Symbolic Execution of Toffoli-Hadamard Quantum Circuits

Jacques Carette
McMaster University
Hamilton, ON, Canada

Gerardo Ortiz
Indiana University
Bloomington, IN, USA

Amr Sabry
Indiana University
Bloomington, IN, USA

ACM Reference Format:
Jacques Carette, Gerardo Ortiz, and Amr Sabry. 2022. Symbolic Execution of Toffoli-Hadamard Quantum Circuits. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The general state of a quantum bit (qubit) is mathematically modeled using an equation parameterized by two angles θ and ϕ as follows:

$$\cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle$$

The description models the fact that the qubit is in a superposition of false $|0\rangle$ and true $|1\rangle$. The angle θ determines the relative amplitudes of false and true and the angle ϕ determines the relative phase between them. A particular case when $\theta = \pi/2$ and $\phi = 0$ is ubiquitous in quantum algorithms. In those cases, the general representation reduces to:

$$1/\sqrt{2}(|0\rangle + |1\rangle)$$

which represents a qubit in an equal superposition of false and true.

The reason this particular case is distinguished is because a rather common template for quantum algorithms is to start with qubits initialized to $|0\rangle$ and immediately apply a Hadamard H transformation whose action is $|0\rangle \mapsto 1/\sqrt{2}(|0\rangle + |1\rangle)$. This superposition is then further manipulated depending on the algorithm in question.

Our observation is that a qubit in the special superposition $1/\sqrt{2}(|0\rangle + |1\rangle)$ is, computationally speaking, indistinguishable from a symbolic boolean variable with an unknown value in the same sense used in symbolic evaluation of classical programs. First, the superposition is not observable. The only way to observe the qubit is via a measurement which collapses the state to be either false or true with equal probability. Second, and more significantly, this remarkably simple observation is quite robust even in the presence of multiple, possibly entangled, qubits.

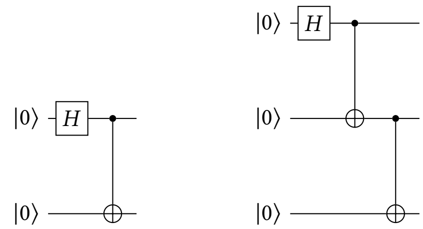


Figure 1. Bell and GHZ States

To see this, consider the conventional quantum circuits for creating the maximally entangled Bell and GHZ states in Fig. 1. On the left, the circuit generates the Bell state $(1/\sqrt{2})(|00\rangle + |11\rangle)$ as follows. First the state evolves from $|00\rangle$ to $(1/\sqrt{2})(|00\rangle+|10\rangle)$. Then we apply the CX-gate whose action is to negate the second qubit when the first one is true. By using the symbol x for $H\,|0\rangle$, the input to the CX-gate is $|x0\rangle$. A simple case analysis shows that the action of CX-gate on inputs $|xy\rangle$ is $|x(x \oplus y)\rangle$ where $\oplus$ is the exclusive-or boolean operation. In other words, the CX-gate transforms $|x0\rangle$ to $|xx\rangle$. Since any measurement of the Bell state must produce either 00 or 11, by producing a symbolic state that shares the same name in two positions, we have accurately represented the entangled Bell state. Similarly, for the GHZ circuit on the right of Fig. 1, the state after the Hadamard gate is $|x00\rangle$ which evolves to $|xx0\rangle$ and then to $|xxx\rangle$ again accurately capturing the entanglement correlations.

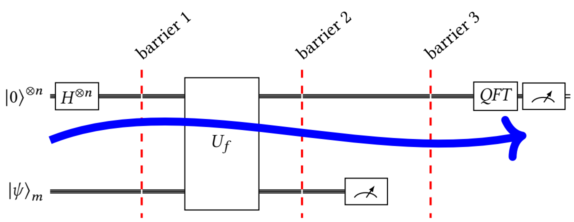
The introduction of symbolic variables opens a host of new exciting possibilities. Consider again the Bell circuit in Fig. 2 with an arbitrary initial value for the second qubit. The right subfigure Fig. 2a removes the explicit use of $H\,|0\rangle$ and replaces the top qubit with another symbolic variable. Because quantum circuits are reversible, we can, at this point, "partially evaluate" the circuit under various regimes. For example, we can set $y_1 = 0$ and $y_2 = 1$ and ask about values of x_1 and x_2 that would be consistent with this setting. We can calculate backwards from $|x_2 1\rangle$ as follows. The state evolves to $|x_2(1 \oplus x_2)\rangle$ which can be reconciled with the initial conditions yielding the constraints $x_1 = x_2$ and $1 \oplus x_2 = 0$ whose solutions are $x_1 = x_2 = 1$.

Conference'17, July 2017, Washington, DC, USA

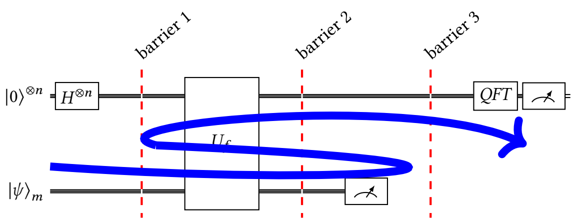
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(a) Bell circuit **(b)** Symbolic variant

Figure 2. A conventional quantum circuit for generating a Bell state (a); its classical symbolic variant.



(a) Conventional Flow



(b) Retrodictive Flow

Figure 3. Template quantum circuit

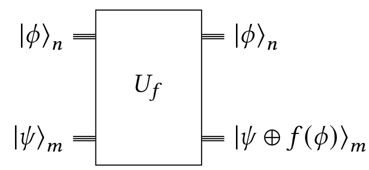


Figure 4. Circuit Abstraction

2 Circuits and Boolean Functions

Special Circuits. We analyze a particular class of circuits, namely those that match the template in Fig. 3 (including Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor's algorithms [2, 6, 7, 9, 13–15]). The U_f block, defined as

$$U_f(|x\rangle\,|y\rangle) = |x\rangle\,|f(x) \oplus y\rangle\,, \tag{1}$$

ends up being completely classical, albeit performing mixed mode execution of the circuit. More precisely, here it means that in all these algorithms, the top collection of wires (which we will call the computational register) is prepared in a uniform superposition which can be represented using symbolic variables. The measurement of the bottom collection of wires (which we call the ancilla register) after barrier 2 provides partial information about the future which is, together with the initial conditions of the ancilla register, sufficient to symbolically execute the circuit. In each case, instead of the conventional execution flow depicted in Fig. 3(a), we find a possible measurement outcome w at barrier (2) and perform a symbolic retrodictive execution with a state $|xw\rangle$ going backwards to collect the constraints on x that enable us to solve the problem in question.

In other words, it suffices to look at circuits that match the template in Fig. 4.

Algebraic Normal Form (ANF).. Furthermore, the circuits we are interested in can all be expressed in terms of *generalized Toffoli gates* with n control qubits: $a_n, \cdots, a_0$ and one target qubit c is $c \oplus \bigwedge_i a_i$, the exclusive-or of the target c with the conjunction of all the control qubits. In fact, we generalize this further, so that we can control either on a qubit or its negation, by using pairs of control qubit and a boolean. In other words, our gates are $(a_n, b_n), \cdots, (a_0, b_0)$ and one target qubit c is $c \oplus \bigwedge_i (a_i == b_i)$, the exclusive-or of the target c with the conjunction of the result of testing each qubit against its corresponding target boolean. Note that $(a_i == 1)$ can be expressed as just a_i, and $(a_i == 0)$ can be expressed as $1 \oplus a_i$. Such generalized Toffolie gates with n control qubits are called $c^n x$ gates. The special cases for $n = 0$ is the *not* gate 'x', for $n = 1$ is the *controlled not* gate 'cx', and for $n = 2$ is the classical *Toffoli* gate 'ccx'.

The *algebraic normal form* (also called ring sum normal form, Zhegalkin normal form or Reed-Muller expansion) of boolean functions $\mathbb{B}^n \to \mathbb{B}$ is as the exclusive-or ($\oplus$) of ands ($\wedge$) of 0 or more of the inputs x_i. Note that the and of 0 inputs is 1. It is then easy to see that generalized Toffoli gates (without the extra boolean) are already in algebraic normal form. Furthermore, circuits that only use x and cx-gates never generate any conjunctions and hence lead to formulae that are efficiently solvable classically [16, 19].

3 Symbolic Evaluation of Circuits

As the b_i are circuit constants,

For quantum circuits over H, CCX restrict to $|0\rangle$ inputs; look at state after the initial group of H gates

```
classical wires treated as known
superpositions |0> + |1> treated as symbols x
run circuit with known values and symbol for superpositions
```

4 Design and Implementation

Our exposition of the design and implementation of our system will follow Parnas' advice on *faking it* [?]: a reconstruction of the requirements as we should have had them if we'd been all-knowing, and a design that fits those requirements. The version history in our github repository can be inspected for anyone who wants to see our actual path.

As we experimented with the idea of partial evaluation and symbolic execution of circuits, we ended up writing a lot of variants of essentially the same code, but with minor differences in representation. From these early experiments, we could see the major variation points:

- representation of *boolean values* and *boolean functions*,
- representation of ANF,
- representation of circuits.

We also wanted to write out circuits only once, and have them be valid across these representation changes.

A number of our examples involve arbitrary boolean functions (given as black boxes at "compile time") for which we want to, offline, generate an equivalent reversible circuit. In other words for $f : \mathbb{B}^n \to \mathbb{B}$, we wish to generate the circuit for $g : \mathbb{B}^{n+1} \to \mathbb{B}^{n+1}$ such that for $\overline{x} : \mathbb{B}^n$, then $g(\overline{x}, y) = (\overline{x}, y \oplus f(\overline{x}))$.

This leads us to the requirements our code must fulfill.

4.1 Requirements

We need to be able to deal with the following variabilities:

1. multiple representations of *boolean values*,
2. multiple representations of *boolean formulas*,
3. different evaluation means (directly and symbolically),

It must also be possible to implement the following:

4. a reusable representation of circuits composed of generalized Toffoli gates,
5. a reusable representation of the inputs, outputs and ancillas associated to a circuit,
6. a *synthesis* algorithm for circuits implementing a certain boolean function,
7. a reusable library of circuits (such as Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor).

From those, we can make a set of design choices that drive the eventual solution.

We eventually want some non-functional characteristics to hold:

8. evaluation of reasonably-sized circuits should be relatively efficient.

4.2 Design

To meet the first requirement, we use *finally tagless* [?] to encode a *language of values*:

```
class (Show v, Enum v) => Value v where
zero :: v
one  :: v
snot :: v -> v
sand :: v -> v -> v
sxor :: v -> v -> v

-- has a default implementation
snand :: [v] -> v -- n-ary and
snand = foldr sand one
```

which is then implemented 4 times, once for `Bool` and then multiple times for different symbolic variations. As a side-effect, this gives us requirement 3 "for free" if we can write a sufficiently polymorphic evaluator (which we will present below).

Unlike for value representation which can be computed from context, we want to explicitly choose formula representation (requirement 2) ourselves. Thus we use an explicit record instead of an implicit dictionary:

```
data FormulaRepr f r = FR
  { fromVar  :: r -> f
  , fromVars :: Int -> r -> [ f ]
  }
```

The main methods are about *variable representation* `r` and how to insert them into the current *formula representation* `f`, singly or n at once.

A Generalized Toffoli gates can be represented by a list of representation of value accessors `br` (short for boolean representation) along with a list of *controls* that tell us whether to use the bit directly or negated, along with which value will potentially be flipped. The implementation of very common gates (negation and controlled not) are also shown.

```
data GToffoli br = GToffoli [Bool] [br] br

xop :: br -> GToffoli br
xop = GToffoli [] []

cx :: br -> br -> GToffoli br
cx a = GToffoli [True] [a]
```

The core of a circuit (requirement 4) is then implemented as a sequence of these (where `Seq` is from Data.Sequence).

```
type OP br = Seq (GToffoli br)
```

Mainly for efficiency reasons, we model circuits as manipulating *locations holding values* rather than directly acting on values. We use `STRefs` (aliased to `Var`) for that purpose. Putting this together with the model of circuits of **??**, we get

```
data Circuit s v = Circuit
  { op          :: OP (Var s v)
  , xs          :: [Var s v]
  , ancillaIns  :: [Var s v]
  , ancillaOuts :: [Var s v]
  , ancillaVals :: [v]
  }
```

which lets us achieve requirement 5.

For requirement 6, we implement a straightforward version of the algorithm of [?]. Our implementation is *language agnostic*, in other words it works via the `Value` interface, so that the resulting circuits are all of type `OP br` for a free representation `br`. As circuit synthesis is only done for generating examples, we are not worried about its efficiency.

The arithmetic circuit generators are also based on classical algorithms, and are not optimized in any way, neither for running time nor for gate count. Neither are the code for the classical quantum algorithms. They are, however, representation polymorphic.

Above, we said we had 3 different symbolic evaluators. These were not driven by having different levels of *precision* but rather by requirement 8, efficiency. Our first evaluator (`FormAsList`) uses xor-lists of and-lists of literals (as strings, i.e. `"x0"`, `"x1"`, ... in lexicographical order of the wires). ANF is then easy: and-lists are sorted, and duplicates removed. Xor-lists are sorted, grouped, even length lists are removed, and then made unique. This is whoefully inefficient, and was the clear bottleneck in our profiles.

A less naïve approach uses a set of bits for representing literals, an `IntSet` for and-lists, and a normalized multiset for xor maps. We found more efficient to use a multiset for intermediate computations with xor maps which is normalized at the end instead of trying to track even/odd number of occurences. Only computing Cartesian products in this representation requires some thought for finding a reasonably efficient algorithm.

While significantly faster, this was still not sufficiently efficient. Our final representation uses Natural numbers as and-maps where the encoding of literals is now positional, and xor maps are again multisets of these "bitmaps".

As a last optimization, our circuits have a very particular property: the control wires are not written to, so that they are all literals. This can be used to further optimize the evaluation of single gates.

4.3 Implementation

The final code consists of 18 modules that implement various services, see Fig. 5 for a full listing. It consists of only 1449 lines of Haskell code, of which 646 lines are blank, import or comments, module declaration, so that 809 are "code". Testing and printing utilities are not counted in the above.

The code that occupies the most volume is that for running the examples, as each circuit needs its own setup for the input and ancilla wires. Next is the implementation of symbolic representations of formulas in ANF. This is largely because there are a lot of pieces that need to be defined, including many instances; the algorithmic aspect rarely span more than 15 lines in total. The code for generating arithmetic circuits is voluminous as well as largely computational, but is a re-implementation of known material, as is the synthesis code.

A few comments on further implementation details. Sharp readers might have noticed `snand` as defined in class `Value` instead of as a polymorphic function outside the class; we do this to enable its implementation to be overridden. Lastly, `GToffoli`'s implementation relies on an unexpressed invariant: that its two lists are of equal length. We really ought to refactor the code to use a single list of tuples, but this is a pervasive change that would not bring much benefit as we use combinators to build circuits, and these already maintain that invariant. Similarly for `Circuit`: the lists `ancillaIns`, `ancillaOut` and `ancillaVals` should all be of the same length. That invariant is not checked in our code.

5 Evaluation

Algorithms.

We implement six well-known quantum algorithms: Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor. We highlight the salient results for the first five algorithms, and then discuss the most interesting case of Shor's algorithm in detail.

De-Quantization. We abbreviate the set $\{0, 1, \ldots, (n-1)\}$ as $[\mathbf{n}]$. In the Deutsch-Jozsa problem, we are given a function $[2^{\mathbf{n}}] \to [2]$ that is promised to be constant or balanced and we need to distinguish the two cases. The quantum circuit Fig. 6 shows the algorithm for the case $n = 1$. Instead of the conventional execution, we perform a retrodictive execution of the U_f block with an ancilla measurement 0, i.e., with the state $|x_{n-1} \cdots x_1 x_0 0\rangle$. The result of the execution is a symbolic formula r that determines the conditions under which $f(x_{n-1}, \cdots, x_0) = 0$. When the function is constant, the results are $0 = 0$ (always) or $1 = 0$ (never). When the function is balanced, we get a formula that mentions the relevant variables. For example, here are the results of three executions for balanced functions $[2^6] \to [2]$:

- $x_0 = 0$,
- $x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 = 0$, and
- $1 \oplus x_3x_5 \oplus x_2x_4 \oplus x_1x_5 \oplus x_0x_3 \oplus x_0x_2 \oplus x_3x_4x_5 \oplus x_2x_3x_5 \oplus x_1x_3x_5 \oplus x_0x_3x_5 \oplus x_0x_1x_4 \oplus x_0x_1x_2 \oplus x_2x_3x_4x_5 \oplus x_1x_3x_4x_5 \oplus x_1x_2x_4x_5 \oplus x_1x_2x_3x_5 \oplus x_0x_3x_4x_5 \oplus x_0x_2x_4x_5 \oplus x_0x_2x_3x_5 \oplus x_0x_1x_4x_5 \oplus x_0x_1x_3x_5 \oplus x_0x_1x_3x_4 \oplus x_0x_1x_2x_4 \oplus x_0x_1x_2x_4x_5 \oplus x_0x_1x_2x_3x_5 \oplus x_0x_1x_2x_3x_4 = 0$.

In the first case, the function is balanced because it produces 0 exactly when $x_0 = 0$ which happens half of the time in all possible inputs; in the second case the output of the function is the exclusive-or of all the input variables which is another easy instance of a balanced function. The last case is a cryptographically strong balanced function whose output pattern is balanced but, by design, difficult to discern [4].

An important insight is that we actually do not care about the exact formula. Indeed, since we are promised that the function is either constant or balanced, then any formula that refers to at least one variable must indicate a balanced

Module	Service
`Value`	representation of a *language of values* (as a typeclass) and some constructors
`FormulaRepr`	abstract representation of formulas, as a mapping from abstract variables to abstract formulas
`Variable`	variables as locations holding values and their constructors
`ModularArith`	modular arithmetic utilities useful in implementing certain algorithms, like Shor's
`BoolUtils`	function to interpret a list of booleans as an Integer
`GToffoli`	representation of generalized Toffoli gates and some constructors
`Circuits`	representation of circuits (sequences of gates) and of the special "wires" of our circuits
`Synthesis`	synthesis algorithm for circuits with particular properties
`ArithCirc`	creation of arithmetic circuits
`EvalZ`	evaluation of circuits on concrete values
`FormAsList`	representation of formulas as xor-lists of and-lists of literals-as-strings
`FormAsMaps`	representation of formulas as xor-maps of and-maps of literals-as-Int
`FormAsBitmaps`	representation of formulas as xor-maps of bitmaps
`SymbEval`	Symbolic evaluation of circuits
`SymbEvalSpecialized`	Symbolic evaluation of circuits specialized to the representation from `FormAsBitmaps`
`QAlgos`	generating the circuits themselves
`RunQAlgos`	running the actual circuits
`Trace`	utilities for tracing and debugging

Figure 5. Modules and their services

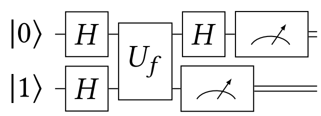


Figure 6. Quantum Circuit for the Deutsch-Jozsa Algorithm ($n = 1$)

function. In other words, the outcome of the algorithm can be immediately decided if the formula is anything other than 0 or 1. Indeed, our implementation correctly identifies all 12870 balanced functions $[\mathbf{2}^4] \to [\mathbf{2}]$. This is significant as some of these functions produce complicated entangled patterns during quantum evolution and could not be de-quantized using previous approaches [1]. A word of caution though: our results assume a "white-box" complexity model rather than a "black-box" complexity model [12].

The discussion above suggests that the details of the equations may not be particularly relevant for some algorithms. This would be crucial as the satisfiability of general boolean equations is, in general, an *NP*-complete problem [5, 11, 17]. Fortunately, this observation does hold for other algorithms as well including the Bernstein-Vazirani algorithm and Grover's algorithm. In both cases, the result can be immediately read from the formula. In the Bernstein-Vazirani case, formulae are guaranteed to be of the form $x_1 \oplus x_3 \oplus x_4 \oplus x_5$; the secret string is then the binary number that has a 1 at the indices of the relevant variables $\{1, 3, 4, 5\}$. In the case for Grover, because there is a unique input u for which $f(u) = 1$, the ANF formula must include a subformula matching the binary representation of u, and in fact that subformula is guaranteed to be the shortest one as shown in Fig. 7.

Shor's Algorithm. The circuit in Fig. 9 uses a hand-optimized implementation of quantum oracle U_f for the modular exponentiation function $f(x) = 4^x \mod 15$ to factor 15 using Shor's algorithm. The white dot in the graphical representation of the first indicates that the control is active when it is 0. In a conventional forward execution, the state before the QFT block is:

$$\frac{1}{2\sqrt{2}}((|0\rangle + |2\rangle + |4\rangle + |6\rangle)\,|1\rangle + (|1\rangle + |3\rangle + |5\rangle + |7\rangle)\,|4\rangle)$$

At this point, the ancilla register is measured to either $|1\rangle$ or $|4\rangle$. In either case, the computational register snaps to a state of the form $\sum_{r=0}^{3} |a + 2r\rangle$ whose QFT has peaks at $|0\rangle$ or $|4\rangle$ making them the most likely outcomes of measurements of the computational register. If we measure $|0\rangle$, we repeat the experiment; otherwise we infer that the period is 2.

In the retrodictive execution, we can start with the state $|x_2x_1x_0001\rangle$ since 1 is guaranteed to be a possible ancilla measurement (corresponding to $f(0)$). The first CX-gate changes the state to $|x_2x_1x_0x_001\rangle$ and the second CX-gate produces $|x_2x_1x_0x_00x_0\rangle$. At that point, we reconcile the retrodictive result of the ancilla register $|x_00x_0\rangle$ with the initial condition $|000\rangle$ to conclude that $x_0 = 0$. In other words, in order to observe the ancilla at 001, the computational register must be initialized to a superposition of the form $|??0\rangle$ where the least significant bit must be 0 and the other two bits are unconstrained. Expanding the possibilities, the first register needs to be in a superposition of the states $|000\rangle, |010\rangle, |100\rangle$ or $|110\rangle$ and we have just inferred using purely classical but retrodictive reasoning that the period is 2.

$u = 0$	$1 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_2x_3$ $\oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 1$	$x_0 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 2$	$x_1 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 3$	$x_0x_1 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 4$	$x_2 \oplus x_2x_3 \oplus x_1x_2 \oplus x_0x_2 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 5$	$x_0x_2 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 6$	$x_1x_2 \oplus x_1x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 7$	$x_0x_1x_2 \oplus x_0x_1x_2x_3$
$u = 8$	$x_3 \oplus x_2x_3 \oplus x_1x_3 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$
$u = 9$	$x_0x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$
$u = 10$	$x_1x_3 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$
$u = 11$	$x_0x_1x_3 \oplus x_0x_1x_2x_3$
$u = 12$	$x_2x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2x_3$
$u = 13$	$x_0x_2x_3 \oplus x_0x_1x_2x_3$
$u = 14$	$x_1x_2x_3 \oplus x_0x_1x_2x_3$
$u = 15$	$x_0x_1x_2x_3$

Figure 7. Result of retrodictive execution for the Grover oracle ($n = 4$, w in the range $\{0..15\}$). The highlighted red subformula is the binary representation of the hidden input u.

Base	Equations				Solution
$a = 11$	$x_0 = 0$				$x_0 = 0$
$a = 4, 14$	$1 \oplus x_0 = 1$	$x_0 = 0$			$x_0 = 0$
$a = 7, 13$	$1 \oplus x_1 \oplus x_0x_1 = 1$	$x_0x_1 = 0$	$x_0 \oplus x_1 \oplus x_0x_1 = 0$	$x_0 \oplus x_0x_1 = 0$	$x_0 = x_1 = 0$
$a = 2, 8$	$1 \oplus x_0 \oplus x_1 \oplus x_0x_1 = 1$	$x_0x_1 = 0$	$x_1 \oplus x_0x_1 = 0$	$x_0 \oplus x_0x_1 = 0$	$x_0 = x_1 = 0$

Figure 8. Equations generated by retrodictive execution of $a^x \mod 15$ for different values of a, starting from observed result 1 and unknown $x_8x_7x_6x_5x_4x_3x_2x_1x_0$. The solution for the unknown variables is given in the last column.

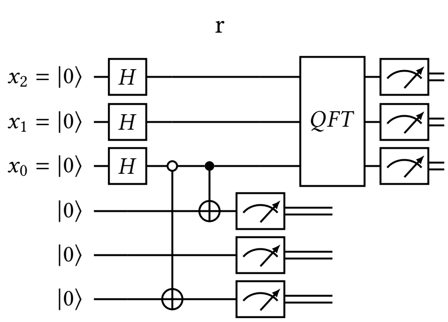


Figure 9. Finding the period of $4^x \mod 15$

This result does not, in fact, require the small optimized circuit of Fig. 9. In our implementation, modular exponentiation circuits are constructed from first principles using adders and multipliers [18]. In the case of $f(x) = 4^x \mod 15$, although the unoptimized constructed circuit has 56,538 generalized Toffoli gates (controlled^n-not gates for all n), the execution results in just two simple equations: $x_0 = 0$ and $1 \oplus x_0 = 1$. Furthermore, as shown in Fig. 8, the shape and size of the equations is largely insensitive to the choice of 4 as the base of the exponent, leading in all cases to the immediate conclusion that the period is either 2 or 4. When the solution is $x_0 = 0$, the period is 2, and when it is $x_0 = x_1 = 0$, the period is 4.

The remarkable effectiveness of retrodictive computation of the Shor instance for factoring 15 is due to a coincidence: a period that is a power of 2 is clearly trivial to represent in the binary number system which, after all is expressly designed for that purpose. That coincidence repeats itself when factoring products of the (known) Fermat primes: 3, 5, 17, 257, and 65537, and leads to small circuits [8]. This is confirmed with our implementation which smoothly deals with unoptimized circuits for factoring such products. Factoring 3*17=51 using the unoptimized circuit of 177,450 generalized Toffoli gates produces just the 4 equations: $1 \oplus x_1 = 1$, $x_0 = 0$, $x_0 \oplus x_0x_1 = 0$, and $x_1 \oplus x0x1 = 0$. Even for 3*65537=196611 whose circuit has 4,328,778 generalized Toffoli gates, the execution produces 16 small equations that refer to just the four variables x_0, x_1, x_2, and x_3 constraining them to be all 0, i.e., asserting that the period is 16.

Since periods that are powers of 2 are rare and special, we turn our attention to factoring problems with other periods.

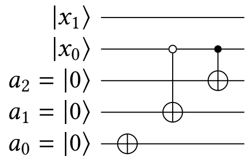


Figure 10. Finding the period of $4^x \mod 21$ using qutrits. The three gates are from left to right are the X, SUM, and $C(X)$ gates for ternary arithmetic [3]. The X gate adds 1 modulo 3; the controlled version $C(X)$ only increments when the control is equal to 2, and the SUM gates maps $|a, b\rangle$ to $|a, a + b\rangle$.

The simplest such problem is that of factoring 21 with an underlying function $f(x) = 4^x \mod 21$ of period 3. The unoptimized circuit constructed from the first principles has 78,600 generalized Toffoli gates; its execution generates just three equations. But even in this rather trivial situation, the equations span 5 pages of text! (Supplementary Material). A small optimization reducing the number of qubits results in a circuit of 15,624 generalized Toffoli gates whose execution produces still quite large, but more reasonable, equations (Supplementary Material). To understand the reason for these unwieldy equations, we examine a general ANF formula of the form $X_1 \oplus X_2 \oplus X_3 \oplus \ldots = 0$ where each X_i is a conjunction of some boolean variables, i.e., the variables in each X exhibit constructive interference as they must all be true to enable that $X = 1$. Since the entire formula must equal to 0, every $X_i = 1$ must be offset by another $X_j = 1$, thus exhibiting negative interference among X_i and X_j. Generally speaking, arbitrary interference patterns can be encoded in the formulae at the cost of making the size of the formulae exponential in the number of variables. This exponential blowup is actually a necessary condition for any quantum algorithm that can offer an exponential speed-up over classical computation [10].

It would however be incorrect to conclude that factoring 21 is inherently harder than factoring 15. The issue is simply that the binary number system is well-tuned to expressing patterns over powers of 2 but a very poor match for expressing patterns over powers of 3. Indeed, we show that by just using qutrits, the circuit and equations for factoring 21 become trivial while those for factoring 15 become unwieldy. The manually optimized circuit in Fig. 10 consists of just three gates; its retrodictive execution produces two equations: $x_0 = 0$ and $x_0 \neq 2$, setting $x_0 = 0$ and leaving x_1 unconstrained. The matching values in the qutrit system as 00, 10, 20 or in decimal 0, 3, 6 clearly identifying the period to be 3. The idea of adapting the computation to simplify the circuit and equations is inspired by the fact that entanglement is relative to a particular tensor product decomposition (Methods).

6 Conclusion

References

[1] Abbott, A. A. The Deutsch-Jozsa problem: de-quantization and entanglement. *Natural Computing 11* (2012).
[2] Bernstein, E., and Vazirani, U. Quantum complexity theory. *SIAM Journal on Computing 26*, 5 (1997), 1411–1473.
[3] Bocharov, A., Cui, S. X., Roetteler, M., and Svore, K. M. Improved quantum ternary arithmetic. *Quantum Info. Comput. 16*, 9–10 (jul 2016), 862–884.
[4] Burnett, L., Millan, W., Dawson, E., and Clark, A. Simpler methods for generating better boolean functions with good cryptographic properties. *Australasian Journal of Combinatorics 29* (2004), 231–247.
[5] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1971), STOC ’71, Association for Computing Machinery, p. 151–158.
[6] Deutsch, D. Quantum theory, the church–turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A 400* (1985).
[7] Deutsch, D., and Jozsa, R. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A 439* (1992).
[8] Geller, M. R., and Zhou, Z. Factoring 51 and 85 with 8 qubits. *Scientific Reports (3023) 3*, 1 (2013).
[9] Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1996), STOC ’96, Association for Computing Machinery, p. 212–219.
[10] Jozsa, R., and Linden, N. On the role of entanglement in quantum-computational speed-up. *Proceedings: Mathematical, Physical and Engineering Sciences 459*, 2036 (2003), 2011–2032.
[11] Karp, R. M. *Reducibility among Combinatorial Problems.* Springer US, Boston, MA, 1972, pp. 85–103.
[12] Komargodski, I., Naor, M., and Yogev, E. White-box vs. black-box complexity of search problems: Ramsey and graph property testing. *J. ACM 66*, 5 (jul 2019).
[13] Nielsen, M. A., and Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.
[14] Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing 26*, 5 (1997), 1484–1509.
[15] Simon, D. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 116–123.
[16] Tokareva, N. Chapter 1 - Boolean functions. In *Bent Functions*, N. Tokareva, Ed. Academic Press, Boston, 2015, pp. 1–15.
[17] Trakhtenbrot, B. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing 6*, 4 (1984), 384–400.
[18] Vedral, V., Barenco, A., and Ekert, A. Quantum networks for elementary arithmetic operations. *Phys. Rev. A 54* (Jul 1996), 147–153.
[19] Wegener, I. *The Complexity of Boolean Functions.* John Wiley & Sons, Inc., USA, 1987.