# Symbolic Execution of Hadamard-Toffoli Quantum Circuits

Jacques Carette
carette@mcmaster.ca
McMaster University
Hamilton, ON, Canada

Gerardo Ortiz
ortizg@iu.edu
Indiana University
Bloomington, IN, USA

Amr Sabry
sabry@indiana.edu
Indiana University
Bloomington, IN, USA

## Abstract

The simulation of quantum programs by classical computers is a critical endeavor for several reasons: it provides proof-of-concept validation of quantum algorithms; it provides opportunities to experiment with new programming abstractions suitable for the quantum domain, and most significantly it is a way to explore the elusive boundary at which a quantum advantage may materialize. Here, we show that traditional techniques of symbolic evaluation and partial evaluation yield surprisingly efficient classical simulations for some instances of textbook quantum algorithms that include the Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover, and Shor's algorithms. The success of traditional partial evaluation techniques in this domain is due to one simple insight: the quantum bits used in these algorithms can be modeled by a symbolic boolean variable while still keeping track of the correlations due to superposition and entanglement. More precisely, the system of constraints generated over the symbolic variables contains all the necessary quantum correlations and hence the answer to the quantum algorithms. With a few programming tricks explained in the paper, quantum circuits with millions of gates can be symbolically executed in seconds. Paradoxically, other circuits with as few as a dozen gates take exponential time. We reflect on the significance of these results in the conclusion.

**ACM Reference Format:**
Jacques Carette, Gerardo Ortiz, and Amr Sabry. 2022. Symbolic Execution of Hadamard-Toffoli Quantum Circuits. In *Proceedings of PEPM*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Classical models of computation are widely believed to be less powerful than quantum ones. Nevertheless, it is of utmost importance to establish when, for a given problem, a classical algorithm is as resource efficient as its quantum counterpart. In this paper we address this problem from a new angle, apparently, not explored before. We consider traditional techniques of symbolic and partial evaluation to classically simulate quantum circuits assembled from Hadamard and Toffoli gates. The latter constitutes a set of quantum gates that is known to be computationally universal.

The gist of our approach is rooted into two important observations. Since quantum algorithms are reversible, depending on the problem at hand, one can always take advantage of retrodictive execution as opposed to forward execution. Secondly, since Hadamard is a purely quantum gate, with no classical correspondence, we need to eliminate the superposition generated by such a gate and replace it with a symbolic variable that preserves the relevant quantum correlations.

revise outline

**Outline.**

We begin in Sec. 3 with background information covering the special quantum circuits of interest and a review of boolean functions and their algebraic normal form (ANF). Sec. 2 discusses the main ideas of symbolic execution of these circuits in detail. Our main contribution, the design and implementation, of a symbolic evaluator for Hadamard-Toffoli quantum circuits, is explained in Sec. 4. The next section (Sec. 5) includes a complexity analysis and a performance evaluation of our symbolic evaluator on major textbook algorithms. Sec. 6 concludes with a summary and a discussion of the broader implications of our approach to the understanding of the classical / quantum performance characteristics.

## 2 Qubits as Symbolic Variables

The general state of a quantum bit (qubit) is mathematically modeled using an equation parameterized by two angles $\theta$ and $\phi$ 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 $\theta$ determines the relative amplitudes of false and true and the angle $\phi$ 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

*PEPM, 2023, Boston*
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

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, a symbolic state that shares the same name in two positions accurately represents 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.

Because quantum circuits are reversible, i.e., executable forwards and backwards, the introduction of symbolic variables opens a host of new exciting possibilities beyond conventional (classical) symbolic evaluation: *any mixture of inputs and outputs can be made symbolic*. For example, consider again the Bell circuit in Fig. 2 but with an arbitrary initial value for the second qubit. The right subfigure Fig. 2b 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

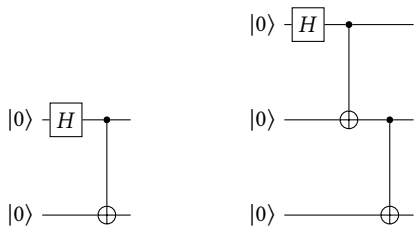



**Figure 1.** Bell and GHZ States

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 $x_1 = x_2 = 1$.

Technically, the problem of symbolic evaluation of our quantum circuits then reduces to a mixture of partial evaluation, slicing, and symbolic evaluation. Like with partial evaluation, we have some inputs dynamic, some static, and a static program. Similarly for slicing, but with outputs. Our situation is significantly simpler than in both cases. First, our language is reversible, which makes backwards evaluation deterministic, unlike for most languages. Second, the values at each step of circuit execution are boolean functions manipulated with conditional exclusive-or operations with a well-understood normal form, the algebraic normal form (ANF).

***Algebraic Normal Form (ANF).*** 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 $c$, where the effect is to leave all the control qubits unchanged and send $c$ to $c \oplus \bigwedge_i a_i$, 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 a control qubit and a boolean. In other words, our gates are specified by a collection $(a_n, b_n), \cdots, (a_0, b_0)$ together with one target qubit $c$; their action is to send the target qubit to $c \oplus \bigwedge_i (a_i == b_i)$, 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 Toffoli gates with $n$ control qubits are called $c^n x$ gates. It is worth noting the following special cases:

- for $n = 0$, we get the *not* gate X,
- for $n = 1$, we get the *controlled not* gate CX, and
- for $n = 2$, we get the *controlled controlled not* or the conventional *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 the exclusive-or ($\oplus$) of $\wedge$-clauses where each clause is the conjunction of 0 or more inputs $x_i$. Note that the conjunction of 0 inputs is 1 and that $1 \oplus F$ is the negation of $F$ which means that negation is not needed as a separate primitive. 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 [19, 22].

To summarize, we never need to residualize a circuit, we can always get a "closed form," in ANF, for evaluation,


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

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

whether forward or backward. Evaluating a circuit in one fixed direction is then quite standard. A novel approach, enabled by the reversibility of quantum mechanics, is what we call the *retrodictive* mode of running circuits [**?** ]. In that mode, illustrated in Fig. **??** , we start execution in the forward direction with a fully static collection of inputs in order to partially determine a possible future; we then execute backwards from the partially specified possible future (with the unknown values represented symbolically). This combination of static and dynamic knowledge of the output produces as its result a *system of constraints* equating the resulting logical polynomials to the circuit's inputs. If the system of constraints has a solution, then that output was feasible. What we will actually see is that for many quantum circuits, we can "read off" the information we need from the system of constraint themselves, *without needing to actually solve them.*

## 3 Quantum Algorithms

Let $[\mathbf{n}]$ denote the finite set $\{0, 1, \ldots, (n-1)\}$. The parameter $n$ determines the problem size for all the problems below (except Deutsch which is a fixed sized problem). In the review below, we adapt the usual presentation of the algorithms [2, 7, 8, 10, 14, 16, 17]). to one better suited to our context. In particular, instead of using the forward flow of execution using exact quantum superpositions, we express the problem as one asking for particular properties of boolean functions.

***Deutsch.*** The conventional statement of the problem is to determine if a function $[2] \to [2]$ is constant or balanced. In this small case, the function is balanced if it is the identity or boolean negation, and is constant otherwise. Equivalently, we can ask about the pre-image of an arbitrary boolean (say false), i.e. the set of inputs that are mapped to false by the function, and check whether the pre-image has an even or odd number of elements. If the cardinality of the pre-image is even i.e. 0 or 2, the function must be constant and if it is odd, i.e., it contains just one element, the function must be balanced.

***Deutsch-Jozsa.*** The problem is a generalization of the previous one: the question is to determine if a function $[2^n] \to [2]$ for some $n$ is constant or balanced. When expressed as a pre-image computation, the problem reduces to a query distinguishing the following three situations about the pre-image of a value in the range of the function: is the cardinality of the pre-image equal to 0, $2^n$, or $2^{n-1}$? In the first two cases, the function is constant and in the last case, the pre-image contains half the values in the domain indicating that the function is balanced.

***Bernstein-Vazirani.*** We are given a function $f : [2^n] \to [2]$ that hides a secret number $s \in [2^n]$. We are promised the function is defined using the binary representations $\sum_i^{n-1} x_i$ and $\sum_i^{n-1} s_i$ of $x$ and $s$ respectively as follows:

$$f(x) = \sum_{i=0}^{n-1} s_i x_i \mod 2$$

The goal is to determine the secret number $s$.

Expressing the problem as a pre-image calculation is slightly more involved than in the previous two cases. To determine $s$, we compute the pre-image of a value in the range of the function, and then make $n$ queries to this pre-image. Query $i$ asks whether $2^i$ is a member of the pre-image and the answer determines bit $i$ of the secret $s$. Indeed, by definition, $f(2^i) = s_i$ and hence $s_i$ is 1 iff $2^i$ is a member of the pre-image of 1.

***Simon.*** We are given a 2-1 function $f : [2^n] \to [2^n]$ with the property that there exists an $a$ such $f(x) = f(x \oplus a)$ for all $x$; the goal is to determine $a$. When expressed as a computation of pre-images, the problem statement becomes the following. Pick an arbitrary $x$ and compute the pre-image of $f(x)$. It must contain exactly two values one of which is $x$. The problem then reduces to finding the other value in the pre-image.

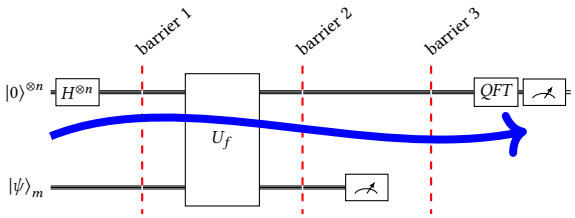
***Grover.*** We are given a function $f : [2^n] \to [2]$ such that there is a unique $u \in [2^n]$ such that $f(u) = 1$. The problem is to find this $u$.

***Shor.*** We are given a periodic function $f(x) = a^x \mod 2^n$ and the goal is to determine the period. As a computation over pre-images, the problem can be recast as follows. For an arbitrary $x$, compute the pre-image of $f(x)$ and query it to determine the period.

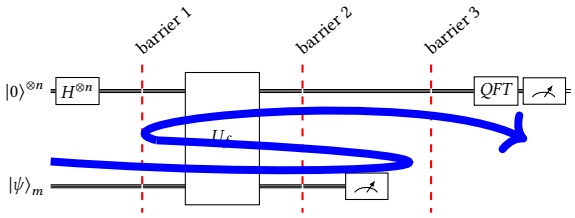
***Template for Circuits.*** All the problems above have solutions using quantum circuits that all fit the template in Fig. 3. The $U_f$ block is uniformlly defined as:

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

for all the problems. After replacing $H\,|0\rangle$ by a symbolic variable, the $U_f$ block ends up being completely classical, albeit performing mixed mode execution of the circuit. More




**(a)** Conventional Flow




**(b)** Retrodictive Flow

**Figure 3.** Template quantum circuit

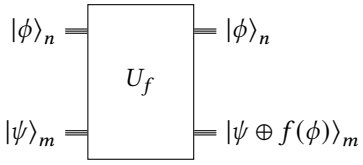



**Figure 4.** Circuit Abstraction

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.

# 4 Design and Implementation

Our exposition of the design and implementation of our system will follow Parnas' advice on *faking it* [15]: 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 quantum 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 and be executable forwards, backwards, and mixed retrodictive mode.

A number of quantum algorithms are expressed in the "black-box model" where the $U_f$ block defines arbitrary boolean functions (given as black boxes at "compile time"). To actually execute these algorithms, we want to, offline, synthesize a circuit from the boolean specification. 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, symbolically, forwards, backwards, retrodictive),

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 ancillae 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* [5] 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
```

| 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

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 circuit template of 2, 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 a well-estbalished algorithm [18]. 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 textbook algorithms, and are not optimized in any way, neither for running time nor for gate count. Neither are the code for the 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 woefully 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 it 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 occurrences. 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

We first give interesting aspects of running the six quantum algorithms outlined in Sec. 5.

### 5.1 Symbolic execution of the Algorithms

Most of the algorithms end up generating differently shaped constraint systems, and thus each need to be examined on its own. It is worth noting that all these algorithms do not have known fast classical versions. We spend more time on the analysis of Shor's algorithm, as it is both more important and displays subtle behaviour.

***Deutsch and Deutsch-Jozsa.*** 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) *regardless of how large the circuit is*. 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].

**Insight.** In these algorithms, 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 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 $[2^4] \to [2]$. 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]. However, it is important to remember that our circuits are "white-box" rather than "black-box", which yields a very different complexity model [13].

**Significance.** That the details of the equations do not matter is crucial as the satisfiability of generally boolean equation is, in general, an *NP*-complete problem [6, 12, 20].

***Bernstein-Vazirani.***

**Insight.** Here too the result can be immediately read off the formula.

$u = 0$ $1 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \oplus x_2x_3 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 1$ $x_0 \oplus x_0x_3 \oplus x_0x_2 \oplus x_0x_1 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 2$ $x_1 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_1 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 3$ $x_0x_1 \oplus x_0x_1x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 4$ $x_2 \oplus x_2x_3 \oplus x_1x_2 \oplus x_0x_2 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 5$ $x_0x_2 \oplus x_0x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 6$ $x_1x_2 \oplus x_1x_2x_3 \oplus x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 7$ $x_0x_1x_2 \oplus x_0x_1x_2x_3$

$u = 8$ $x_3 \oplus x_2x_3 \oplus x_1x_3 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 9$ $x_0x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 10$ $x_1x_3 \oplus x_1x_2x_3 \oplus x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 11$ $x_0x_1x_3 \oplus x_0x_1x_2x_3$

$u = 12$ $x_2x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_3 \oplus x_0x_1x_2x_3$

$u = 13$ $x_0x_2x_3 \oplus x_0x_1x_2x_3$

$u = 14$ $x_1x_2x_3 \oplus x_0x_1x_2x_3$

$u = 15$ $x_0x_1x_2x_3$

**Figure 6.** 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 $u$.

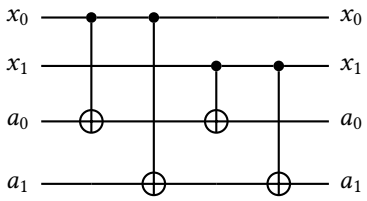
The formulae are guaranteed to be of the form $x_1 \oplus x_3 \oplus x_4 \oplus x_5$; the secret string is then the binary number that has a 1 at the indices of the relevant variables $\{1, 3, 4, 5\}$.

***Grover.***

**Insight.** For Grover as well, the result can be immediately read off the formula.

For Grover, 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. 6.

***Simon.*** The circuit below demonstrates the situation when $n = 2$ and $a = 3$.




The circuit implements the black box $U_f(x, a) = (x, f(x) \oplus a)$. We first pick a random $x$, say $x = 3$, fix the initial condition $a = 0$ and run the circuit forward. This execution produces, in the second register, the value of $f(x) = 0$. We now run a symbolic retrodictive execution with $a = 0$ at the output site. That execution produces information on all values of $a$ that are consistent with the observed result. In this case, we get: $a_0 = x_0 \oplus x_1$ and $a_1 = x_0 \oplus x_1$. In other words, when $x_0 = x_1$, we have $a = 0$, and when $x_0 \neq x_1$, we have $a = 3$ which is indeed the desired hidden value.

**Insight.** Simon's problem does not seem to have a resolution that is easy to read from the resulting equations.

***Shor.*** The circuit in Fig. 8 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 be 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.

This result does not, in fact, require the small optimized circuit of Fig. 8. In our implementation, modular exponentiation circuits are constructed from first principles using

| 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 7.** 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.

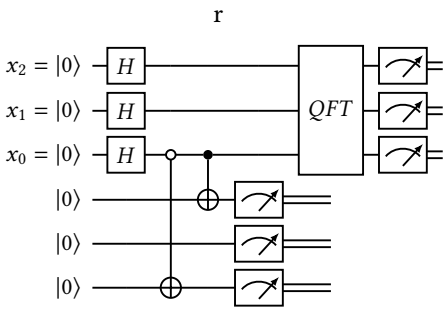



**Figure 8.** Finding the period of $4^x \mod 15$

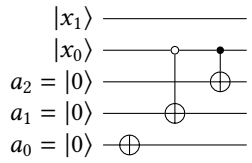



**Figure 9.** Finding the period of $4^x \mod 21$ using qutrits. The three gates are from left to right are the $X$, SUM, and $C(X)$ gates for ternary arithmetic [3]. The $X$ gate adds 1 modulo 3; the controlled version $C(X)$ only increments when the control is equal to 2, and the SUM gates maps $|a, b\rangle$ to $|a, a + b\rangle$.

adders and multipliers [21]. In the case of $f(x) = 4^x \mod 15$, although the unoptimized constructed circuit has 56,538 generalized Toffoli gates, the execution results in just two simple equations: $x_0 = 0$ and $1 \oplus x_0 = 1$. Furthermore, as shown in Fig. 7, 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 [9]. 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. 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! 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. 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 [11].

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. 9 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.

### 5.2 Complexity

In the general case, we have a circuit containing $T$ generalized Toffoli gates over $n+m$ qubits split in two registers $A$ ($n$ qubits) and $B$ (m qubits). The typical symbolic execution takes the following steps with the given worst-case complexity:

1. If the quantum algorithm is expressed in terms of calls to a black-box oracle (all the problems we consider except Shor), then the first step is to design the oracle *efficiently*. Perhaps surprisingly, it turns out we don't have to be particularly clever in designing that circuit: textbook designs with million of gates can work well.
2. Let $A = |00\ldots0\rangle$ and $B = |00\ldots0\rangle$ and run the circuit with classical inputs. This has complexity $O(T)$ as it takes $T$ steps where each step takes constant time. The result of this evaluation will leave $A$ intact and produce some value $b$ for the $B$ registerr.
3. We now run the circuit backwards with the symbolic values $A = |x_0x_1\ldots x_{n-1}\rangle$ and $B = |b\rangle$. This takes $T$ steps. At each step, we have $m$ ANF equations over the $\{x_0, x_1, \ldots, x_{n-1}\}$ variables. The size of each equation might be $O(2^n)$ in the worst case. So the overall complexity of this step is $O(Tm2^n)$.
4. The answer to the algorithm is obtained by either inspecting or, in the worst case, solving the resulting $m$ equations. In the Deutsch-Jozsa and Grover algorithms, the solution is immediate by inspection of the equations.

There are two potential bottlenecks: steps (3) above which has a worst-case complexity of $O(Tm2^n)$, and step (4) in the solve case, which is an NP-complete problem. The $O(T)$ factor is inevitable because we have a white box implementation of the oracle and we must touch every gate in that implementation. The $O(m)$ factor is also inevitable as it represents the number of variables. What varies from one function to the other and, for a particular function, from one oracle implementation to the other is the $O(2^n)$ factor.

What our examples show is that while the worst-case is $O(2^n)$, it seems that the expected case actually depends on the length of the encoding (in binary) of the information contained in the answer, especially in the case where we do not need to solve the constraints.

## 6 Conclusion

Symbolic execution is a way of evaluating a given program abstractly, so that the abstraction represents multiple inputs sharing an evolution path through the program, with solutions encoded in equations or constraints. So far, this way of execution has been limited to the classical realm. In this work, we extended these ideas to the quantum realm by considering the computational quantum universality of Hadamard and Toffoli gates. The proposed replacement of $H|0\rangle$ by $|x\rangle$, where $x$ is a symbol, provides the key to capturing some of the entanglement (non-local correlations) present in those programs; however, the execution is classical. Surprisingly, in many well-known quantum algorithms (such as Deutsch, Deutsch-Jozsa, Bernstein-Vazirani, Simon, Grover) these correlations are sufficient to obtain the solution efficiently for some inputs with a plain *classical symbolic execution* as opposed to a purely quantum execution (involving states that belong to a complex vector space endowed with an inner product). This raises many questions, in particular, those foundational ones related to the origin behind the power of quantum computation.

## 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] Carette, J., Kiselyov, O., and Shan, C.-c. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming 19*, 5 (2009), 509–543.

[6] 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.

[7] Deutsch, D. Quantum theory, the church–turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A 400* (1985).

[8] Deutsch, D., and Jozsa, R. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A 439* (1992).

[9] Geller, M. R., and Zhou, Z. Factoring 51 and 85 with 8 qubits. *Scientific Reports (3023) 3*, 1 (2013).

[10] 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.

[11] 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.

[12] Karp, R. M. *Reducibility among Combinatorial Problems.* Springer US, Boston, MA, 1972, pp. 85–103.

[13] 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).

[14] Nielsen, M. A., and Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.

[15] Parnas, D. L., and Clements, P. C. A rational design process: How and why to fake it. *IEEE transactions on software engineering*, 2 (1986), 251–257.

[16] 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.

[17] Simon, D. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 116–123.

[18] Soeken, M., Dueck, G. W., and Miller, D. M. A fast symbolic transformation based algorithm for reversible logic synthesis. In *International Conference on Reversible Computation* (2016), Springer, pp. 307–321.

[19] Tokareva, N. Chapter 1 - Boolean functions. In *Bent Functions*, N. Tokareva, Ed. Academic Press, Boston, 2015, pp. 1–15.

[20] Trakhtenbrot, B. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing 6*, 4 (1984), 384–400.

[21] Vedral, V., Barenco, A., and Ekert, A. Quantum networks for elementary arithmetic operations. *Phys. Rev. A 54* (Jul 1996), 147–153.

[22] Wegener, I. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., USA, 1987.