A Quantum Algorithm for Automata Encoding

Edison Tsai

Marek Perkowski
Portland State University, marek.perkowski@pdx.edu

Citation Details
Tsai, E., & Perkowski, M. (2020). A QUANTUM AlgoRIThM FoR AUToMATA ENCodINg. Facta Universitatis, Series: Electronics and Energetics, 33(2), 169-215.

FACTA UNIVERSITATIS
Series: **Electronics and Energetics** Vol. 33, No 2, June 2020, pp. 169 - 215
https://doi.org/10.2298/FUEE2002169T

A QUANTUM ALGORITHM FOR AUTOMATA ENCODING

Edison Tsai, Marek Perkowski

Department of Electrical and Computer Engineering, Portland State University, USA

Abstract. *Encoding of finite automata or state machines is critical to modern digital logic design methods for sequential circuits. Encoding is the process of assigning to every state, input value, and output value of a state machine a binary string, which is used to represent that state, input value, or output value in digital logic. Usually, one wishes to choose an encoding that, when the state machine is implemented as a digital logic circuit, will optimize some aspect of that circuit. For instance, one might wish to encode in such a way as to minimize power dissipation or silicon area. For most such optimization objectives, no method to find the exact solution, other than a straightforward exhaustive search, is known.*
Recent progress towards producing a quantum computer of large enough scale to surpass modern supercomputers has made it increasingly relevant to consider how quantum computers may be used to solve problems of practical interest. A quantum computer using Grover's well-known search algorithm can perform exhaustive searches that would be impractical on a classical computer, due to the speedup provided by Grover's algorithm. Therefore, we propose to use Grover's algorithm to find optimal encodings for finite state machines via exhaustive search. We demonstrate the design of quantum circuits that allow Grover's algorithm to be used for this purpose. The quantum circuit design methods that we introduce are potentially applicable to other problems as well.

Key words: *Quantum Algorithm, Automata Encoding, Finete State Machines.*

1 Introduction

Although the concept of quantum computing has existed for decades, only recently has it appeared that quantum computers of sufficient scale to solve realistic problems may become available in the near future.[1] The development of such quantum computers is of great interest because they are capable of solving certain classes of problems with a time complexity better than the best achievable with a classical computer. Grover's

Received September 7, 2017; received in revised form January 28, 2020
Corresponding author: Marek Perkowski
Department of Electrical and Computer Engineering, Portland State University, USA
(E-mail: mperkows@ee.pdx.edu)

[1]See [1], Introduction, paragraph 6: "A physical quantum computer. . . is still an outstanding challenge. However, in recent work, physical qubits. . . have reached the point where errors are at or below the threshold, and networks of 4–9 superconducting qubits with individual control and readout have been used to show concepts of error correction. Over the next few years, the field will be in a stage of building interesting quantum devices with a complexity that could never be emulated in full generality on a classical computer ($\sim$50 or more qubits)."

well-known search algorithm [2, 3, 4] provides a commonly cited example of such a class of problems. However, despite the theoretical capabilities of quantum computing, virtually no work has been published that demonstrates a specific, concrete example of Grover's algorithm (or another quantum algorithm) applied to solve a problem of practical interest. This gap in the literature has become increasingly relevant as large-scale quantum computers move ever closer to reality.[2]

To fill this gap, we demonstrate in this work how Grover's algorithm can be applied to the problem of state encoding for finite state machines (FSMs). A well-known problem in digital logic design, which has been recognized for over 50 years [6, 7] 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 [6]. Usually, one wishes to find a state encoding which is minimal with respect to some metric, *e.g.*, power [8, 9] or silicon area [10, 11] 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 [12] has attempted to find exact minimum solutions to this problem. Furthermore, the methods in [12] 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 [12] 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 [12] for a quantum computer. Instead, we use a simplified cost metric from [6, 7], 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.

[2]See above; see also [5], Abstract: "Advanced quantum simulation experiments have been shown with up to nine qubits, while a demonstration of quantum supremacy with fifty qubits is anticipated in just a few years. Quantum supremacy means that the quantum system can no longer be simulated by the most powerful classical supercomputers."

2 Finite State Machines and State Encodings

2.1 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:

$$\mathrm{S}_{t+1} = \delta(\mathrm{S}_t, \mathrm{I}_t),$$

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 [13, 14] whereas the latter corresponds to a so-called Mealy machine [15, 14]. 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 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 $\mathbf{I}$, 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 $\mathbf{S}$, 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

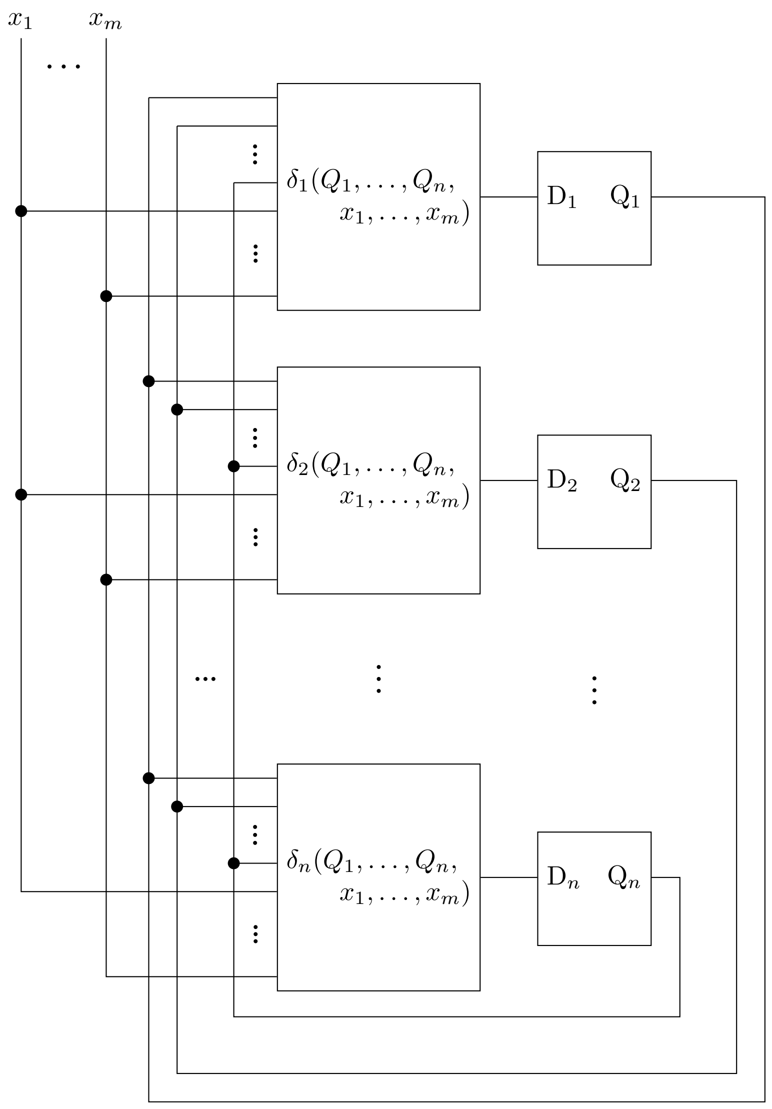

x_1
x_m
$\delta_1(Q_1, \ldots, Q_n, x_1, \ldots, x_m)$
$\delta_2(Q_1, \ldots, Q_n, x_1, \ldots, x_m)$
$\delta_n(Q_1, \ldots, Q_n, x_1, \ldots, x_m)$
D_1 Q_1
D_2 Q_2
D_n Q_n

Figure 1: General structure of an FSM implemented as a digital logic circuit; output logic not shown.

variables Q_1 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 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.

2.2 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 2(a). Figure 2(b) 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 2(c) then shows the resulting encoded transition functions. These functions may be represented by the following logical expressions:

$$\begin{aligned} Q_1^+ = {} & (Q_1 \wedge x_2) \vee (Q_1 \wedge \neg Q_2 \wedge x_1) \vee (\neg Q_1 \wedge Q_2 \wedge x_1) \\ & \vee (Q_2 \wedge \neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2), \end{aligned}$$

$$\begin{aligned} Q_2^+ = {} & (Q_1 \wedge Q_2 \wedge \neg x_1) \vee (Q_1 \wedge Q_2 \wedge \neg x_2) \vee (\neg Q_1 \wedge \neg Q_2 \wedge \neg x_1) \\ & \vee(\neg Q_1 \wedge \neg Q_2 \wedge \neg x_2) \vee (\neg x_1 \wedge \neg x_2), \end{aligned}$$

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 2(d) is used instead, then the encoded transition functions are as shown in Figure 2(e) and may be represented by the expressions

$$Q_1^+ = Q_2 \vee \neg x_1,$$

$$Q_2^+ = (Q_1 \wedge \neg x_1) \vee (Q_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2),$$

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).

We therefore see that between these two encodings, the encoding from Figure 2(d) 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 2(d) to ultimately result in a

	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)

S	Q_1Q_2
S_1	0 0
S_2	0 1
S_3	1 0
S_4	1 1

I	$x_1\,x_2$
I_1	0 0
I_2	0 1
I_3	1 0
I_4	1 1

(b)

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)

S	Q_1Q_2
S_1	0 1
S_2	1 0
S_3	1 1
S_4	0 0

I	$x_1\,x_2$
I_1	1 0
I_2	1 1
I_3	0 1
I_4	0 0

(d)

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)

Figure 2: (a) Transition table for an FSM; (b) an encoding for that FSM; (c) resulting encoded transition functions from that encoding; (d) another encoding for the same FSM; (e) resulting encoded transition functions.

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 2(d) 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 [12]. However, the authors in [12] 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 [6] and [7].

3 Grover's Search Algorithm

3.1 Operation of Grover's Algorithm

In this section we briefly review Grover's algorithm and its relevance to the state encoding problem. We assume the reader's familiarity with quantum computing in general, and in particular with qubits, quantum states, circuits, gates, and the matrix-vector representations thereof. For

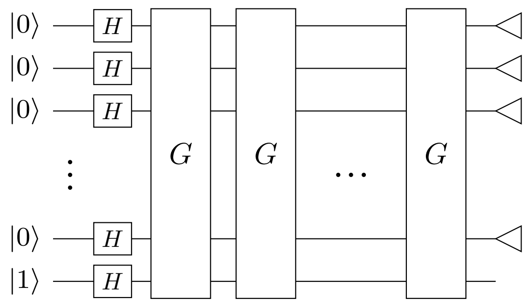


Figure 3: A high-level schematic for Grover's algorithm.

a detailed treatment of the preceding concepts, we refer the reader to [16]. We concentrate on reviewing the conceptual aspects of Grover's algorithm rather than proving that the algorithm functions as claimed. Full proofs of the validity of Grover's algorithm are well known and may be found, for instance, in [3, 16].

Grover's algorithm [2, 3, 4] is a well-known quantum algorithm and one of the first to demonstrate how quantum computers are in a sense more powerful than classical computers. Specifically, Grover's algorithm solves the problem of satisfying a Boolean function; that is, given a function $f : \{0,1\}^n \to \{0,1\}$, find x_1 through x_n such that[3] $f(x_1, \dots, x_n) = 1$. This type of problem is commonly known as a satisfaction or decision problem. Classically, assuming no additional information is known regarding the function f, one can on average solve this problem no faster than by exhaustive search. 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 $\mathcal{O}(\sqrt{N})$ time, a quadratic speedup.

We first describe Grover's algorithm with the assumption that exactly one solution exists; that is, there is exactly one assignment of the values $\{0,1\}$ to x_1 through x_n such that $f(x_1, \dots, x_n) = 1$. This is the original scenario considered by Grover. Later, we will relax this assumption and describe how the algorithm can be extended to cases where there is more than one solution, or no solution at all.

Figure 3 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 a 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 amplitude of a certain component state of a superposition, while decreasing the amplitudes of all other

[3]The variables x_1 through x_n in this case are simply inputs to the function f and are unrelated to the input variables of an FSM.

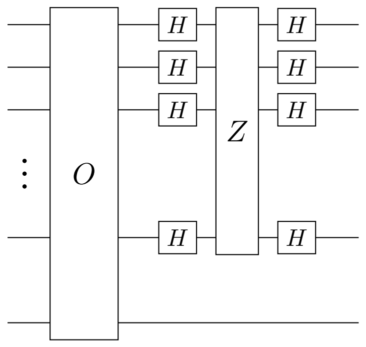


Figure 4: Structure of the Grover iterate G from Figure 3 with quantum oracle O and zero-state phase shift Z.

component states. In Grover's algorithm, the component state whose amplitude is increased corresponds to the variable assignment satisfying the function f. By applying the Grover iterate enough times to a superposition, one obtains a quantum state where the component corresponding to the satisfactory variable assignment has an amplitude near 1; all other components have zero or negligibly small amplitudes. A measurement then gives (with probability near 1) a variable assignment satisfying f.

Figure 4 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 acts as a "controlled inverter" controlled by f. That is, the quantum oracle acts on qubits x_1 through x_n together with an additional *output qubit*, and, for each possible basis state of x_1 through x_n, the oracle inverts the output qubit if and only if $f(x_1, \ldots, x_n) = 1$. Figure 5 depicts an alternative schematic symbol for the oracle that graphically suggests this "controlled" nature.

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. In the subsequent sections of this paper, we will concentrate on how to design a quantum oracle that can, when used in Grover's algorithm, help find the optimal state encoding for an FSM. 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 component of a superposition that corresponds to the variable assignment satisfying the function f—from now on, we will

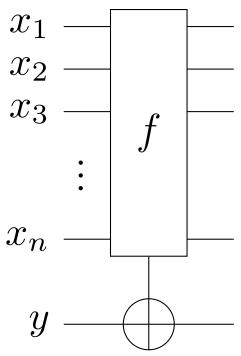


Figure 5: Controlled inverter representation of a quantum oracle.

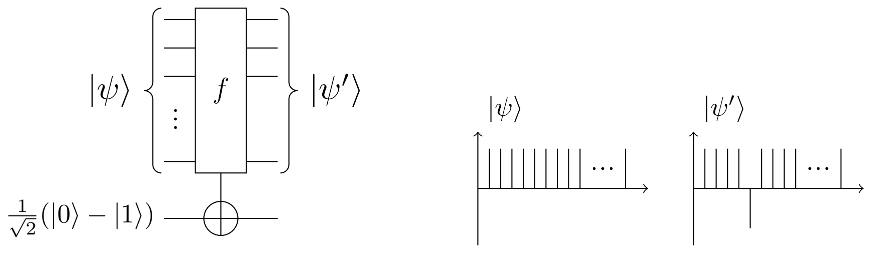


Figure 6: (a) Quantum oracle set up to perform a phase flip; (b) amplitude-graph visualization of the phase flip.

refer to this component as the "solution" component since it represents the solution to the satisfaction problem. Specifically, one can prove that if one supplies the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ on the oracle's output qubit (y in Figure 5) and supplies any state $|\psi\rangle$ on the input qubits, then the oracle inverts the phase of the solution component of $|\psi\rangle$. Figure 6(a) illustrates the quantum circuit diagram for an oracle set up to perform the phase inversion, showing the output qubit initialized to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Figure 6(b) then visualizes the initial and final states, $|\psi\rangle$ and $|\psi'\rangle$, of the oracle's input qubits by plotting their component amplitudes. In other words, given the state $|\psi\rangle$ (or $|\psi'\rangle$) expressed as a sum of components,

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

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.

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 so-

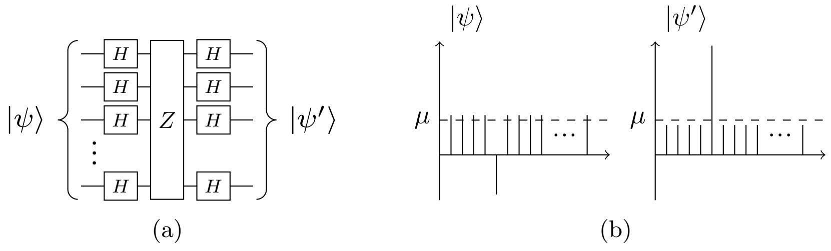


Figure 7: (a) Circuit to perform inversion about the mean; (b) amplitude-graph visualization.

lution is encoded as a phase variation which is not directly observable. This problem is solved by the second part of Grover's algorithm—the amplitude amplification procedure, which converts this phase difference into a measurable amplitude difference. This procedure is seen on the right-hand side of Figure 4 and involves the Hadamard gates together with the zero-state phase shift Z. The zero-state phase shift is defined to invert the phase of the $|0\rangle$ component of any quantum state on which it acts. One can prove that when Hadamard gates are applied before and after a zero-state phase shift as shown in Figure 4, the result is an *inversion about the mean*, defined as follows: if μ denotes the mean amplitude of 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:

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

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

Figure 7 depicts the effect of inversion about the mean when applied immediately following the quantum oracle. Figure 7(a) shows the portion of the Grover iterate G (from Figures 3 and 4) that performs inversion about the mean. Figure 7(b) then visualizes the initial and final quantum states in the same manner as Figure 6. We see that following inversion about the mean, the amplitudes of the solution component has increased while those of all other components have decreased. Successive application of the Grover iterate similarly increases the amplitude of the solution component further. After enough iterations, the superposition consists entirely or nearly entirely of only the solution component. 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). Then, a measurement gives (with near certainty) a variable assignment satisfying the function f.

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 input qubit (x_1 through x_n) to $|0\rangle$ and then applying a Hadamard gate to each input qubit, as shown in Figure 3. Similarly, the quantum oracle's output qubit is initialized to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ by applying a Hadamard gate to an initial state of $|1\rangle$, also shown in Figure 3. 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; Figure 3 indicates this measurement on the right-hand side using the standard left-pointing triangular symbol. We therefore see that the complete Grover's algorithm consists of the following steps:

1. Begin with n input qubits and one output 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 input 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 $\mathcal{O}(\sqrt{N})$ 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 $\mathcal{O}(\sqrt{N})$.

Equipped with an understanding of Grover's original algorithm, which assumes that exactly one solution exists, we now consider how the algorithm can be modified if more than one solution exists, or if there are no solutions at all. There are several possible variations of this scenario, depending on one's assumptions and objectives:

- One may or may not know the number of solutions beforehand.
- If at least one solution exists, one may wish to find just a single solution, or all possible solutions.
- One may or may not know beforehand whether at least one solution exists.
- If the existence of at least one solution is not guaranteed, one may be interested only in determining the existence or non-existence of a solution, and not in actually finding that solution.

For our purposes, we will need to assume that we do not have prior knowledge of the existence and number of solutions. We will be interested in first determining whether any solutions exist, and then finding just a single solution if at least one exists. In Section 3.2, we will demonstrate that satisfying this requirement suffices to solve the state encoding problem using a quantum computer.

Fortunately, for the assumptions stated above, one can extend Grover's algorithm to the case of an unknown number of solutions while maintaining an $\mathcal{O}(\sqrt{N})$ run time complexity. In [17], the authors show that if one has foreknowledge of the exact number of solutions, Grover's original algorithm as previously described (which assumes the existence of exactly one solution) can still be used with one modification: instead of $\mathcal{O}(\sqrt{N})$ iterations, one now requires $\mathcal{O}(\sqrt{N/k})$ iterations, where k is the number of solutions. Furthermore, the authors in [17] also demonstrate a method by which one can find a solution in $\mathcal{O}(\sqrt{N/k})$ expected time even if the number of solutions is unknown. This same method is additionally capable of detecting the non-existence of a solution in $\mathcal{O}(\sqrt{N})$ time with an arbitrarily high level of certainty. Alternatively, Brassard *et al.* [18] have described a *quantum counting* algorithm, with which one may obtain an estimate of the number of solutions. One could use this quantum counting algorithm together with the previously-mentioned extended Grover's algorithm for a known number of solutions (in [17]) to find a single solution, if it exists.[4]

Using the above approaches, one can satisfy the previously stated requirement—determine whether any solutions exist, and find one if it does—with the same $\mathcal{O}(\sqrt{N})$ time complexity as Grover's original algorithm. For the sake of brevity, we will from now on use "Grover's algorithm" to refer to all of these approaches collectively. In other words, when we say we are "using Grover's algorithm" to solve a satisfaction problem, we mean that we are using any of the above approaches to determine if a solution to the problem exists, and then find one if it does.

There is one more aspect of Grover's algorithm that we have not yet mentioned—the use of *ancillary ("ancilla") qubits* in the quantum oracle. Strictly speaking, ancilla qubits are an aspect of quantum oracle design and not of Grover's algorithm itself. We discuss them here nevertheless, since we will concentrate on designing a quantum oracle to solve the FSM state encoding problem in the remainder of this paper. Ancilla qubits are extra qubits that are used by a quantum oracle to store intermediate results while it is performing its computations. They are initialized to a value of the quantum circuit designer's choice. A quantum oracle only uses ancilla qubits internally; externally, it should appear as if ancilla qubits simply "pass through" the oracle with no change. In other words, ancilla qubits do not externally appear to participate in the oracle's operation at all; they participate internally and only for the purpose of acting as temporary storage. Ancilla qubits are usually employed to reduce the quantum circuit cost of a quantum oracle. A quantum circuit using ancilla qubits often has lower quantum cost than an equivalent circuit without ancilla qubits. For instance, Figure 8 shows a quantum oracle for the function $f = (x_1 + x_2)(x_3 + x_4)$ using two ancilla qubits. It is possible to design a quantum oracle for this function without ancilla qubits, but the

[4]Strictly speaking, the viability of this second method is actually not absolutely clear, although it should likely work. The difficulty is that the quantum counting algorithm in [18] only provides an approximate estimate of the number of solutions, while [17] assumes knowledge of the exact number of solutions. A brief consideration of the mathematical analysis used in [17] suggests that the method will still succeed with high probability with only an estimate of the number of solutions, as long as this estimate is reasonably accurate. However, the authors in [17] do not rigorously prove this assertion. In any case, the validity of our present work is unaffected since [17] also give a method for finding one of an unknown number of solutions, as mentioned in the main text.

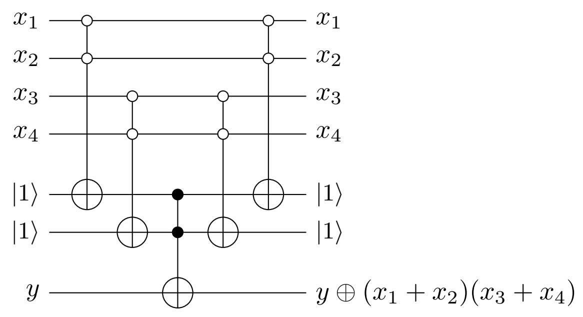


Figure 8: A quantum oracle using ancilla qubits.

quantum circuit cost of such an oracle would be higher than the cost of the oracle in Figure 8. In many cases, creating a quantum oracle without using ancilla qubits is in fact completely impractical.

Although the judicious employment of ancilla qubits can greatly reduce quantum circuit costs, one must also keep in mind that ancilla qubits themselves are a limited resource—in any real quantum computer, only a finite number of qubits will be available. Consequently, quantum oracles using excessive quantities of ancilla qubits are unrealistic in the sense that no quantum computer will be able to execute them. The process of designing a quantum oracle (indeed, designing any quantum circuit) nearly always involves a trade-off between its quantum cost and the number of ancilla qubits it uses.

3.2 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 2.2) 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.

4 Procedure to Calculate the Cost of a Given Encoding

4.1 Computing Cost by Considering Pairs of States

We now consider a systematic procedure for computing the cost, as defined in Section 2.2, 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,

$$Q_i^+ = \delta_i(Q_1, \dots, Q_n, x_1, \dots, x_m)$$

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 2 and the encoding in Figure 2(b). The encoded transition function $Q_1^+ = \delta_1(Q_1, Q_2, x_1, x_2)$ resulting from this encoding, shown in Figure 2(c), 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

$$D_j(\mathrm{S}, \mathrm{S}') \Rightarrow \forall \mathrm{I} \in \mathbf{I}\ A_i(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}', \mathrm{I})) \tag{1}$$

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

$$D_j(\mathrm{I}, \mathrm{I}') \Rightarrow \forall \mathrm{S} \in \mathbf{S}\ A_i(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}')) \tag{2}$$

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.

4.2 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 specified functions. In general, when δ_i is incompletely specified, the question of whether or

	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)

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	$x_1 x_2$
I_1	0 0
I_2	0 1
I_3	1 0

(b)

$Q_1Q_2Q_3 \backslash x_1x_2$	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)

Figure 9: (a) FSM specification with numbers of states and inputs not powers of 2; (b) a possible encoding; (c) resulting form of Q_2^+.

not δ_i depends on a given variable cannot be answered by the simple procedure described above.

Figure 9 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:

$$Q_2^+ = \neg Q_1 \wedge \neg Q_2, \tag{3}$$

$$Q_2^+ = \neg Q_1 \wedge \neg Q_3, \tag{4}$$

$$Q_2^+ = \neg Q_2 \wedge \neg Q_3. \tag{5}$$

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, Eq. 5 implements δ_2 without a dependency on Q_1. Similarly, Eq. 4 and Eq. 3 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 Eq. 5 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[5] 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[6]

$$f(x_1, x_2, \ldots, x_n) = f(x_1, {x_2}', \ldots, x_n) = f({x_1}', {x_2}', \ldots, x_n),$$

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.

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

[5]As in Section 3.1, 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.

[6]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.

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.

5 Design of a Quantum Oracle to Find Optimal Encodings

5.1 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 3.2 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, 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.

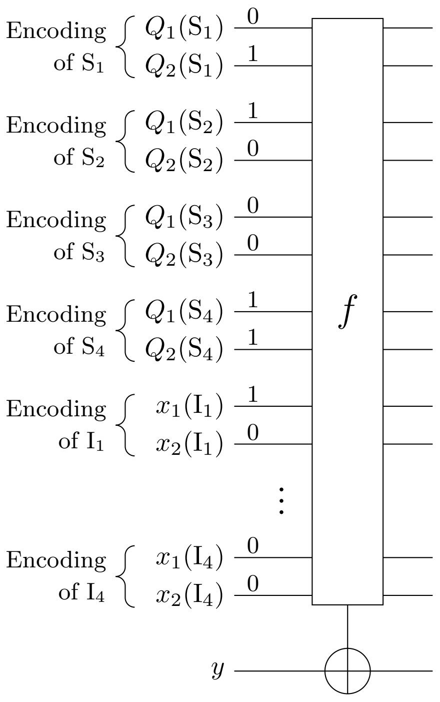
Figure 10 provides an example of an encoding represented as binary data that can be input to a quantum oracle. In this figure as well as others in this section, the notation $Q_i(\mathrm{S}_j)$ denotes the value assigned to Q_i in the encoding of S_j. Similarly, $x_i(\mathrm{I}_j)$ denotes the value 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.

S	Q_1Q_2
S_1	0 1
S_2	1 0
S_3	0 0
S_4	1 1

I	$x_1\,x_2$
I_1	1 0
I_2	1 1
I_3	0 1
I_4	0 0

(a)



(b)

Figure 10: The quantum oracle's representation of an encoding as binary data—(a) state and input encodings for an FSM; (b) their representations as supplied to the oracle.

5.2 Quantum Circuit to Detect Dependencies

We next demonstrate a quantum circuit design that uses the procedure from Section 4.1 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 Eq. 1 and Eq. 2 for each pair of states or input values, respectively. In turn, Eq. 1 and Eq. 2 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 4.1.

Using the binary representation described in Section 5.1, 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 11(a) shows a quantum circuit that evaluates $D_1(\mathrm{S}, \mathrm{S}')$ for any two distinct states S and S′.[7] 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 Feynman (a.k.a. controlled-NOT) 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 $\neg(Q_2(\mathrm{S}) \oplus Q_2(\mathrm{S}')) \wedge \neg(Q_3(\mathrm{S}) \oplus Q_3(\mathrm{S}'))$, which evaluates to 1 if and only if the encodings of S and S′ agree in all state variables except Q_1, exactly the definition of $D_1(\mathrm{S}, \mathrm{S}')$.

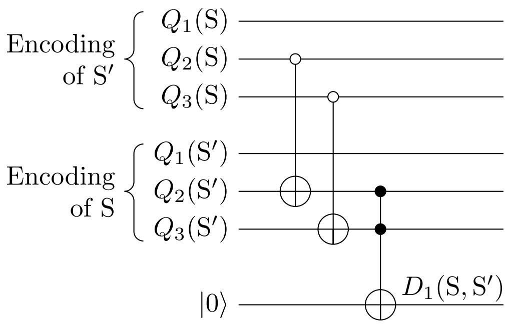
Although we assumed for the purpose of this particular illustration that each state is encoded by three bits, the general circuit structure applies to encodings of any size. Likewise, although this particular circuit evaluates $D_1(\mathrm{S}, \mathrm{S}')$, similar circuits suffice to evaluate $D_j(\mathrm{S}, \mathrm{S}')$ 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 11(b) shows a quantum circuit that evaluates $A_1(\mathrm{S}, \mathrm{S}')$. This circuit is extremely simple, as it simply evaluates $\neg(Q_1(\mathrm{S}) \oplus Q_1(\mathrm{S}'))$, 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.

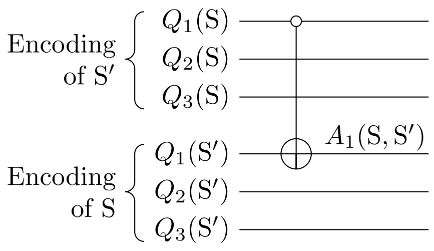
Next, to check the conditions specified by Eq. 1 and Eq. 2, our quantum circuit must be able to evaluate a logical implication. Recalling that the expression $a \Rightarrow b$ is defined as $\neg a \vee b$, we observe that a single Toffoli gate suffices to evaluate a logical implication, as shown in Figure 12.

Finally, we are ready to turn to the design of the full quantum circuit for checking Eq. 1 or Eq. 2. This task is complicated by the fact that checking Eq. 1 or Eq. 2 involves evaluating $A_i(\mathrm{S}, \mathrm{S}')$ simultaneously for

[7] 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 2.2. If S and S′ happen to have the same encoding, the circuit will erroneously output 1. However, later on, in Section 5.5, 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 5.6, 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.



(a)



(b)

Figure 11: (a) Quantum circuit to check $D_1(\mathrm{S},\mathrm{S}')$; (b) circuit to check $A_1(\mathrm{S},\mathrm{S}')$.

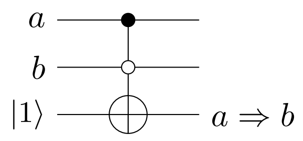


Figure 12: Logical implication evaluated using a Toffoli gate.

many pairs of states. For instance, consider the state machine from Figure 2, and suppose that we wish to design a quantum circuit to evaluate Eq. 2 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 $A_1(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}'))$ holds for all states S. In this case, given $\mathrm{I} = \mathrm{I}_1$ and $\mathrm{I}' = \mathrm{I}_2$, the corresponding pairs $(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}, \mathrm{I}'))$ 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 13 demonstrates a subcircuit that accomplishes this task. The lower portion of Figure 13 then combines this subcircuit with another subcircuit (from Figure 11(b)) to evaluate the full Eq. 2. In other words, the final output of this circuit, on the bottommost qubit, will be 1 if Eq. 2 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 11(a).

In some cases, checking Eq. 1 or 2 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. 1 or 2 using a slightly different design. For example, suppose that we now wish to design a quantum circuit to evaluate Eq. 2 for the same state machine and $i = 1$, $j = 2$ as before, but a different pair of input values $(\mathrm{I}_1, \mathrm{I}_3)$. The pairs of next states corresponding to current states of S_1, S_2, S_3, and S_4 are $(\mathrm{S}_2, \mathrm{S}_2)$, $(\mathrm{S}_4, \mathrm{S}_3)$, $(\mathrm{S}_2, \mathrm{S}_3)$, and $(\mathrm{S}_4, \mathrm{S}_2)$, respectively. Therefore, the quantum circuit must evaluate

$$D_2(\mathrm{I}_1, \mathrm{I}_3) \Rightarrow A_1(\mathrm{S}_4, \mathrm{S}_3) \wedge A_1(\mathrm{S}_2, \mathrm{S}_3) \wedge A_1(\mathrm{S}_4, \mathrm{S}_2). \tag{6}$$

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 $\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4$ must *all* agree in the value of Q_1.[8] We denote this condition by $A_1(\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4)$, a natural generalization of our earlier $A_i(\mathrm{S}, \mathrm{S}')$ notation for pairs of states. Figure 14 illustrates the quantum circuit for evaluating Eq. 2 in this case. The critical difference between Figures 13 and 14 is that the Feynman gates in the upper portion of Figure 14 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 Eq. 6 contains the term $A_1(\mathrm{S}_4, \mathrm{S}_2)$, the circuit in Figure 14 does not contain a corresponding Feynman gate to evaluate this term. Such a gate is unnecessary because of transitivity—it is enough to check both $A_1(\mathrm{S}_2, \mathrm{S}_3)$ and $A_1(\mathrm{S}_3, \mathrm{S}_4)$ since they are together equivalent to $A_1(\mathrm{S}_2, \mathrm{S}_3, \mathrm{S}_4)$, which implies $A_1(\mathrm{S}_4, \mathrm{S}_2)$.

More generally, one can construct similar circuits to evaluate $A_i(\mathrm{S}, \mathrm{S}')$ 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 15(a).

[8]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.

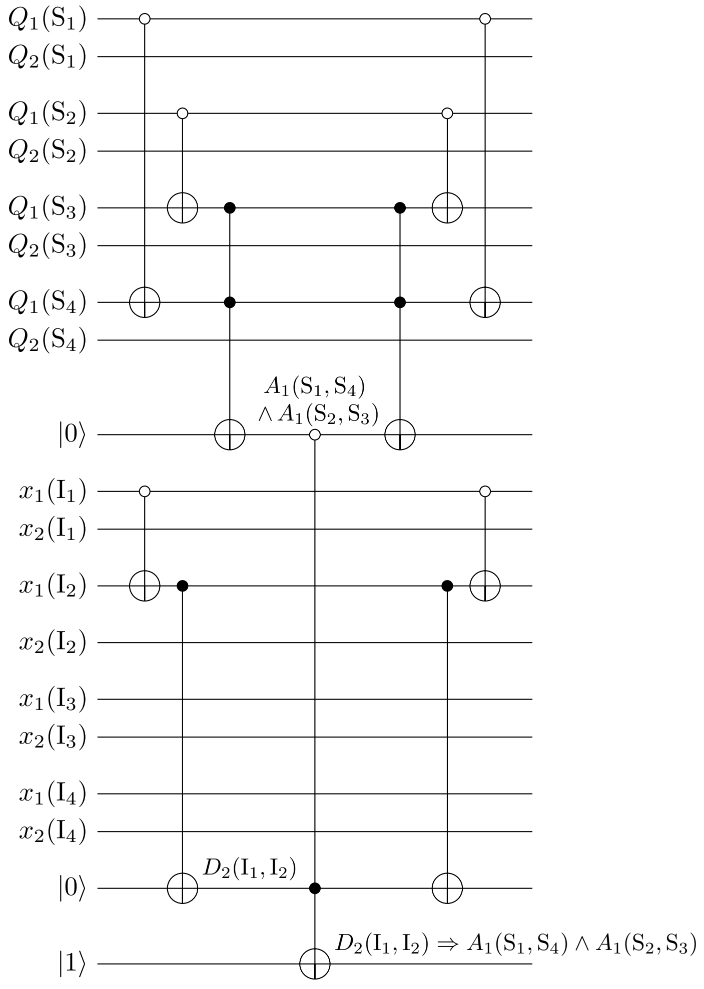


Figure 13: Quantum circuit to evaluate Eq. 2 for input pair $(\mathrm{I}_1, \mathrm{I}_2)$.

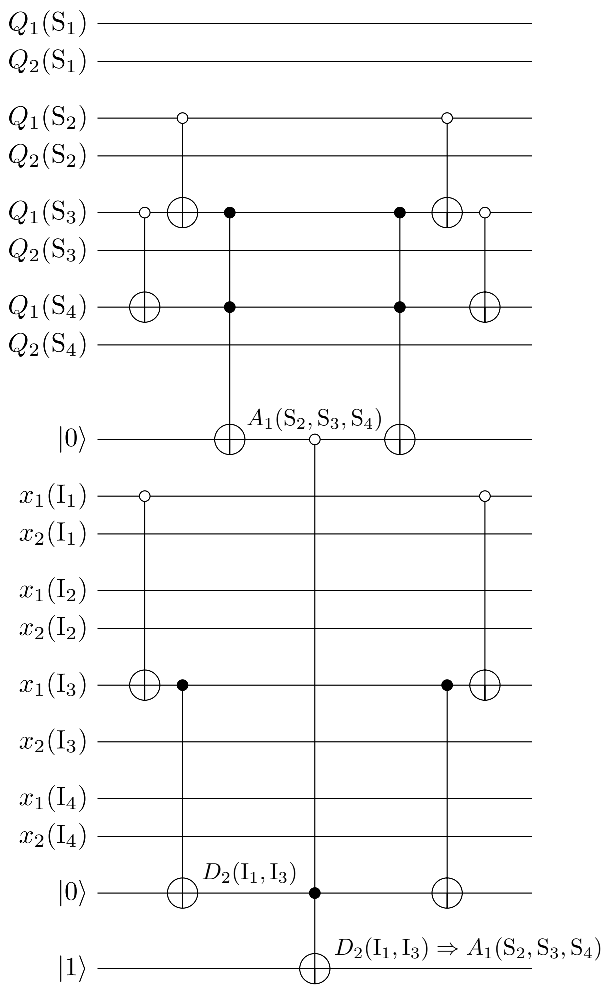


Figure 14: Quantum circuit to evaluate Eq. 2 for input pair $(\mathrm{I}_1, \mathrm{I}_3)$.

Figure 15(b) 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 Eq. 1 or 2 in any case. Specifically, given any FSM with any number of states and input values, the following procedure constructs a quantum circuit to evaluate Eq. 1 for any i, j, and pair of states[9] $(\mathrm{S}, \mathrm{S}')$:

1. List all of the pairs of states appearing on the right-hand side of Eq. 1 when expanded. In other words, create a list containing the pair of states $(\delta(\mathrm{S}, \mathrm{I}), \delta(\mathrm{S}', \mathrm{I}))$ 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 15, 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 Eq. 1, following the general pattern illustrated in Figures 13 and 14.

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 Eq. 1 for a single pair of states while the existence of a dependency is determined by checking Eq. 1 for all possible pairs of states.

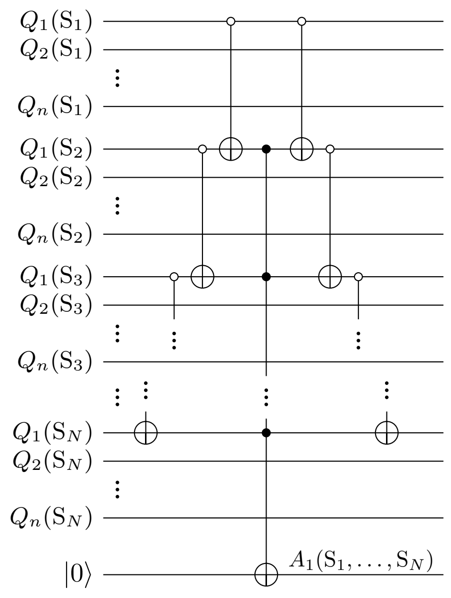
We observe that crucially, the quantum circuit design procedure described in this section requires at compile time only knowledge of the transition function of the FSM being encoded. In particular, the quantum circuit cannot be modified depending on the encoding whose cost is being evaluated, because 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.

5.3 Quantum Circuit to Calculate Total Cost of an Encoding

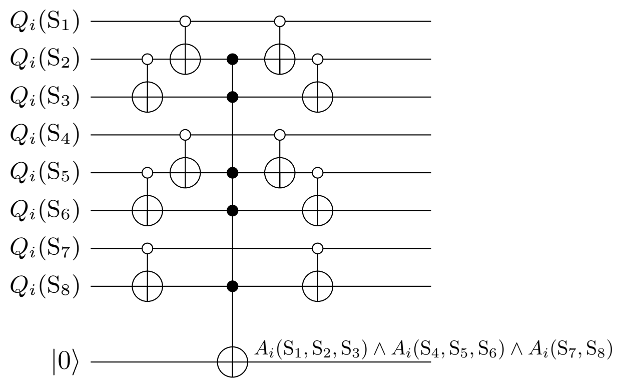
With the ability to generate a quantum circuit to evaluate Eq. 1 or 2 for any i, j, and pair of states or inputs, we now consider the task of designing a quantum circuit to calculate the full cost of a given encoding. Following the procedure from Section 4.1, the quantum circuit determines whether a given Q_i^+ depends on Q_j by checking if Eq. 1 is satisfied for *all* possible pairs of states.[10] Figure 16 shows a circuit structure that accomplishes this task. From now on, we will refer to this circuit as a *dependency checker*.

[9] We describe the procedure here using Eq. 1 with a pair of states, but it functions equally well for Eq. 2 with a pair of input values.

[10] 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.



(a)



(b)

Figure 15: (a) Circuit to evaluate $A_1(\mathrm{S}_1, \ldots, \mathrm{S}_n)$ for an arbitrary number of states; (b) circuit for multiple sets of states.

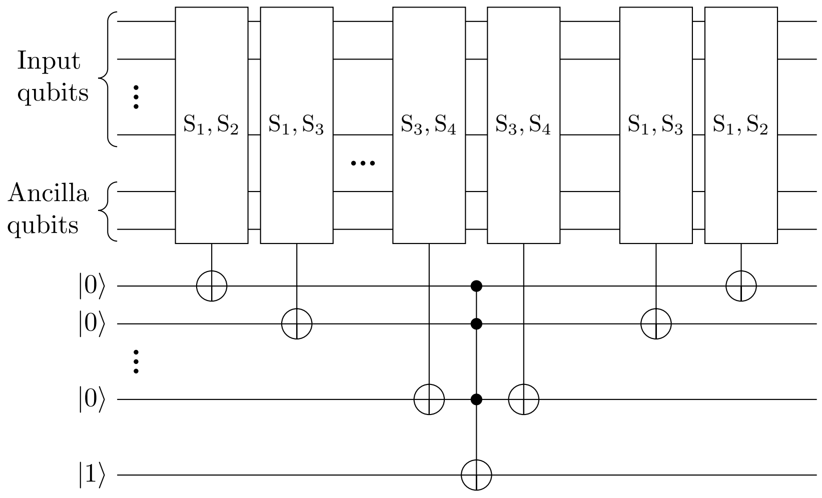


Figure 16: Quantum circuit to check dependency of Q_i^+ on a single state or input variable.

In Figure 16, the labeled "input qubits" represent the entire collection of qubits storing the binary representation of an encoding, as described in Section 5.1. Each block represents a partial dependency checker that checks Eq. 1 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 13 and 14. 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 previously discussed in Section 3.1, 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 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 17(a) 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

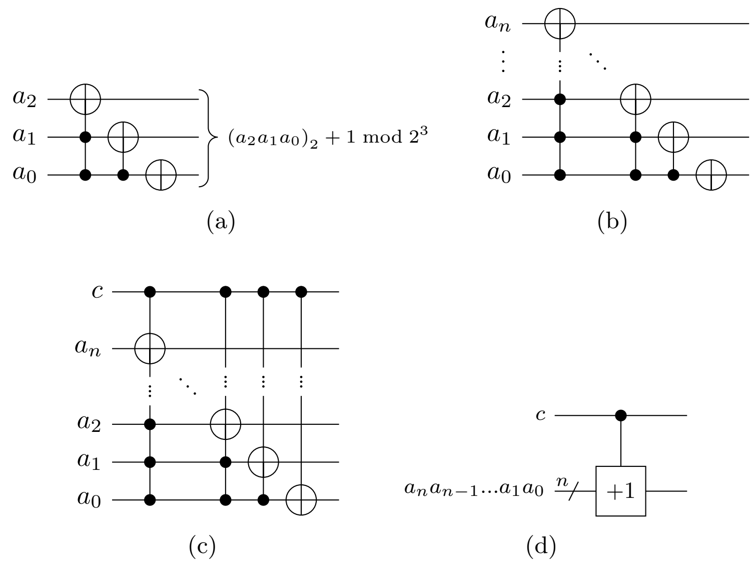


Figure 17: (a) A three-qubit incrementer; (b) general n-qubit incrementer; (c) controlled three-qubit incrementer; (d) schematic symbol for the controlled incrementer.

around to 0 upon increment. Figure 17(b) illustrates how the incrementer is extended to any number of qubits. Figure 17(c) 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 the control qubit c is in the state $|1\rangle$. Figure 17(d) 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 18 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 $2^n - 1$.

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

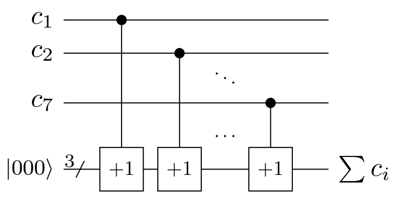


Figure 18: an example of a quantum counter.

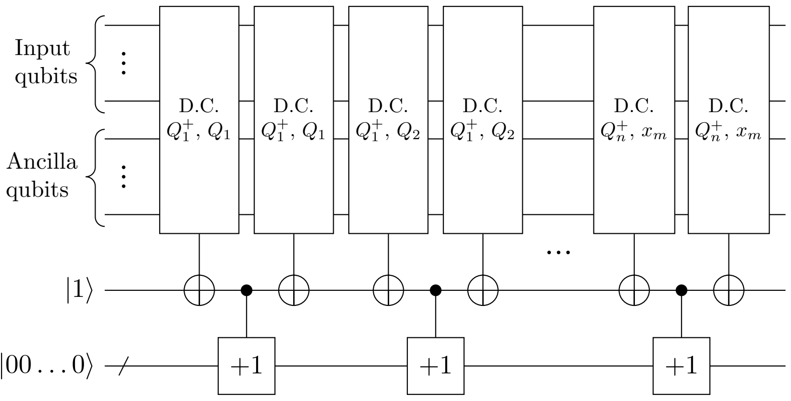


Figure 19: Quantum circuit to calculate total cost using a quantum counter.

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 19 depicts the structure of the resulting circuit.

In Figure 19, 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 19, 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 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 19 is constructed slightly differently from the example shown in Figure 18.

Specifically, the quantum counter in Figure 19 is able to count using only one control qubit, while the one in Figure 18 counts the number of ones present in a whole collection of control qubits. The reason for this difference is that the circuit in Figure 19 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 19, 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.

5.4 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 $(a_n a_{n-1} \dots a_1 a_0)_2$ is less than or equal to a threshold $(r_n r_{n-1} \dots r_1 r_0)_2$:

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 $(a_{n-1} \dots a_0)_2 \leq (r_{n-1} \dots r_0)_2$.
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

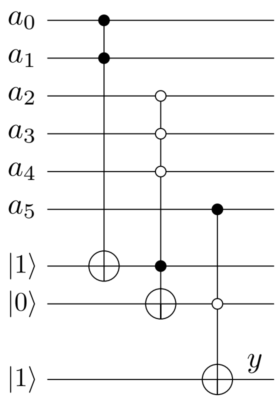


Figure 20: Quantum circuit implementing the expression $y = \neg a_5 \vee (\neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge (\neg a_1 \vee \neg a_0))$.

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 $\neg a_2 \vee \neg a_1$.

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 20, 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 3.2.

5.5 Quantum Circuit to Enforce Bijectivity of Encodings

In Section 4.2, 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 21. 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. Observe that the exact same circuit applies for checking both state and input encodings, and that although Figure 21 assumes four states, the same structure may of course be scaled up for FSMs with any number of states or inputs.

5.6 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 5.3), 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 5.4). 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 5.5). 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 22.

In Figure 22, 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 17.

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 20 would require additional mirror gates (as shown in Figure 23) 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 20, 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 23. 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 21, which become unnecessary when the subcircuits are used as components of the oracle in Figure 22.

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 3.2. The other parts of the oracle are generated depending on the

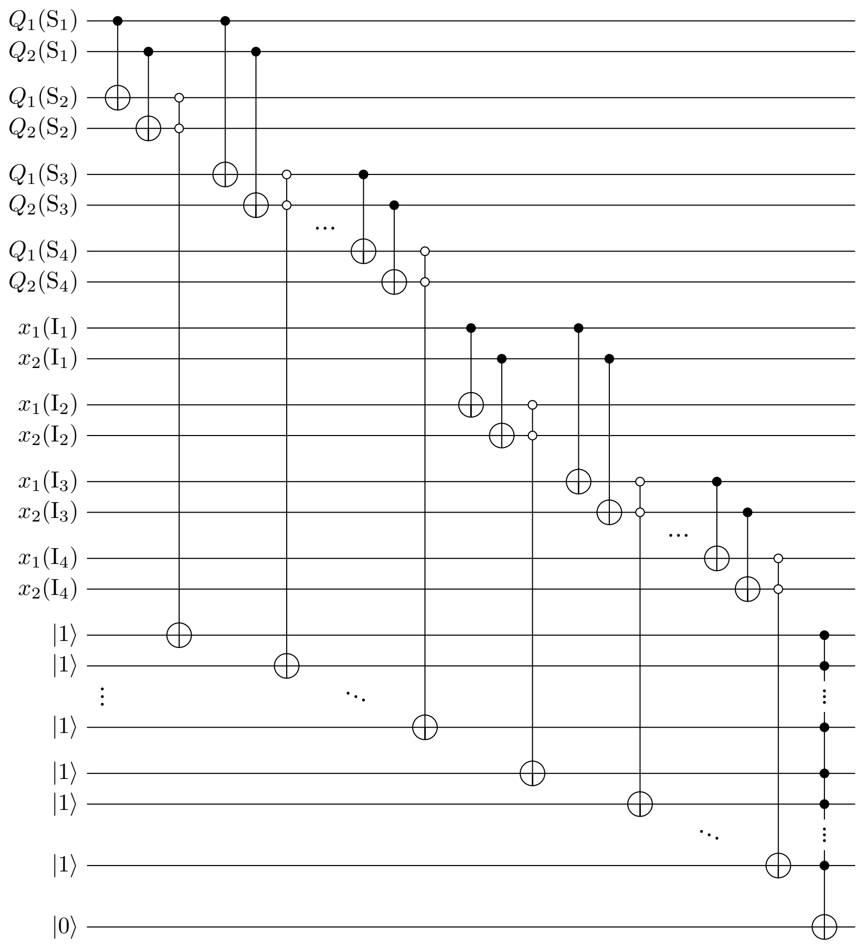


Figure 21: Quantum circuit to verify uniqueness of encoding for each state/input.

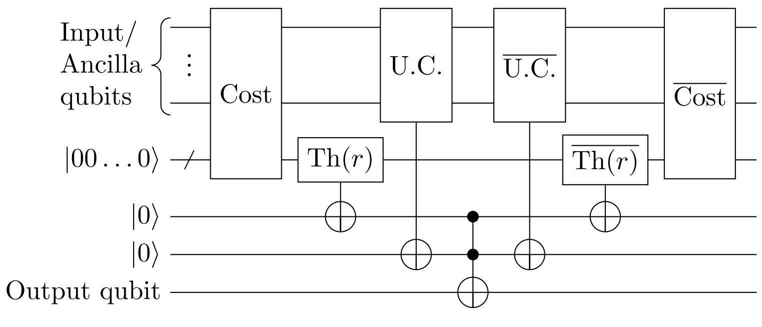


Figure 22: High-level view of the quantum oracle.

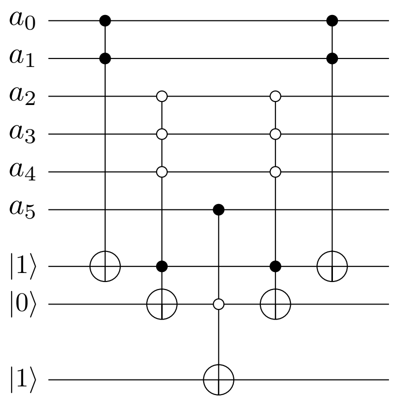


Figure 23: Threshold circuit from Figure 20 with mirror gates. The mirror gates turn out to be unnecessary as explained in the main text.

FSM being encoded, but are independent of the chosen threshold; thus, they remain unchanged throughout the entire search procedure described in Section 3.2. With this oracle,[11] we have met the objective stated at the end of Section 3.2 and therefore, the procedure described in Section 3.2 gives a complete algorithm for finding an exact optimal encoding of an FSM with the help of Grover's algorithm.

6 Run Time Complexity Analysis

6.1 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 Feynman, 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

[11]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 3.2.

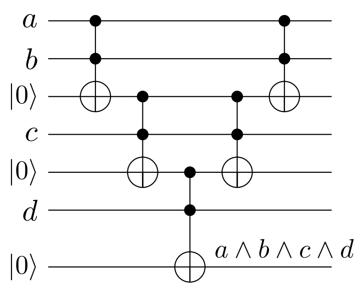


Figure 24: Implementation of a multiple-control Toffoli gate with quantum cost scaling linearly with gate size.

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[12] 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 24, accomplishes this task.

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 integers n_{S} and n_{I}, where n_{S} and n_{I} are respectively the number of state and input variables. Conversely, $n_{\mathrm{S}} = \log_2 N_{\mathrm{S}} = \mathcal{O}(\log N_{\mathrm{S}})$ and $N_{\mathrm{I}} = \log_2 N_{\mathrm{I}} = \mathcal{O}(\log N_{\mathrm{I}})$. We first

[12]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 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.

consider the run time complexity for the dependency checking quantum circuits described in Sections 5.2 and 5.3. Calculation of this complexity is complicated by the fact that the partial dependency checkers described in Section 5.2 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 11, 13, and 14 as a reference. Evaluating $D_j(\mathrm{S}, \mathrm{S}')$ for a pair of states requires $\mathcal{O}(n_\mathrm{S}) = \mathcal{O}(\log N_\mathrm{S})$ time, since $n_\mathrm{S} - 1$ Feynman gates are required together with a multiple-control Toffoli gate of size n_S ($n_\mathrm{S} - 1$ control qubits and one target qubit). By analogous reasoning, evaluating $D_j(\mathrm{I}, \mathrm{I}')$ requires $\mathcal{O}(n_\mathrm{I}) = \mathcal{O}(\log N_\mathrm{I})$ time. Meanwhile, evaluating $A_i(\mathrm{S}, \mathrm{S}')$ is an $\mathcal{O}(1)$ operation, since only a single Feynman gate is needed. Observe that the upper portion of every partial dependency checker (as in Figures 13 and 14) must evaluate $A_i(\mathrm{S}, \mathrm{S}')$ for up to $N_\mathrm{S} - 1$ state pairs, with this maximum being obtained when all state pairs are overlapping, as in $(\mathrm{S}_1, \mathrm{S}_2)$, $(\mathrm{S}_2, \mathrm{S}_3)$, $(\mathrm{S}_3, \mathrm{S}_4)$, etc. The lower portion of each partial dependency checker always evaluates $D_j(\mathrm{S}, \mathrm{S}')$ or $D_j(\mathrm{I}, \mathrm{I}')$ for only one pair of inputs/states. The total complexity of a partial dependency checker is therefore $(N_\mathrm{S} - 1) \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(\log N_\mathrm{S}) = \mathcal{O}(N_\mathrm{S} + \log N_\mathrm{S}) = \mathcal{O}(N_\mathrm{S})$, for a partial dependency checker that evaluates $D_j(\mathrm{S}, \mathrm{S}')$ for a pair of states and checks Eq. 1, or $(N_\mathrm{S} - 1) \cdot \mathcal{O}(1) + 1 \cdot \mathcal{O}(\log N_\mathrm{S}) = \mathcal{O}(N_\mathrm{S} + \log N_\mathrm{I})$, for a partial dependency checker that evaluates $D_j(\mathrm{I}, \mathrm{I}')$ for a pair of inputs and checks Eq. 2.

From Figure 16 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 $N_\mathrm{S}(N_\mathrm{S} - 1)/2$ (for a state variable) or $N_\mathrm{I}(N_\mathrm{I} - 1)/2$ (for an input variable) partial dependency checkers. Combining with the previously derived complexity for a single partial dependency checker gives $\mathcal{O}({N_\mathrm{S}}^3)$ (for a state variable) or $\mathcal{O}({N_\mathrm{I}}^2(N_\mathrm{S} + \log N_\mathrm{I}))$ (for an input variable). The central Toffoli gate in Figure 16 has either $N_\mathrm{S}(N_\mathrm{S} - 1)/2$ or $N_\mathrm{I}(N_\mathrm{I} - 1)/2$ control qubits, so its time complexity is dominated by that of the partial dependency checkers.

In Figure 19, we can see that there are ${n_\mathrm{S}}^2 = (\log N_\mathrm{S})^2$ checks for dependency of Q_i^+ (for any i) on a state variable, and $n_\mathrm{S} n_\mathrm{I} = \log N_\mathrm{S} \log N_\mathrm{I}$ checks for dependency on an input variable. Therefore, the total complexity of all the dependency checkers in the circuit from Figure 19 is

$$\begin{aligned}(\log N_\mathrm{S})^2 \cdot \mathcal{O}({N_\mathrm{S}}^3) + \log N_\mathrm{S} \log N_\mathrm{I} \cdot \mathcal{O}({N_\mathrm{I}}^2(N_\mathrm{S} + \log N_\mathrm{I}))\\ = \mathcal{O}({N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{I}}^2(N_\mathrm{S} + \log N_\mathrm{I})(\log N_\mathrm{S} \log N_\mathrm{I})). \qquad (7)\end{aligned}$$

The complexity of the incrementer circuits in Figure 19 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 22—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 21, the uniqueness-checking circuit performs a comparison for every possible pair of states, of which there are $N_{\mathrm{S}}(N_{\mathrm{S}}-1)/2$, and every possible pair of input values, of which there are $N_{\mathrm{I}}(N_{\mathrm{I}}-1)/2$. Furthermore, a comparison of a pair of states has complexity $\mathcal{O}(n_{\mathrm{S}}) = \mathcal{O}(\log N_{\mathrm{S}})$, since each individual state variable assignment must be compared between the two states, and a comparison of a pair of inputs similarly has complexity $\mathcal{O}(n_{\mathrm{I}}) = \mathcal{O}(\log N_{\mathrm{I}})$. The final Toffoli gate in Figure 21 has $N_{\mathrm{S}}(N_{\mathrm{S}}-1)/2 + N_{\mathrm{I}}(N_{\mathrm{I}}-1)/2$ control qubits, giving the entire uniqueness-checking circuit a complexity of

$$\begin{aligned}&\frac{N_{\mathrm{S}}(N_{\mathrm{S}}-1)}{2}\cdot\mathcal{O}(\log N_{\mathrm{S}}) + \frac{N_{\mathrm{I}}(N_{\mathrm{I}}-1)}{2}\cdot\mathcal{O}(\log N_{\mathrm{I}})\\ &\qquad + \mathcal{O}\left(\frac{N_{\mathrm{S}}(N_{\mathrm{S}}-1)}{2} + \frac{N_{\mathrm{I}}(N_{\mathrm{I}}-1)}{2}\right)\\ &= \mathcal{O}(N_{\mathrm{S}}^2\log N_{\mathrm{S}} + N_{\mathrm{I}}^2\log N_{\mathrm{I}}).\end{aligned}$$

Comparison with Eq. 7 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 Eq. 7.

As we recall from Section 3.2, 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 3.2 in order to find the minimum possible cost.

We discussed in Section 3.1 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 $\mathcal{O}(\sqrt{M})$ 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 5.1 and shown in Figure 10(b), our proposed oracle requires $N_{\mathrm{S}}n_{\mathrm{S}} + N_{\mathrm{I}}n_{\mathrm{I}} = \mathcal{O}(N_{\mathrm{S}}\log N_{\mathrm{S}} + N_{\mathrm{I}} + \log N_{\mathrm{I}})$ input qubits, corresponding to a search space of size $\mathcal{O}(2^{N_{\mathrm{S}}\log N_{\mathrm{S}} + N_{\mathrm{I}} + \log N_{\mathrm{I}}}) = \mathcal{O}({N_{\mathrm{S}}}^{N_{\mathrm{S}}}{N_{\mathrm{I}}}^{N_{\mathrm{I}}})$. Hence, a single execution of Grover's algorithm requires $\mathcal{O}(\sqrt{{N_{\mathrm{S}}}^{N_{\mathrm{S}}}{N_{\mathrm{I}}}^{N_{\mathrm{I}}}})$ 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

$$\mathcal{O}(\sqrt{{N_{\mathrm{S}}}^{N_{\mathrm{S}}}{N_{\mathrm{I}}}^{N_{\mathrm{I}}}}({N_{\mathrm{S}}}^3(\log N_{\mathrm{S}})^2 + {N_{\mathrm{I}}}^2(N_{\mathrm{S}} + \log N_{\mathrm{I}})(\log N_{\mathrm{S}}\log N_{\mathrm{I}}))). \quad (8)$$

Eq. 8 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

Eq. 8 simplifies to the much more manageable

$$\begin{aligned}&\mathcal{O}(\sqrt{{N_\mathrm{S}}^{N_\mathrm{S}}{N_\mathrm{I}}^{N_\mathrm{I}}}({N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{I}}^2(N_\mathrm{S} + \log N_\mathrm{I})(\log N_\mathrm{S} \log N_\mathrm{I})))\\ &\quad = \mathcal{O}({N_\mathrm{S}}^{N_\mathrm{S}}({N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{S}}^2(N_\mathrm{S} + \log N_\mathrm{S})(\log N_\mathrm{S} \log N_\mathrm{S})))\\ &\quad = \mathcal{O}({N_\mathrm{S}}^{N_\mathrm{S}}({N_\mathrm{S}}^3(\log N_\mathrm{S})^2))\\ &\quad = \mathcal{O}({N_\mathrm{S}}^{N_\mathrm{S}+3}(\log N_\mathrm{S})^2). \end{aligned} \tag{9}$$

6.2 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. 1 and 2 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 Eq. 1 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 Eq. 1 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 Eq. 2 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 $\mathcal{O}({N_\mathrm{S}}^2 N_\mathrm{I}(\log N_\mathrm{S})^2 + N_\mathrm{S}{N_\mathrm{I}}^2(\log N_\mathrm{S})(\log N_\mathrm{I}))$ 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

$$\mathcal{O}(N_\mathrm{S}!N_\mathrm{I}!({N_\mathrm{S}}^2 N_\mathrm{I}(\log N_\mathrm{S})^2 + N_\mathrm{S}{N_\mathrm{I}}^2(\log N_\mathrm{S})(\log N_\mathrm{I}))). \tag{10}$$

Just as for the quantum algorithm, we can simplify this expression if we assume that $N_\mathrm{S} = N_\mathrm{I}$; in this case, Eq. 10 reduces to

$$\begin{aligned}&\mathcal{O}(N_\mathrm{S}!N_\mathrm{S}!({N_\mathrm{S}}^3(\log N_\mathrm{S})^2 + {N_\mathrm{S}}^3(\log N_\mathrm{S})(\log N_\mathrm{S})))\\ &\quad = \mathcal{O}(N_\mathrm{S}!^2{N_\mathrm{S}}^3(\log N_\mathrm{S})^2). \end{aligned} \tag{11}$$

Comparing Eqs. 9 and 11, 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 algorithms, if $N_\mathrm{S} = N_\mathrm{I}$, by looking only at the terms ${N_\mathrm{S}}^{N_\mathrm{S}}$ (for the quantum algorithm) and $N_\mathrm{S}!^2$ (for the classical algorithm). Table 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

Table 1: Comparison of relative complexities of quantum and classical algorithms, assuming $N_\mathrm{S} = N_\mathrm{I}$.

N_S	${N_\mathrm{S}}^{N_\mathrm{S}}$	$N_\mathrm{S}!^2$	$\dfrac{N_\mathrm{S}!^2}{{N_\mathrm{S}}^{N_\mathrm{S}}}$
4	256	576	2.25
8	$1.68 \cdot 10^{7}$	$1.63 \cdot 10^{9}$	96.9
16	$1.84 \cdot 10^{19}$	$4.38 \cdot 10^{26}$	$2.37 \cdot 10^{7}$
32	$1.46 \cdot 10^{48}$	$6.92 \cdot 10^{70}$	$4.74 \cdot 10^{22}$

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 $CN_\mathrm{S}!^2/{N_\mathrm{S}}^{N_\mathrm{S}}$, 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 1, we can see that for $N_\mathrm{S} = 16$, 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 $N_\mathrm{S} = 32$, 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 10(b), 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 21. 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 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, [12] is the only other published method that finds exact minimum solutions; however, [12] only solves the problem of state, and not input, encoding. Additionally, the method presented in [12] 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 3.2. 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 Section 3.2. 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.

References

[1] 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.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pp. 212–219, May 1996.

[3] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Phys. Rev. Lett.*, vol. 79, pp. 325–328, Jul 1997.

[4] L. K. Grover, "A framework for fast quantum mechanical algorithms," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), pp. 53–62, ACM, 1998.

[5] G. Wendin, "Quantum information processing with superconducting circuits: a review," *Reports on Progress in Physics*, vol. 80, no. 10, p. 106001, 2017.

[6] J. Hartmanis, "On the state assignment problem for sequential machines. I," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 157–165, June 1961.

[7] R. E. Stearns and J. Hartmanis, "On the state assignment problem for sequential machines. II," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 593–603, Dec. 1961.

[8] 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.

[9] G. D. Hachtel, M. Hermida, A. Pardo, M. Poncino, and F. Somenzi, "Re-encoding sequential circuits to reduce power dissipation," in *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, (Los Alamitos, CA, USA), pp. 70–73, IEEE Computer Society Press, 1994.

[10] G. D. Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, pp. 269–285, July 1985.

[11] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 905–924, Sept. 1990.

[12] S. Devadas and A. R. Newton, "Exact algorithms for output encoding, state assignment and four-level boolean minimization," in *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, vol. 1, pp. 387–396, IEEE, 1990.

[13] E. F. Moore, "Gedanken Experiments on Sequential Machines," in *Automata Studies*, pp. 129–153, Princeton U., 1956.

[14] J. Hartmanis, “Loop-free structure of sequential machines,” *Information and Control*, vol. 5, no. 1, pp. 25–43, 1962.

[15] G. H. Mealy, “A method for synthesizing sequential circuits,” *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[16] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information.* Cambridge University Press, 10th anniversary ed., 2010.

[17] 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.

[18] G. Brassard, P. Høyer, and A. Tapp, “Quantum counting,” in *Automata, Languages and Programming*, pp. 820–831, Springer Berlin Heidelberg, 1998.

[19] D. Maslov and G. Dueck, “Improved quantum cost for n-bit toffoli gates,” *Electronics Letters*, vol. 39, pp. 1790–1791, December 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.