the move-restriction circuit should check whether either of the coordinates has attained a value of 5 or 7 (7 being the maximum

integer representable by three bits, and therefore being the value that a coordinate will wrap around to on underflow). If n is one less than a power of 2, then the values attained by overflowing and underflowing coordinates coincide, as they did for the 3×3 puzzle, which simplifies the move-restriction circuit. For instance, in a 7×7 puzzle, the valid range for a coordinate is 0 through 6, so on either overflow or underflow, a coordinate will attain the value 7. The move-restriction circuit therefore only needs to check whether either coordinate has a value of 7, just as the move-restriction circuit for the 3×3 puzzle checks whether either coordinate has a value of 3.

If n, the size of the puzzle, is a power of two, then the entire range of representable coordinates is valid, so the move-restriction circuit design for the 4×4 puzzle must be used instead. Specifically, the move-restriction circuit should check, for each coordinate, whether either one of the following is true: the coordinate has a value of 0 and the next move would have the effect of decrementing it, or the coordinate has a value of $2^n - 1$ and the next move would have the effect of incrementing it. This description can easily be converted into a Boolean-algebraic expression, which can then be realized using any of the options described in Section 8.3.3. In this case, the move-restriction circuit must be placed before the next-state circuit, as in Figure 8.13b.

In sum, the oracle design described in this section can be scaled up for puzzles of any size, and it can be described in terms of low-level gates by using a realization method for Boolean functions together with the known scalable implementations for functional blocks like incrementers and multiplexers.

8.4 Design of a quantum oracle to solve the labyrinth problem

8.4.1 Encoding scheme for states and moves

A state in the labyrinth problem consists of a position together with the state of the door (open or closed). Position may be encoded using the same method used for the 8- and 15-puzzles: the x- and y-coordinates are represented by two qubits each, for a total of four position qubits. This scheme is easily extended to labyrinths of larger sizes, so three qubits per coordinate is sufficient for an 8×8 labyrinth, four qubits per coordinate is sufficient for 16×16, and so on. Since there are only two possible states of the door, it could be represented by only a single qubit. However, for reasons that are discussed in Section 8.4.2, the oracle design presented here will use multiple qubits to represent the state of the door, where a nonzero value on any of these qubits indicates that the door is open.

Possible moves consist of a one-cell movement in any of the four cardinal directions, together with a fifth move of activating the switch for the door. A minimum of three qubits is therefore required to encode a single move. Following the same notational convention used in Section 8.3.1, the three qubits are labeled $m_{i,2}$, $m_{i,1}$, and $m_{i,0}$. The moves are then encoded by the following scheme: $m_{i,2} = 1$ indicates that the current move is to activate the switch, regardless of the values of $m_{i,1}$, and $m_{i,0}$, while if $m_{i,2} = 0$, then $m_{i,1}$ and $m_{i,0}$ specify a direction in the same manner as in Table 8.1. This encoding scheme is summarized by Table 8.2, where dashes (–) indicate that the corresponding qubit may have any value.

8.4.2 Next-state circuit and handling of non-reversible moves

Since the movement moves (*i.e.*, all moves except for activating the switch) are encoded in the same manner as the moves for the 8- and 15-puzzles, the next-position circuit for

Table 8.2: Encoding of moves for the labyrinth problem.

$m_{i,2}m_{i,1}m_{i,0}$	Move
0 0 0	Left
0 0 1	Right
0 1 0	Up
0 1 1	Down
1 – –	Activate switch

that problem, described in Section 8.3.2 and shown in Figure 8.5, can be reused. One minor modification must be made, however. Since there is a third move qubit, $m_{i,2}$, and $m_{i,2} = 1$ indicates activating the switch rather than a movement move, we must add $m_{i,2}$ as a negative-polarity control qubit to every gate in the next-position circuit from Figure 8.5d. This modification ensures that the position qubits are not subject to change if the move is to activate the switch. Figure 8.15 shows the circuit with these additional controls added.

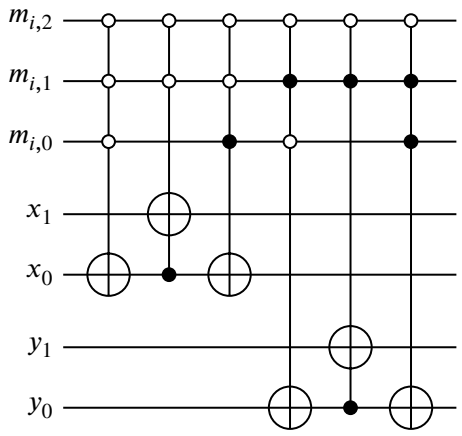


Figure 8.15: Next-position circuit for the labyrinth, obtained by adding another control qubit to the circuit from Figure 8.5d.

We are now faced with the question of how to design a circuit to update the state of the door when the switch is activated. In Section 8.1.2, it was observed that the move of activating the switch and opening the door constitutes a non-reversible state change. Since

activating the switch more than once has no additional effect, the next-state function fails to be reversible, because if the most recent move was to activate the switch, then one cannot tell whether or not the door was already open prior to that move. This presents a problem since all quantum circuits are reversible.

The same counter-based approach used in Section 8.3.3 to implement a move-restriction system can also be used here. Specifically, we may allocate a register of qubits, the *door-open counter register*, to track the state of the door, where a non-zero value of this register indicates that the door is open. The next-state circuit should then be designed to increment this counter every time the switch is activated. This mechanism allows a reversible next-state circuit to simulate the desired non-reversible behavior where the door remains open when the switch is activated multiple times.

One must be at the switch, which has coordinates $(3, 3)$, in order to activate it. Therefore, the circuit that increments the door-open counter register should check whether the x- and y-coordinates, which are part of the state register, both have values 11 (the base-two representation of 3). Of course, even when at the coordinates $(3, 3)$, the switch is only activated if the current move is the switch-activation move. According to the encoding scheme given in Section 8.4.1, this move is represented by setting the bit $m_{i,2}$ to 1. In sum, the bits x_1, x_0, y_1, y_0, and $m_{i,2}$ must all have values 1 in order for the current move to activate the switch. This logic is implemented by a 5-control Toffoli gate, as shown in Figure 8.16.

A difference between the door-open counter and the invalid-move counter is that the former cannot be implemented using a DIPS-based counter. A DIPS-based counter is not based on incrementer circuits and does not allow the current value of the counter register to be accessed partway through the counting process, and such access is required for the door-open counter because the move-restriction circuit must be able to check whether the

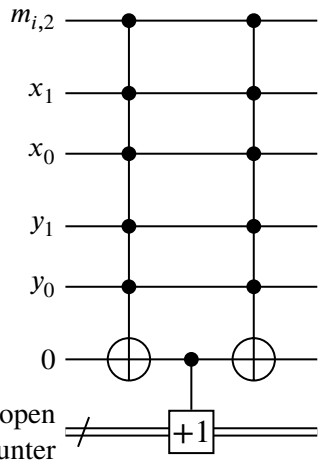


Figure 8.16: A circuit for updating the door-open counter register.

door is open, as detailed in the next section.

With the state of the door represented by the door-open counter, the complete state of the labyrinth problem is then encoded by the combination of position qubits and door-open counter. The complete next-state circuit consists of the circuits from Figures 8.15 and 8.16 concatenated in either order. The order of concatenation is unimportant because the two circuits operate on independent parts of the state: the next-position circuit (Figure 8.15) affects only the position qubits while the door-opening circuit (Figure 8.16) affects only the door-open counter register.

8.4.3 Move-restriction circuit

Invalid moves for the labyrinth come in two types. The first type are moves that attempt to move through a wall. These moves are always invalid, so they can be detected by checking the combination of the current move and current position. For instance, if the position qubits have values $x_1 x_0 y_1 y_0 = 1001$, then the current position is $(2, 1)$, and if the move qubits are $m_{i,2} m_{i,1} m_{i,0} = 010$, then the current move is to move upwards. Referring to Figure 8.3, we

can see that this move is attempting to move through a wall, so it is invalid.

The second type of invalid move is one that attempts to move through the door when it has not yet been opened. Unlike the first type of invalid move, correct detection of these moves therefore requires the oracle to also check the state of the door, in addition to the current position and current move. As described in Section 8.4.2, the state of the door is represented by a door-open counter in which a value of zero represents a closed door and any nonzero value represents an open door. If the qubits making up the door-open counter register are denoted $d_{k_D-1}, d_{k_D-2}, \ldots, d_1, d_0$, where k_D is the number of qubits making up this register, then the oracle must evaluate $\bigwedge_{j=0}^{k_D-1} \neg d_j$ to determine the state of the door, with this expression evaluating to 1 if and only if the door is closed.

There is in fact a third type of move that could potentially be considered invalid, although I choose not to do so. These are moves that attempt to activate the switch when the current position is anything other than $(3, 3)$, the position of the switch. We could design the move-restriction circuit to treat such moves as invalid, but we can also ignore them with no ill effect: the next-position circuit from Figure 8.15 will simply do nothing when $m_{i,2} = 1$, while the door-opening circuit from Figure 8.16 refuses to update the door-open counter when any of x_1, x_0, y_1, or y_0 are not equal to 1. The oracle as-is therefore already ignores moves that attempt to activate the switch when not at the correct position, treating such moves as no-ops. We can gain a significant advantage from ignoring these moves rather than treating them as invalid. Recall from Chapter 1, Section 1.4 that, when faced with a function that can be satisfied in multiple ways, Grover's algorithm has a run time of $\mathcal{O}\left(\sqrt{N/k}\right)$, where N is the size of the search space and k is the number of solutions. If we treat all switch-activation moves as invalid when not in the correct position, then out of all possible sequences of moves, a very large fraction will be considered invalid, causing Grover's algorithm to require a long

run time. This issue is exacerbated by the fact that, due to the switch-activation move having four different encodings while the other moves only have one encoding each, as shown in Table 8.2, sequences of moves containing many switch activations are disproportionately represented in the search space. In contrast, if switch-activation moves are simply ignored for incorrect positions, then the number of move sequences that the oracle considers to be valid solutions greatly increases: a solution can in that case consist of any number of switch-activation moves interspersed among moves that actually solve the labyrinth. This allows Grover's algorithm to find a solution in a shorter amount of time.

The move-restriction circuit must now be designed to detect the two types of invalid moves described previously. Based on the discussion in the previous paragraph, a switch-activation move will never be considered invalid. The switch-activation move is encoded by assigning a value of 1 to $m_{i,2}$, so a move can only be considered invalid if $m_{i,2} = 0$. This leads to the following expression that defines invalid moves:

$$\neg m_{i,2} \wedge \Big(f_{\text{walls}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0) \\ \vee f_{\text{door}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0, d_{k_\text{D}-1}, d_{k_\text{D}-2}, \ldots, d_0) \Big), \tag{8.7}$$

where the expression evaluates to 1 if a move is invalid. The functions f_{walls} and f_{door} define the two types of invalid moves: f_{walls} evaluates to 1 when a move attempts to move through a wall, while f_{door} evaluates to 1 when a move attempts to move through the door and the door has not been opened yet. The output of f_{door} depends on the state of the door, so it takes the qubits of the door-open counter register, $d_{k_\text{D}-1}$ through d_0, as inputs. The output f_{walls} does not depend on the state of the door and so it does not take these qubits as inputs. Furthermore, f_{door} does not depend on the values of $d_{k_\text{D}-1}$ through d_0 individually but only on whether the door is open or closed. As previously discussed, the door is closed only

when all of these qubits are set to zero. When the door is open, no move that attempts to move through the door can be invalid. Therefore, f_{door} can be expressed as

$$f_{\text{door}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0, d_{k_{\text{D}}-1}, d_{k_{\text{D}}-2}, \ldots, d_0) = \left(\bigwedge_{j=0}^{k_{\text{D}}-1} \neg d_j \right) \wedge f_{\text{closed door}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0), \tag{8.8}$$

where $f_{\text{closed door}}$ is a function that checks whether the move defined by $m_{i,1}$ and $m_{i,0}$ is attempting to move through the door. If $f_{\text{closed door}}$ evaluates to 1, then the move is only considered invalid if the door-open counter register has a value of 0, which is reflected in (8.8) by the first term after the equals sign.

The design of the move-restriction circuit now reduces to obtaining specifications of and realizing the two functions f_{walls} and $f_{\text{closed door}}$. I consider the second function first because it is slightly simpler. There are only two possible combinations of move and position that result in attempting to move through the door. Either the move is upwards and the position is $(1, 2)$, or the move is downwards and the position is $(1, 1)$. Mapping these moves and positions to the corresponding values of $m_{i,1}$, $m_{i,0}$, x_1, x_0, y_1, and y_0, we therefore obtain

$$f_{\text{closed door}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0) = (m_{i,1} \wedge \neg m_{i,0} \wedge \neg x_1 \wedge x_0 \wedge y_1 \wedge \neg y_0) \vee (m_{i,1} \wedge m_{i,0} \wedge \neg x_1 \wedge x_0 \wedge \neg y_1 \wedge y_0), \tag{8.9}$$

which can be realized by the circuit shown in Figure 8.17.

Note that the circuit from Figure 8.17 avoids using additional ancillary qubits to take the logical OR of the two terms from (8.9), as was done in Figure 8.9. The two terms of (8.9) are mutually exclusive—they cannot both evaluate to 1 at the same time—so their logical (inclusive) OR is the same as their exclusive-OR, the latter being easily realized

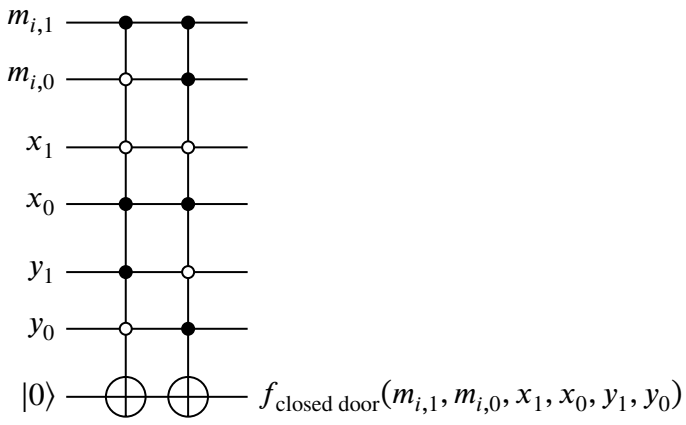


Figure 8.17: A circuit that realizes the function $f_{\text{closed door}}$ using (8.9).

using Toffoli gates. Using Figure 8.17 as a subcircuit together with (8.8) then leads to the circuit of Figure 8.18.

Moving on to the function f_{walls} from (8.7), this function can be expressed as the logical OR of a large number of logical-AND (conjunctive) terms, each of which corresponds to a combination of move and position that results in moving through a wall. I list a few of them here:

$$\begin{aligned} f_{\text{walls}}(m_{i,1}, m_{i,0}, x_1, x_0, y_1, y_0) = (&m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1 \wedge y_0) \\ &\vee (m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1 \wedge \neg y_0) \\ &\vee (m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge \neg x_0 \wedge y_1 \wedge y_0) \\ &\vee \cdots \\ &\vee (\neg m_{i,1} \wedge \neg m_{i,0} \wedge \neg x_1 \wedge \neg x_0 \wedge \neg y_1 \wedge \neg y_0). \end{aligned} \tag{8.10}$$

For instance, the first term of (8.10) corresponds to a downwards move at position $(3, 3)$, and referring to Figure 8.3, we can see that this move does indeed attempt to move through a wall and so is invalid. As in (8.9), all of the terms on the right-hand side of (8.10) are

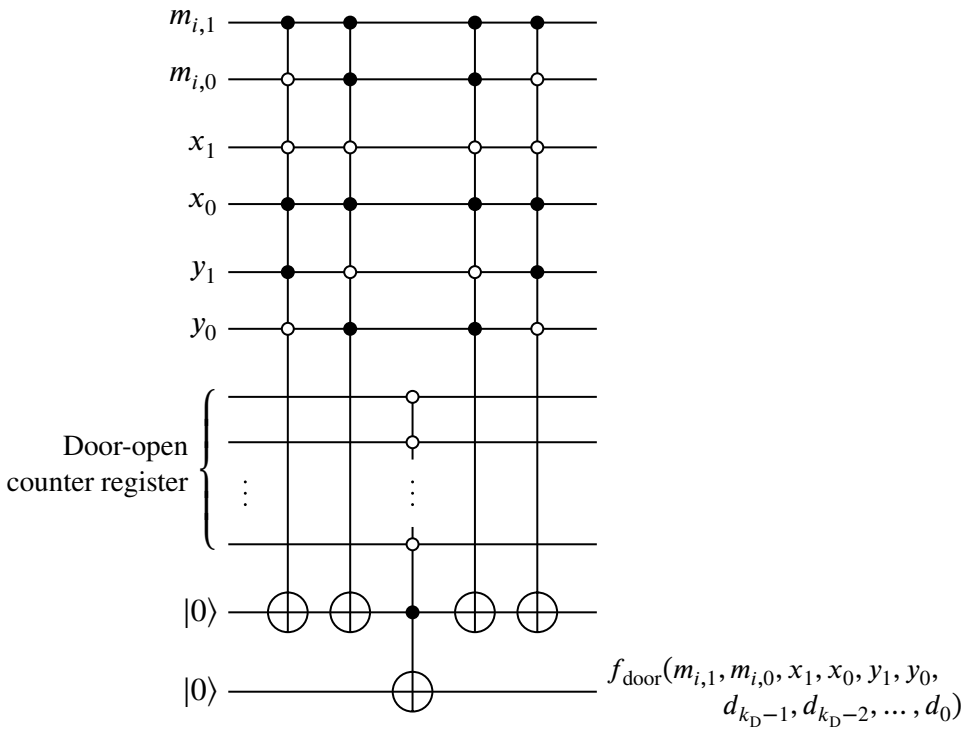


Figure 8.18: A circuit that realizes the function f_{door} using (8.8).

mutually exclusive, so f_{walls} can be realized by simply mapping each term to a Toffoli gate, with all such gates targeting the same output qubit. This produces the circuit shown in Figure 8.19. However, we can immediately see that this circuit is non-optimal, because the terms of (8.10) can be combined in ways that clearly result in a circuit with lower quantum cost. For instance, the first two terms of (8.10) can be combined as

$$\begin{aligned}
&(m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1 \wedge y_0) \vee (m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1 \wedge \neg y_0) \\
&\quad = \big(m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1 \wedge (y_0 \wedge \neg y_0)\big) \\
&\quad = (m_{i,1} \wedge m_{i,0} \wedge x_1 \wedge x_0 \wedge y_1), \qquad (8.11)
\end{aligned}$$

which allows the first two 6-control Toffoli gates in Figure 8.19 to be replaced by a single

5-control Toffoli gate. Combining terms of (8.10) to produce a realization of f_{walls} with low quantum cost is a task that can be performed by a program such as EXMIN2 [15] or EXORCISM [17]. Alternatively, as previously pointed out in Section 8.3.3 for the function given by eq. (8.2), the DIPS-based realization method from Chapter 4 can also be used to realize the function f_{walls}.

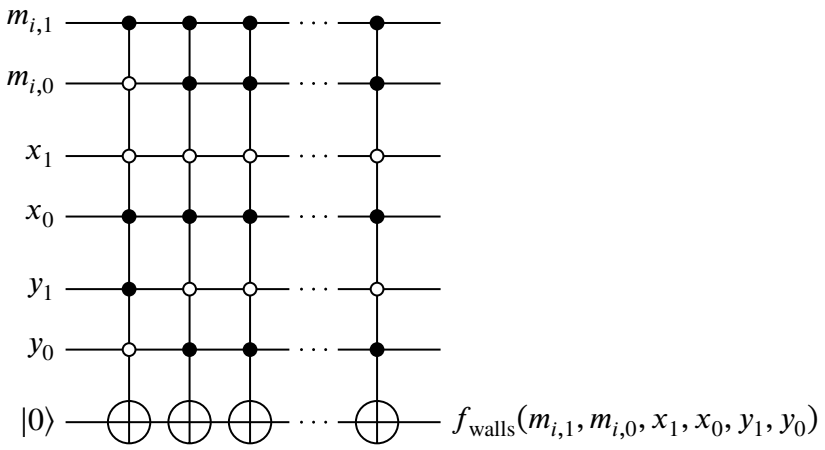


Figure 8.19: A circuit that realizes the function f_{walls} using (8.10).

Once the function f_{walls} has been realized, regardless of what method is used to do so, the full move-restriction circuit can then be created as shown in Figure 8.20. This circuit uses realizations of f_{walls} and $f_{\text{closed door}}$, and also uses Figure 8.18 as a subcircuit to compute f_{door}. The realization of $f_{\text{closed door}}$ is presented as a "black box" rather than using the circuit from Figure 8.17 because it is also possible that one of the realization methods mentioned before can produce a circuit for $f_{\text{closed door}}$ with lower quantum cost than the one from Figure 8.17. The use of "black boxes" in Figure 8.20 additionally helps to highlight the fact that the move-restriction circuit has been described in terms of Boolean functions (f_{walls} and $f_{\text{closed door}}$) whose specifications (eqs. (8.10) and (8.9)) are directly derived from the layout of the labyrinth, plus some simple connecting logic. From this point on, the

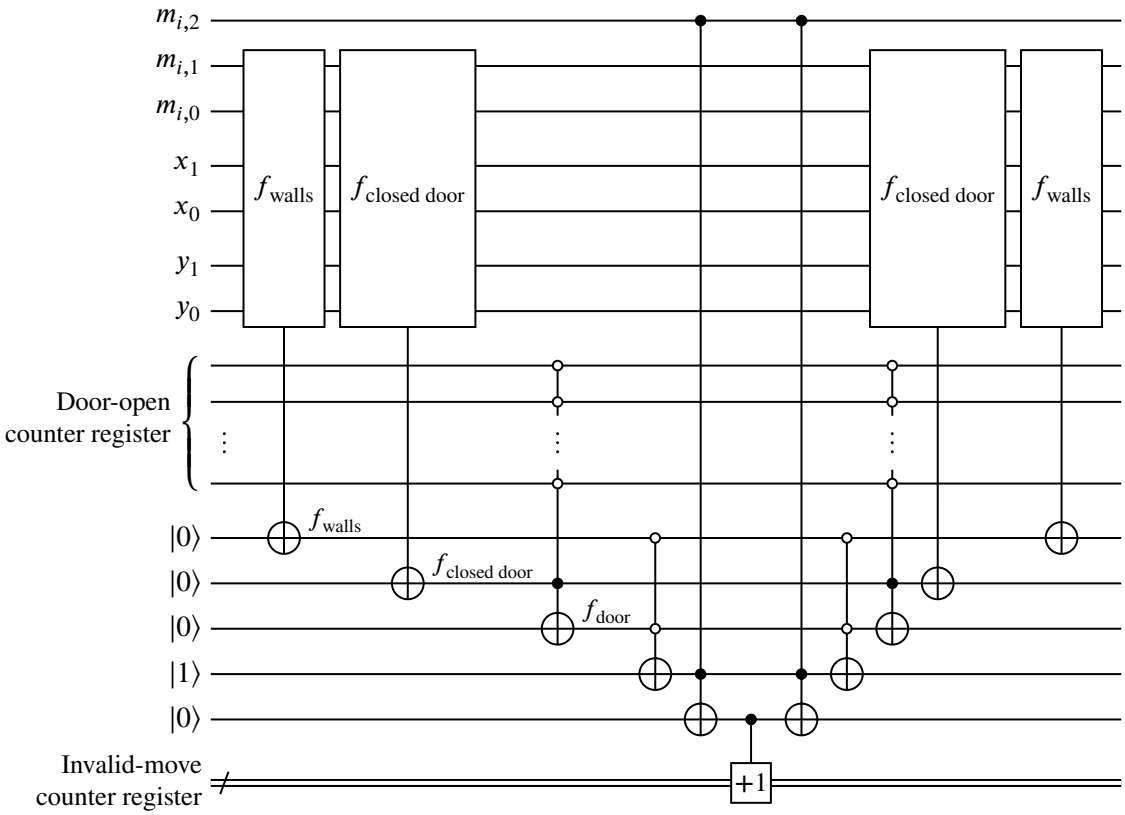


Figure 8.20: Full move-restriction circuit for the labyrinth problem, using realizations of f_{walls} and $f_{\text{closed door}}$ as subcircuits.

move-restriction circuit can be automatically generated in full detail down to the level of two-qubit controlled gates using any realization method for single-output Boolean functions.

8.4.4 Ending-state-checking circuit and summary of full oracle

Compared to the move-restriction circuit, the ending-state-checking circuit for the labyrinth problem is extremely simple. The desired ending state is simply one with a position of $(2, 0)$, which has the position-qubit representation $x_1 = 1$ and $x_0 = y_1 = y_0 = 0$. This condition can be checked using a single Toffoli gate, as shown in Figure 8.21.

Note that the state of the door does not participate in the ending-state-checking circuit at

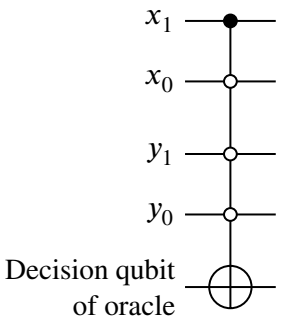


Figure 8.21: Ending-state-checking circuit for the labyrinth problem.

all. From the layout of the labyrinth as shown in Figure 8.2, it is clear that the ending position cannot be reached without opening the door, but this fact is irrelevant to the design of a quantum oracle for this problem: since solving the labyrinth is defined simply as reaching the position $(2, 0)$, that is all that the oracle needs to check. However, the state of the door does form part of the overall state when the labyrinth problem is formulated as a state-space path planning problem. This means that, strictly speaking, the labyrinth problem actually has more than one possible ending state: one can have position $(2, 0)$ with the door open, or position $(2, 0)$ with the door closed. The latter state turns out to be unreachable from the given starting state, but it is still a valid element of the state space. The possibility to have more than one ending state is incorporated into the more precise definition of a state-space path planning problem given in Section 8.5.

Having designed the next-state, move-restriction, and ending-state-checking circuits, the full quantum oracle for the labyrinth problem is obtained by inserting these circuits into Figure 8.13b. The move-restriction circuit designed here works with the current state before it is updated and should therefore be placed before the next-state circuit.

As with the oracles for the 8- and 15-puzzles, the oracle design described here is scalable and works for labyrinths of any size. The number of position qubits can be increased and the

incrementers and decrementers scaled up accordingly to handle larger ranges of coordinates. The move-restriction circuit can be generated as before by realizing Boolean functions whose specifications are directly derived from the layout of the labyrinth; the number of inputs to these functions will simply increase. It is also possible to create an oracle for a labyrinth with multiple doors and switches. In this case, each door is assigned its own door-open state counter and a door-opening circuit like the one from Section 8.4.2 is created for every switch. The expression given in (8.7), which is used to generate the move-restriction circuit, then has to incorporate additional terms, one for each door. Just as for the 8- and 15-puzzles, any such oracle can be automatically described in terms of low-level gates once its high-level design has been created, by realizing the appropriate Boolean functions used in the move-restriction circuit and using the known scalable implementations of the incrementers and decrementers.

8.5 General strategy to design oracles for state-space path planning problems

Modern digital logic design follows a strategy or approach of designing circuits from high-level functional blocks, where a human designer uses such blocks to design circuits and a CAD tool automatically generates low-level implementations of these blocks on the transistor level. The high-level blocks include components such as adders, multipliers, comparators, multiplexers, counters, etc. This procedure is not completely automatic because the human designer must still rely on their own experience and intuition to decide, given a set of requirements, how to design a circuit satisfying those requirements using the available high-level components. Such a strategy or approach might be called a “methodology”, although it is not a methodology in a narrow sense because it does not consist of a sequence of fully systematic steps.

In a similar manner, the examples I presented in this chapter lead to a strategy or approach for designing quantum oracles for state-space path planning problems, where a human designer uses high-level blocks to design a next-state circuit, move-restriction circuit, and ending-state-checking circuit. As with classical digital logic design, this strategy might be called a "methodology" in a broad sense but not in the narrow sense of a fully systematic sequence of steps. The human designer must rely on experience and intuition to determine how to translate the parameters of a given state-space path planning problem into a high-level oracle design. However, once the high-level design has been created, its implementation in terms of low-level gates can be automatically created because components such as incrementers and multiplexers can be implemented by regular circuit structures that are easily generated algorithmically, while Boolean functions can be realized using algorithms such as those described in Chapters 3 through 5. Furthermore, the human designer may be able to parameterize the high-level oracle design so that its low-level implementation can be automatically updated with no further intervention if the parameters of the original problem change slightly. For instance, the move-restriction circuit for the labyrinth problem, as shown in Figure 8.20, uses Boolean functions (f_{walls} and $f_{\text{closed door}}$) whose precise specifications depend on the layout of the labyrinth. In the future, if a sufficiently powerful quantum circuit description language is created, a human designer will be able to use this language to describe how the specifications of these functions are derived from the labyrinth layout, and a compiler for the language will be able to automatically update the functions if the labyrinth layout is changed.

In order to describe the general oracle-design strategy for state-space path planning problems in more detail, I first give a revised and more precise definition of a state-space path planning problem, based on the considerations encountered while designing oracles for

both the 8-/15-puzzle and labyrinth problems:

1. One has a set of states and a set of moves that define allowed transitions between these states. The objective is to reach a desired ending state, or one state out of a set of desired ending states, from a given starting state in as few moves as possible.

2. For every state, a subset of the set of moves is legal. This subset can be improper, *i.e.*, the whole set of moves can be legal for some or all states. The legal moves can be defined by a *move-legality* function that takes as input a state and a move, and outputs 0 if the move is legal for that state and 1 otherwise.

3. Each state S is representable as a combination of a *reversible* component, S_R, plus a fixed number of *flags*, s_1 through s_n, which take on Boolean values.

4. Every legal state-move combination is mapped to exactly one next state. Given a state with components $S_R, s_1, s_2, \ldots, s_n$, together with a legal move M for that state, denote the next state's components as $S'_R, s'_1, s'_2, \ldots, s'_n$. Then these components can be obtained using functions $f_R, f_1, f_2, \ldots, f_n$, which have the following properties:

   (a) f_R takes the reversible component of the state together with the move and produces the reversible component of the next state. In other words, $f_R(S_R, M) = S'_R$. f_R must be reversible in the sense that given S'_R and M, it is always possible to recover the value of S.

   (b) For $1 \leq i \leq n$, f_i takes the reversible component of the state together with all of the flags *except* for s_i, and produces a Boolean output. The new value of the i-th flag, s'_i, is then given by $s'_i = s_i \vee f_i(S_R, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$.

Item 1 above is essentially the same as before, although the possibility of multiple ending

states has been added based on the observation from Section 8.4.4 that the labyrinth problem effectively has two possible ending states. If there are multiple ending states, they may or may not be related to each other in some way. In the labyrinth problem, the two ending states arose from the possibility of the door being either open or closed, but it is also possible to have multiple ending states that are conceptually unrelated.

Item 2 replaces the somewhat-vague notion of "uniformity" originally used in Section 8.1. That notion is replaced by having a fixed set of *potential* moves that remains the same for every state, but allowing only a subset of the moves to actually be legal for each state.

Items 3 and 4 reflect the possibility for the next-state function to not be completely reversible. In particular, items 4a and 4b essentially separate the next-state function into a reversible component, f_R, plus a number of non-reversible components, the f_i's, that can set the value of a flag to 1. Note that the equation for the evolution of a flag from one state to the next, $s'_i = s_i \vee f_i(S_R, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$, guarantees that a flag must remain with a value of 1 permanently once set. A flag therefore behaves like the state of the door in the labyrinth problem: once the door is opened, it remains open permanently. A labyrinth with multiple doors, as mentioned at the end of Section 8.4.4, can be represented by multiple flags, with each flag corresponding to the state of one door. The flag-updating function f_i corresponding to a flag s_i can depend on other flags as well. This possibility was not used in the labyrinth problem considered in this chapter, since there was only one door there, but it could be used to model a labyrinth with multiple doors in which one of the doors can only be opened after another one has been opened first.

We can now readily see how the quantum oracle design strategy used in this chapter can be applied to other state-space path planning problems and partially automate the creation of quantum oracles for them. The move-legality function is represented in the oracle by the

move-restriction circuits, which in general can be automatically synthesized as long as a high-level specification of this function is available. The flag-updating functions are similar and are represented in the oracle by circuits like the door-opening circuit from Section 8.4.2, while the flags themselves are represented using counters like the door-open counter. Just like the move-restriction circuits, these flag-updating circuits can also be automatically synthesized from a high-level specification of the flag-updating functions. If the state-space path planning problem in question has multiple possible ending states, then the ending-state-checking circuit can be represented as yet another Boolean function to be automatically synthesized. Finally, the reversible part of the next-state function, f_R, corresponds to all other portions of the next-state circuit other than the flag-updating circuits. The strategy for designing this part of the oracle is to represent the function f_R in terms of operations such as addition, subtraction, comparison, and copying of values between registers, which can then be implemented by functional blocks such as incrementers, counters, multiplexers, and comparators. Other useful functional blocks such as quantum arithmetic circuits [90] can be included to further increase the generality and applicability of this strategy.

8.6 Conclusion

In this chapter, I presented detailed designs for quantum oracles to solve two problems, the 8-/15-puzzle and a labyrinth problem. The similarity in oracle design strategy used for these problems led to the formulation of a whole class of problems, the state-space path planning problems, for which quantum oracles can also be designed using similar strategies. Specifically, I created an oracle design approach for state-space path planning problems where the oracle at the highest level is composed of several types of subcircuits: a next-state circuit, a move-restriction circuit, and an ending-state-checking circuit. These circuits

are in turn designed using high-level components: functional blocks such as incrementers, decrementers, comparators, and multiplexers, as well as "black boxes" that are defined by a Boolean function and can be automatically decomposed into low-level gates using any realization method for Boolean functions. In the future, this type of approach will allow human designers to therefore design quantum oracles to solve state-space path planning problems by working only with these high-level components; the details of implementing these components on a lower level will be handled automatically by a CAD tool or quantum "compiler". This workflow for quantum circuit design is analogous to modern computer-aided design of digital logic circuits or software programming in high-level languages, where the human designer never needs to work directly with low-level primitives—individual transistors in the case of hardware and machine-language instructions in the case of software.

In the process of designing quantum oracles for the two considered problems, I also demonstrated two other uses for quantum counters. In Chapter 7, a quantum counter was used to compute the cost of an encoding by adding the outputs of multiple dependency checkers together, but here, a quantum counter is instead used as a flag to model an irreversible process using a reversible circuit. This use of a quantum counter also allows a Boolean function represented as the logical OR of many terms to effectively be evaluated without the need to allocate one ancillary qubit to every term.

The class of state-space path planning problems includes problems that are of interest in the areas of robotics and artificial intelligence. For instance, robot path-planning or motion planning problems [91, 92, 93, 94], which involve navigating a robot or a part of a robot like an arm through an environment containing obstacles, can be naturally expressed as state-space path planning problems. The set of states is the configuration space of the robot, discretized to a sufficient degree of precision, and the set of moves consists of the actions

and movements that the robot is capable of performing. Legality of moves is then defined by the obstacles that are present as well as inherent limits to the robot's range of motion. Another example of a state-space path planning problem is the Wumpus World game [95], which is essentially a more complicated version of the labyrinth problem considered in this chapter and has been used as an example to demonstrate artificial intelligence algorithms. In Wumpus World, the player must search for treasure in an environment containing pits and a wumpus, both of which will kill the player. The player also has a single arrow that can kill the wumpus, but can only be used once. When formulated as a state-space path planning problem, the state space for Wumpus World consists of the player's position together with several status variables such as whether they are dead, whether they have shot their arrow, etc. These status variables correspond to flags in the terminology of Section 8.5, since they represent irreversible state changes—once the player has died, they are permanently dead, once they have used up the arrow, they cannot use it again, etc. We therefore see that the approach I introduced in this chapter is applicable to both the robot motion planning and Wumpus World problems as well.

Chapter 9

Summary of contributions

In this dissertation, I presented a general approach to designing quantum algorithms for certain problems of practical interest using Grover's algorithm as a subroutine. I examined the implementations of these algorithms at both low and high levels. At a low level, I created synthesis methods for both non-reversible and reversible Boolean functions, and formalized these methods as algorithms. At a high level, I described a strategy for quantum oracle design in terms of high-level blocks, although this strategy is not completely systematic—it still requires a human designer relying on experience and intuition to perform the first step of creating a high-level design of a quantum oracle. Once such a high-level design has been created, the process of implementing it at a low level is systematic because this process involves realizing Boolean functions as well as using known regular circuit structures to implement components such as counters and multiplexers. To solve optimization problems, I introduced a "hybrid" algorithm that uses Grover's algorithm as a subroutine in a classical algorithm. This algorithm provides an example of cooperation between classical and quantum computing systems, because it would not be possible with only one or the other.

9.1 Contributions to logic synthesis for quantum circuits

The realization of Boolean functions using quantum circuits is of great interest for several reasons. Universality, the ability to compute any arbitrary function that can in principle be computed, is an essential trait of any useful computational system. If the computational system is a quantum device that uses qubits, then universality requires that any Boolean function must be realizable using quantum gates. Furthermore, for a quantum computational device to be as useful as possible, these realizations should be as resource-efficient as possible.

The ability to realize Boolean functions is also a basic capability needed for the creation of CAD tools, and will therefore be an essential ingredient of future automated workflows for quantum algorithm design. Modern digital logic design, especially of VLSI (very-large-scale integrated) chips, relies heavily on design automation. A human designer only needs to write a high-level description of a desired circuit, and CAD tools perform the tasks of synthesizing that circuit on the gate level and generating a transistor layout, transistors being the ultimate low-level primitives of modern digital logic circuits. Similarly, most modern software is written in high-level programming languages, with a compiler automatically translating high-level statements into low-level machine language instructions. An analogous automated workflow for quantum circuit design will allow future designers to work on a larger scale and on a higher level than is currently possible with quantum languages like Qiskit, which describe quantum circuits in terms of low-level gates.

The vast majority of existing strategies for the realization of Boolean functions by quantum circuits use Toffoli gates along with inverters and CNOT gates because they are quantum counterparts of the classical AND, NOT, and exclusive-OR gates and therefore allow some existing Boolean logic realization techniques for classical digital circuits to

be applied to quantum circuits as well. In Chapters 2, 3, and 4, I proposed a completely different alternative approach to universality where symmetric functions are directly realized using low-level quantum gates that have no classical counterpart. Other non-symmetric functions are realized by "symmetrizing" them through the use of repeated variables. Instead of relying on existing Boolean logic realization techniques, my approach uses controlled rotation gates to effectively count the number of 1s present in a given collection of input qubits and thereby realize symmetric functions in a highly efficient manner. In this way, I take full advantage of the quantum nature of the underlying computational system, because rotation gates with angles of less than 180° have no classical counterpart.

More specifically, I first demonstrated in Chapters 2 and 3 that a particular class of symmetric functions, which I call dyadic index-periodic symmetric functions (DIPS), can be realized in an especially efficient manner using only two-qubit controlled root-of-NOT gates. Chapter 4 then showed that symmetric functions could be realized by combining DIPS using a method based on the Walsh-Hadamard transform. This is the first time that the Walsh-Hadamard transform has been used to realize symmetric functions using quantum circuits. This method involved using additional controlled rotation gates to effectively add multiple DIPS together arithmetically, which is difficult and would require complicated adder circuits in classical Boolean logic since all variables therein may only take on binary values. My realization of symmetric functions therefore actually takes advantage of quantum-only gates, the controlled root-of-NOT gates, in two distinct ways. Other than [14], very little existing published work directly leverages quantum-only gates to realize arbitrary Boolean functions in ways that are not possible with classical digital logic circuits.

The concept of realizing arbitrary Boolean functions by "symmetrizing" them, converting them to symmetric functions with repeated variables, was previously mentioned in the context

of quantum circuits by Maslov [33], and such a symmetrization algorithm was described in [48]. However, when combined with DIPS-based realization of symmetric functions, the realization of non-symmetric Boolean functions through symmetrization becomes especially powerful. The nature of the circuit structure used to realize DIPS allows variables to be repeated without the use of ancillary qubits and with no increase in quantum cost whatsoever. As a consequence, any n-variable Boolean function can be realized at a quantum cost no greater than that required to realize an n'-variable symmetric function, where n' is the number of variables (including all repetitions) after the function has been symmetrized. In Chapter 4, I calculated a rigorous upper bound for the quantum cost and number of ancillary qubits required to realize any symmetric function. These upper bounds can be extended to all Boolean functions, symmetric or not, if the number of variable repetitions required to symmetrize the function is known. For symmetric functions, the calculated metrics compare favorably to the results reported by Maslov [33]. Since Maslov's method only uses NOT, CNOT, and Toffoli gates, my results show that advantages can indeed be gained by designing quantum circuits directly on the level of two-qubit controlled rotation gates without using Toffoli gates as intermediates.

I also made contributions to the problem of realizing multiple-input, multiple-output reversible functions "in place" using quantum circuits. "In place" means that the same qubits are used for the inputs and outputs of the function, and such realizations are only possible when the function to be realized is itself reversible. Due to the constraint of reversibility, several types of methods which are not available for single-output functions become available in the multiple-output case. In particular, for reversible functions, there is a whole category of cycle-based methods, which are often well-suited for functions in which only a small fraction of the possible input values are altered. In Chapter 5, I introduced the new concept of

distance gates, which form the foundation for my own cycle-based method. These distance gates realize arbitrary transpositions, which are the smallest possible cycles and can form all other cycles via composition. I also introduced a method of distance gate reduction, which greatly reduces the quantum cost of certain sets of distance gates by combining them into a simplified form. Distance gate reduction effectively reduces the problem of realizing a multiple-input, multiple-output reversible function to that of realizing a sequence of multiple-input, single-output reversible functions. These single-output functions can then be realized using either my proposed symmetric function-based method from Chapters 2, 3, and 4, or other existing methods, making the distance gate-based synthesis approach very flexible. In Chapter 6, I further expanded on the flexibility of distance gates by showing that they can also be adapted for multiple-valued quantum computation. This ability is particularly valuable because of the shortage of published work on realizing reversible functions for multiple-valued quantum computers. Distance gates can be adapted for quantum computers using qudits of any radix, and even for one that combines qubits of different radices in the same circuit. This dissertation is the first published work to demonstrate a reversible circuit synthesis method with this ability.

9.2 Contributions to quantum algorithm design

My contributions to the area of high-level quantum algorithm design include a method by which Grover's algorithm, which directly solves only satisfaction problems, can also be used in solving optimization problems. This is achieved by repeatedly executing Grover's algorithm using different quantum oracles to determine the minimum possible value of whatever cost function is being optimized. My method takes advantage of the reconfigurable nature of quantum circuits, which could be better described as quantum "programs" rather

than "circuits" because they consist of software instructions rather than physical hardware components. This reconfigurability plays a critical role in my "hybrid" algorithm, because it allows the classical computer that is controlling the quantum hardware executing Grover's algorithm to modify the oracle used in a subsequent execution of Grover's algorithm based on the result of a previous execution.

In Chapters 7 and 8, I also demonstrated the method that the classical computer in such a hybrid algorithm uses to modify the quantum oracle. For two important classes of practical problems, I showed that appropriate quantum oracles can be created using modular designs that reuse a small set of functional circuits and are easily scaled up and down. The process of scaling up and down is completely systematic and can therefore be automated. The ability to automatically generate different versions of an oracle from the same overall design enables hybrid classical-quantum algorithms to work—if every iteration of Grover's algorithm required a new oracle to be designed manually from scratch, the process of executing Grover's algorithm multiple times in an adaptive manner would be impossible to automate.

For instance, in Chapter 7, for the state encoding problem, I presented an oracle design that is easily modified to search for encodings with any given maximum cost by simply changing a single threshold circuit. This oracle design is also easily scalable to solve the encoding problem for state machines with any number of states and encodings with any number of bits. The state encoding problem is one of the most important problems in classical automata theory and my algorithm finds the exact minimum solution using an exhaustive search. An exhaustive search on a classical computer quickly becomes impractical as the number of states increases, but the quadratic speedup provided by Grover's algorithm will allow the exhaustive search to remain practical for a larger number of states, as long as a

quantum computer with sufficiently many qubits is available.

Another contribution made in Chapter 7 was to demonstrate the utility of quantum counters in quantum oracle design. This provides a practical use for the DIPS-based quantum counter design introduced in Chapter 3. Quantum counters are resource-efficient because the size of the counter only grows logarithmically with the maximum value it must handle, and the DIPS-based counter design from Chapter 3 has a quantum cost that grows linearly with the number of signals being counted and quadratically with the number of counter qubits.

In Chapter 8, I created an approach to the design of quantum oracles for state-space path planning problems. Like the oracle design from Chapter 7, the resulting quantum oracles have a modular design based on next-state circuits, move-restriction circuits and an ending-state-checking circuit. They may similarly be scaled up and down to search for paths with any given maximum number of moves—the maximum number of moves is increased by simply adding additional next-state circuits and move-restriction circuits to the oracle and increasing the size of the invalid move counter, if needed. The next-state circuits are designed using functional blocks like counters and multiplexers, while the move-restriction and ending-state-checking circuits can be obtained by realizing one or more Boolean functions and possibly adding some simple connecting logic between them. All of the component subcircuits used in the oracle can therefore be designed using a partially-automated workflow, where a human designer produces a high-level design of the next-state circuit and specifications of the Boolean functions used in the move-restriction and ending-state-checking circuits, and a CAD tool automatically generates their low-level realizations using quantum gates. In the future, this type of workflow will allow very large quantum oracles, perhaps involving tens of thousands of qubits and hundreds of thousands of gates or more, to be designed without too much difficulty.

References

[1] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, no. 6/7, 1982.

[2] D. Simon, "On the power of quantum computation," in *Proc. 35th Annual Symposium on Foundations of Computer Science*, pp. 116–123, IEEE, 1994.

[3] D. R. Simon, "On the power of quantum computation," *SIAM journal on computing*, vol. 26, no. 5, pp. 1474–1483, 1997.

[4] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proc. 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, IEEE, 1994.

[5] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[6] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annual ACM Symposium on Theory of Computing*, pp. 212–219, May 1996.

[7] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Phys. Rev. Lett.*, vol. 79, pp. 325–328, July 1997.

[8] L. K. Grover, "A framework for fast quantum mechanical algorithms," in *Proc. 30th Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), pp. 53–62, ACM, 1998.

[9] Gambetta, Jay, "IBM's roadmap for scaling quantum technology." https://research.ibm.com/blog/ibm-quantum-roadmap, 2020. Accessed: 2022-4-18.

[10] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[11] Emily Conover, "Google claimed quantum supremacy in 2019—and sparked controversy." https://www.sciencenews.org/article/google-quantum-supremacy-claim-controversy-top-science-stories-2019-yir, 2019. Accessed: 2022-4-18.

[12] Qiskit Development Team, "Qiskit 0.36.0 documentation." https://qiskit.org/documentation/index.html, 2021. Accessed: 2022-4-18.

[13] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary ed., 2010.

[14] M. Szyprowski and P. Kerntopf, "Low quantum cost realization of generalized Peres and Toffoli gates with multiple-control signals," in *Proc. 13th IEEE International Conference on Nanotechnology*, pp. 802–807, IEEE, 2013.

[15] T. Sasao, "EXMIN2: A simplification algorithm for exclusive-OR-sum-of-products expressions for multiple-valued-input two-valued-output functions," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 621–632, May 1993.

[16] N. Song and M. A. Perkowski, "EXORCISM-MV-2: minimization of exclusive sum of products expressions for multiple-valued input incompletely specified functions," in *Proc. 23rd International Symposium on Multiple-Valued Logic*, pp. 132–137, IEEE, 1993.

[17] A. Mishchenko and M. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," in *Proc. 5th International Reed-Muller Workshop*, pp. 242—250, Aug. 2001.

[18] J. M. Gambetta, J. M. Chow, and M. Steffen, "Building logical qubits in a superconducting quantum computing system," *npj Quantum Information*, vol. 3, p. 2, Jan. 2017.

[19] D. Maslov and G. Dueck, "Improved quantum cost for *n*-bit Toffoli gates," *Electronics Lett.*, vol. 39, pp. 1790–1791, Dec. 2003.

[20] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Phys. Rev. A*, vol. 52, pp. 3457–3467, Nov. 1995.

[21] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, "Tight bounds on quantum searching," *Fortschritte der Physik*, vol. 46, no. 4–5, pp. 493–505, 1998.

[22] G. Brassard, P. Høyer, and A. Tapp, "Quantum counting," in *Automata, Languages and Programming*, pp. 820–831, Springer Berlin Heidelberg, 1998.

[23] J. A. Smolin and D. P. DiVincenzo, "Five two-bit quantum gates are sufficient to implement the quantum Fredkin gate," *Phys. Rev. A*, vol. 53, no. 4, p. 2855, 1996.

[24] M. A. Harrison, *Introduction to Switching and Automata Theory*. New York: McGraw-Hill, 1965.

[25] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1970.

[26] I. Wegener, "The complexity of symmetric Boolean functions," *Computation theory and logic*, pp. 433–442, 1987.

[27] L. Babai, P. Pudlák, V. Rödl, and E. Szemerédi, "Lower bounds to the complexity of symmetric Boolean functions," *Theoretical Computer Science*, vol. 74, no. 3, pp. 313–323, 1990.

[28] C. Moraga and R. S. Stanković, "Properties of the Reed-Muller spectrum of symmetric functions," *Facta Universitatis, Series: Electronics and Energetics*, vol. 20, no. 3, pp. 281–294, 2007.

[29] E. Demenkov, A. Kojevnikov, A. Kulikov, and G. Yaroslavtsev, "New upper bounds on the Boolean circuit complexity of symmetric functions," *Information Processing Lett.*, vol. 110, no. 7, pp. 264–267, 2010.

[30] M. Perkowski, P. Kerntopf, A. Buller, M. Chrzanowska-Jeske, A. Mishchenko, X. Song, A. Al-Rabadi, L. Jozwiak, and A. Coppola, "Regularity and symmetry as a base for efficient realization of reversible logic circuits," in *International Workshop on Logic Synthesis*, 2001.

[31] D. A. Maslov, "Dynamic programming algorithms as quantum circuits: symmetric function realization," in *Quantum Information and Computation II* (E. Donkor, A. R. Pirich, and H. E. Brandt, eds.), vol. 5436, pp. 386–393, International Society for Optics and Photonics, SPIE, 2004.

[32] S.-H. Kim and S. Choi, "Scalable systolic structure to realize arbitrary reversible symmetric functions," *GESTS Int'l Trans. on Computer Science and Engineering (Korea)*, vol. 18, no. 1, pp. 27–36, 2005.

[33] D. Maslov, "Efficient reversible and quantum implementations of symmetric Boolean functions," *IEE Proc. Circuits, Devices and Systems*, vol. 153, no. 5, pp. 467–472, 2006.

[34] A. Deb, D. K. Das, H. Rahaman, and B. B. Bhattacharya, "Reversible synthesis of symmetric Boolean functions based on unate decomposition," in *Proc. 23rd ACM international conference on Great lakes symposium on VLSI*, pp. 351–352, 2013.

[35] A. Deb, D. K. Das, H. Rahaman, B. B. Bhattacharya, R. Wille, and R. Drechsler, "Reversible circuit synthesis of symmetric functions using a simple regular structure," in *Proc. 5th international conference on Reversible Computation*, pp. 182–195, 2013.

[36] P. Sarkar, B. Mondal, A. K. Pramanik, S. Chakraborty, and R. Duttagupta, "Symmetric function realization using reversible circuit synthesis," in *TENCON 2014—2014 IEEE Region 10 Conference*, pp. 1–6, IEEE, 2014.

[37] A. Deb, D. K. Das, H. Rahaman, R. Wille, R. Drechsler, and B. B. Bhattacharya, "Reversible synthesis of symmetric functions with a simple regular structure and easy testability," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 12, no. 4, pp. 1–29, 2016.

[38] E. Tsai and M. A. Perkowski, "Synthesis of permutative quantum circuits with Toffoli and TISC gates," in *Proc. 42nd International Symposium on Multiple-Valued Logic*, pp. 50–56, IEEE, 2012.

[39] K. G. Beauchamp, *Applications of Walsh and related functions, with an introduction to sequency theory*, vol. 2. New York: Academic Press, 1984.

[40] B. Golubov, A. Efimov, and V. Skvortsov, *Walsh series and transforms: theory and applications*, vol. 64 of *Mathematics and Applications: Soviet Series*. Boston: Kluwer Academic Publishers, 2012.

[41] J. R. Bitner, G. Ehrlich, and E. M. Reingold, "Efficient generation of the binary reflected Gray code and its applications," *Commun. ACM*, vol. 19, pp. 517–521, Sept. 1976.

[42] J. L. Shanks, "Computation of the fast Walsh-Fourier transform," *IEEE Trans. on Computers*, vol. C-18, no. 5, pp. 457–459, 1969.

[43] R. Born and A. Scidmore, "Transformation of switching functions to completely symmetric switching functions," *IEEE Trans. on Computers*, vol. C-17, no. 6, pp. 596–599, 1968.

[44] S. Yau and Y. Tang, "Transformation of an arbitrary switching function to a totally symmetric function," *IEEE Trans. on Computers*, vol. C-20, no. 12, pp. 1606–1609, 1971.

[45] R. Born, "An iterative technique for determining the minimal number of variables for a totally symmetric function with repeated variables," *IEEE Trans. on Computers*, vol. C-21, no. 10, pp. 1129–1131, 1972.

[46] B. Dahlberg, "On symmetric functions with redundant variables—weighted functions," *IEEE Trans. on Computers*, vol. C-22, no. 5, pp. 450–458, 1973.

[47] D. T. Lee and S. J. Hong, "An algorithm for transformation of an arbitrary switching function to a completely symmetric function," *IEEE Trans. on Computers*, vol. C-25, no. 11, pp. 1117–1123, 1976.

[48] M. Chrzanowska-Jeske, Y. Xu, and M. A. Perkowski, "Logic synthesis for a regular layout," *VLSI Design*, vol. 10, no. 1, pp. 35–55, 1999.

[49] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Proc. 40th Design Automation Conference (DAC '03)*, pp. 318–323, June 2003.

[50] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 710–722, June 2003.

[51] M. Saeedi, M. S. Zamani, M. Sedighi, and Z. Sasanian, "Reversible circuit synthesis using a cycle-based approach," *J. Emerg. Technol. Comput. Syst.*, vol. 6, pp. 13:1–13:26, Dec. 2010.

[52] P. Gupta, A. Agrawal, and N. K. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 2317–2330, Nov. 2006.

[53] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Proc. 46th Design Automation Conference (DAC '09)*, (New York, NY, USA), pp. 270–275, ACM, 2009.

[54] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits—a survey," *ACM Comput. Surv.*, vol. 45, pp. 21:1–21:34, Mar. 2013.

[55] N. Song and M. A. Perkowski, "Minimization of exclusive sum-of-products expressions for multiple-valued input, incompletely specified functions," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 4, pp. 385–395, 1996.

[56] M. Kumar, "Realization of incompletely specified reversible functions," Master's thesis, Portland State University, 2008.

[57] M. Perkowski, R. Fiszer, P. Kerntopf, and M. Lukac, "An approach to synthesis of reversible circuits for partially specified functions," in *12th IEEE International Conference on Nanotechnology (IEEE-NANO)*, pp. 1–6, IEEE, 2012.

[58] R. A. Fiszer, "Synthesis of irreversible incompletely specified multi-output functions to reversible EOSOPS multi-output circuits with PSE gates," Master's thesis, Portland State University, 2014.

[59] M. Rawski and P. Szotkowski, "Reversible synthesis of incompletely specified Boolean functions using functional decomposition," in *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, vol. 10445, pp. 806–813, SPIE, 2017.

[60] J. I. Cirac and P. Zoller, "Quantum computations with cold trapped ions," *Phys. Rev. Lett.*, vol. 74, pp. 4091–4094, May 1995.

[61] A. Muthukrishnan and C. R. Stroud, "Multivalued logic gates for quantum computation," *Phys. Rev. A*, vol. 62, p. 052309, Oct. 2000.

[62] A. B. Klimov, R. Guzmán, J. C. Retamal, and C. Saavedra, "Qutrit quantum computer with trapped ions," *Phys. Rev. A*, vol. 67, p. 062313, June 2003.

[63] X. Wang, B. C. Sanders, and D. W. Berry, "Entangling power and operator entanglement in qudit systems," *Phys. Rev. A*, vol. 67, p. 042323, Apr. 2003.

[64] S. S. Bullock, D. P. O'Leary, and G. K. Brennen, "Asymptotically optimal quantum circuits for *d*-level systems," *Phys. Rev. Lett.*, vol. 94, p. 230502, June 2005.

[65] G. K. Brennen, S. S. Bullock, and D. P. O'Leary, "Efficient circuits for exact-universal computation with qudits," *Quantum Inf. Comput.*, vol. 6, no. 4–5, pp. 436–454, 2006.

[66] S. D. Bartlett, H. de Guise, and B. C. Sanders, "Quantum encodings in spin systems and harmonic oscillators," *Phys. Rev. A*, vol. 65, p. 052316, May 2002.

[67] Y. I. Bogdanov, M. V. Chekhova, S. P. Kulik, G. A. Maslennikov, A. A. Zhukov., C. H. Oh, and M. K. Tey, "Qutrit state engineering with biphotons," *Phys. Rev. Lett.*, vol. 93, p. 230503, Dec. 2004.

[68] R. Bianchetti, S. Filipp, M. Baur, J. M. Fink, C. Lang, L. Steffen, M. Boissonneault, A. Blais, and A. Wallraff, "Control and tomography of a three level superconducting artificial atom," *Phys. Rev. Lett.*, vol. 105, p. 223601, Nov. 2010.

[69] S. X. Cui and Z. Wang, "Universal quantum computation with metaplectic anyons," *J. Mathematical Physics*, vol. 56, no. 3, p. 032202, 2015.

[70] A. Bocharov, X. Cui, V. Kliuchnikov, and Z. Wang, "Efficient topological compilation for a weakly integral anyonic model," *Phys. Rev. A*, vol. 93, p. 012313, Jan. 2016.

[71] T. C. Ralph, K. J. Resch, and A. Gilchrist, "Efficient Toffoli gates using qudits," *Phys. Rev. A*, vol. 75, p. 022313, Feb. 2007.

[72] Y.-M. Di and H.-R. Wei, "Synthesis of multivalued quantum logic circuits by elementary gates," *Phys. Rev. A*, vol. 87, p. 012325, Jan. 2013.

[73] C. Moraga, "On some basic aspects of ternary reversible and quantum computing," in *IEEE 44th International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 178–183, 2014.

[74] V. Deibuk, I. Turchenko, and V. Shults, "Optimized design of the universal ternary gates for quantum/reversible computing," in *IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, pp. 987–991, 2015.

[75] C. Moraga, "Quantum *p*-valued Toffoli and Deutsch gates with conjunctive or disjunctive mixed polarity control," in *IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 241–246, 2016.

[76] A. Bocharov, S. X. Cui, M. Roetteler, and K. M. Svore, "Improved quantum ternary arithmetics," *Quantum Inf. Comput.*, vol. 16, July 2016.

[77] J. Hartmanis, "On the state assignment problem for sequential machines. I," *IRE Trans. on Electronic Computers*, vol. EC-10, pp. 157–165, June 1961.

[78] R. E. Stearns and J. Hartmanis, "On the state assignment problem for sequential machines. II," *IRE Trans. on Electronic Computers*, vol. EC-10, pp. 593–603, Dec. 1961.

[79] L. Benini and G. D. Micheli, "State assignment for low power dissipation," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 258–268, Mar. 1995.

[80] G. D. Hachtel, M. Hermida, A. Pardo, M. Poncino, and F. Somenzi, "Re-encoding sequential circuits to reduce power dissipation," in *Proc. 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, (Los Alamitos, CA, USA), pp. 70–73, IEEE Computer Society Press, 1994.

[81] G. D. Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, pp. 269–285, July 1985.

[82] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 905–924, Sept. 1990.

[83] S. Devadas and A. Newton, "Exact algorithms for output encoding, state assignment, and four-level Boolean minimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 13–27, 1991.

[84] E. F. Moore, "Gedanken experiments on sequential machines," in *Automata Studies*, pp. 129–153, Princeton U., 1956.

[85] J. Hartmanis, "Loop-free structure of sequential machines," *Information and Control*, vol. 5, no. 1, pp. 25–43, 1962.

[86] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[87] D. Ratner and M. Warmuth, "Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable," in *Proc. 5th AAAI National Conference on Artificial Intelligence*, AAAI'86, pp. 168–172, AAAI Press, 1986.

[88] D. Ratner and M. Warmuth, "The $(n^2 - 1)$-puzzle and related relocation problems," *J. Symbolic Computation*, vol. 10, no. 2, pp. 111–137, 1990.

[89] R. E. Korf, "Recent progress in the design and analysis of admissible heuristic functions," in *Abstraction, Reformulation, and Approximation*, (Berlin, Heidelberg), pp. 45–55, Springer Berlin Heidelberg, 2000.

[90] M. K. Thomsen, R. Glück, and H. B. Axelsen, "Reversible arithmetic logic unit for quantum arithmetic," *J. Physics A: Mathematical and Theoretical*, vol. 43, p. 382002, Aug. 2010.

[91] J.-C. Latombe, *Robot motion planning*. Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[92] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[93] S. S. Ge and Y. J. Cui, "New potential functions for mobile robot path planning," *IEEE Trans. on Robotics and Automation*, vol. 16, no. 5, pp. 615–620, 2000.

[94] A. Ismail, A. Sheta, and M. Al-Weshah, "A mobile robot path planning using genetic algorithm in static environment," *J. Computer Science*, vol. 4, no. 4, pp. 341–344, 2008.

[95] D. Bryce, "Wumpus world in introductory artificial intelligence," *J. Comput. Sci. in Colleges*, vol. 27, no. 2, pp. 58–65, 2011.