**Implementing Associative Memory on a Quantum** **Computer Simulator**

William Bernoudy

**Abstract**

This paper presents an implementation of associative memory on a quantum computer down to the gate level, using an algorithm by Ventura and Martinez (2000). It gives an overview of the original algorithm as well as explains how the different components of the algorithm were accomplished using simple quantum gates. To illustrate this, the new algorithm is demonstrated by writing computer code for a quantum computer simulator.

**Introduction**

**How a Quantum Computer Works**

A quantum computer is a computer that takes advantage of quantum phenomena. Specifically, it is a computer that operates on qubits instead of bits. Qubits are pieces of information that interact in ways not possible with a classical understanding. For example, unlike a classical bit, a qubit can be in a superposition of multiple states at once.

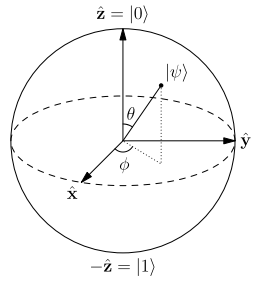
At any given point in time in a classical computer, one that operates only on bits, the bits are in a single state: a zero or a one. We can describe the two states with the basis vectors

We can now say that a given bit is in the state or the state. We can also use tensor products to conveniently describe the state of multiple bits. For example, if we want to represent a 3-bit system in the 010 state, we can write

Notice that 010 is two in binary, or the third number (starting from zero), and that the vector produced also has a “1” in the third position. In fact, for a given system of bits in the th binary state counting from zero, the vector corresponding to the state will consist of all zeros and one “1” in the th position.

Qubits function a little differently. Let us have a single qubit called . Instead of simply being in the or state like a classical bit, its state is given by the vector:

where and are complex numbers under the single constraint

This means that the values have an infinite number of possibilities. Unlike a classical bit, for which the vector could only consist of all zeros and one “1”, there are an infinite number of directions can be pointing. This is a mathematical way of saying that is not really in either the state or the state, but in a superposition of both. It exists as some amount of both states at the same time.

A single qubit can be visualized with a Bloch sphere, depicted to the left. As we saw with the single bit before, when the qubit is pointed directly upwards (the positive z axis), it has a value of 0. When it is pointed directly downwards (the negative z axis), it has a value of 1. However, unlike the bit, we can see that there are now two axes of rotation afforded to the qubit that were not present before (represented by and ). The current state of the qubit in the Bloch sphere is shown by the line with a dot on the end labeled .

Though a qubit may at some point exist in multiple states at the same time, when we measure the qubit, or observe it, its state will “collapse” to either the or the state. This means that we can never directly observe the superposition of any qubit. For the example qubit in the Bloch sphere, we can see that is pointed more closely upwards, towards the vector, than it is downwards. This means that if we were to observe this qubit, it is more likely that it would collapse to the state. If the qubit were pointed more towards the bottom of the sphere at the vector, it would be more likely that we would observe it collapse to the state. Note that these likelihoods correspond to the equation, where is the likelihood that the qubit will collapse to the th state.

The variability of qubits is furthered amplified when adding more qubits. Like a classical computer, the possible states for an -qubit quantum computer exist in a dimensional space. However, with a quantum computer, the vector which represents the state of the qubits does not have to be pointing in only one of the dimensions—it can exist in all of them (i.e. it can be in a superposition of all of them). For example, if we have a three-qubit system, then the general state of the qubits is given by:

Thus, at any point in the computation, the state of the qubits can be described with a vector of length . Similarly to the single qubit, when we measure the system, it will cause the “quantum-ness” of the qubits collapse and each will pick a single value of or . The probability of the system collapsing into a certain state is again given by the square of the magnitude of the value corresponding to that state.

Just like classical computation, quantum computation is done by sending qubits through “quantum gates.” These gates are simply ways of manipulating qubits according to known rules. In the same vein as the state of the computer, gates can conveniently represented as matrices. We can multiply the state of the system by a matrix corresponding to a gate to “simulate” the gate, as the resulting vector will be representative of the system after it has gone through the gate. The matrices representing gates are always square, matrices where is the number of qubits the gate acts upon. For example, let us have a system of qubits and a gate that acts on one qubit. We want to simulate applying the gate to the first of our qubits. This can be done by first creating a matrix produced by where is the identity matrix and the series of tensor products is long including the matrix. Then we multiply the vector representing the system state by this matrix, producing a vector representing the system state after going through the gate. We can think about “applying” the identity matrix to every qubit which is not going through a gate as returning the original value of the qubit.

**Associative Memory**

Associative memory deals with the problem of “learning” a set of patterns, and then being able to “recall” a certain pattern when presented with a part of the pattern. This problem has natural applications in AI, where we need a fast way to be able to “recall” information when presented with certain inputs, just as humans are able to quickly recognize and fill in the blanks of partial patterns. There are many simple classical algorithms that can achieve this task, such as storing the patterns in a database and then searching over the database (and each pattern) to find the one that matches. However, these can be slow and require lots of storage space. Another classical solution to this problem is using a Hopfield network, a type of neural network which can be trained with a set of patterns and then used to produce patterns when presented with a portion of one. However, if the patterns are of length , the neural network requires neurons, and is then limited in how many patterns it can store (usually less than half of ).

Quantum computing allows a significant improvement over neural network techniques. Dan Ventura and Tony Martinez, in a paper titled *Quantum Associative Memory*, present two algorithms that achieve associative memory that allow up to patterns to be stored on qubits, with a training complexity of and a recall complexity of .[[1]](#footnote-1)

In order to show how this algorithm might actually be implemented on a universal quantum computer, each stage of Ventura’s and Martinez’s (V&M) was achieved and simulated using only common quantum gates. Unless noted, all parts of the algorithms are taken directly from their paper.

**Explanation of Algorithms**

**Overview**

Given a set of patterns to be learned, each pattern is loaded one at a time onto the qubits until we have achieved a superposition of equal distribution among all the patterns. A slightly modified Grover’s algorithm can then be used to retrieve a pattern based on a known portion of it (Grover, 1996).

**Learning Stage**

For patterns of length , qubits are needed to store the pattern (called the storage qubits). However, during the learning stage, three extra qubits are necessary: one as an intermediate qubit which is always returned to (), and two as control bits ( and ). Note that this is a slight modification of V&M’s algorithm, which required qubits. This is because I used a Z-gate with controls on every one of the storage bits for the sake of much shorter simulation times.[[2]](#footnote-2) For the algorithm, we require two special gates:

is simply a controlled-not gate that flips the second bit if the first is zero, instead of one. It can be constructed by doing a NOT on the first qubit, a CNOT, and then another NOT on the first. allows us to slice the state that has a control bit value of into two states, one being of the superposition, and the other being of the superposition where is the number of patterns left.[[3]](#footnote-3)

Here are the steps of the algorithm for each pattern in *P*:

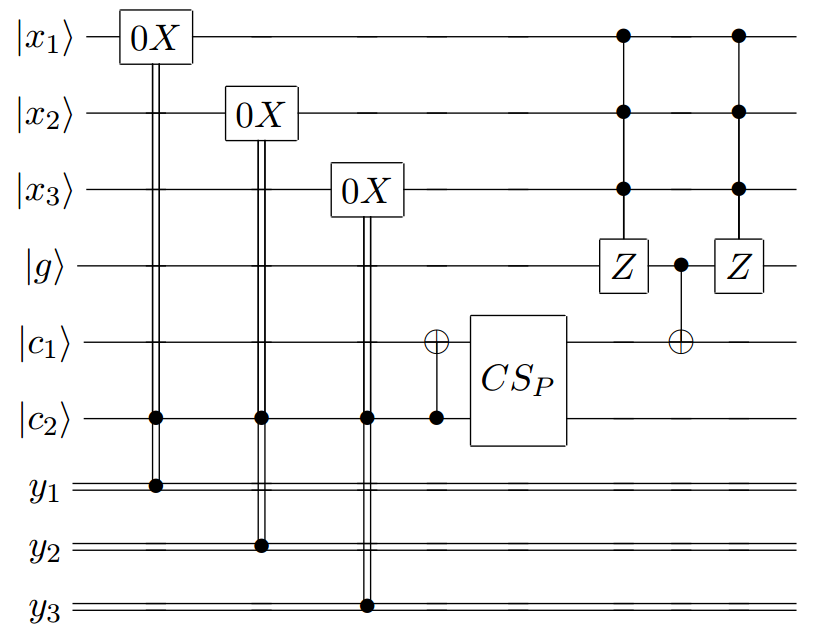
**Learning a single pattern on** *n* **qubits:**

1. Apply gates with control on to storage qubit if the corresponding bit in the pattern is a 1.
2. Apply gate with control on to .
3. Apply with control on onto , where is the number of patterns left including the pattern being learned.
4. Apply NOT gates to every qubit where the corresponding bit in the pattern is a 0.
5. Apply controlled NOT gate with controls on every storage qubit to .
6. Apply CNOT gate with control on to .
7. Repeat step 5.
8. Repeat step 4.

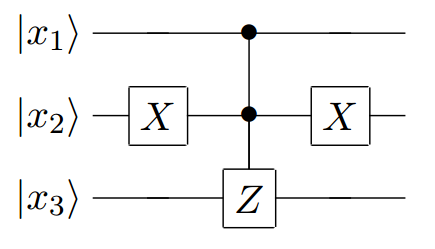
We can repeat this algorithm for every pattern until has been “learned” by the quantum computer. By the end of the algorithm, all three extra qubits, , and , will be unentangled from the rest of the system so we can safely discard them. Note that the single pattern learning algorithm is , and as we must apply it for patterns, the entire learning algorithm is .

**Recall Algorithm**

**Figure 1** This example is for loading patterns with a length of 3. The pattern would come in loaded on the classical bits, , and . The gate represents a controlled NOT except it applies the NOT when has a value of instead of .

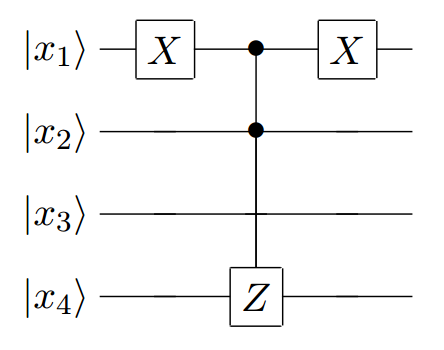


As previously stated, the recall algorithm uses a modified Grover search to find the matching learned pattern. The first difference is that we do not prepare the system in any way by applying Hadamard gates at the beginning, as we already have the superposition we want. The second is that at the beginning, since the Grover diffusion operator will also flip the phases of all the states representing the patterns (instead of just the desired pattern), we need another operator that flips all the phases of the patterns back. We call this operator . In terms of its matrix, it can be represented as an identity matrix where the th 1 is changed to a *−*1 if is the decimal representation of one of the patterns. We can construct it with gates by going through every pattern, applying a NOT gate on the qubits that correspond to zeros in the pattern, applying a controlled Z with controls on every qubit (except the one with the Z-gate, which can be any of them), and then re-applying the same NOT gates. This method can be seen in the function *constr-patterns-phaser* in my code (Appendix A). We will notate the normal Grover diffusion operator with .



**Figure 2** This example represents one pattern for . The pattern would be (1 0 1), so we need to apply the NOT gate to the second qubit because it is a zero.

We want the oracle for Grover’s algorithm to be a quantum circuit which flips the phase of the system if it is in the correct pattern. Thus, we can construct it by applying NOT gates to all qubits that correspond to zeros in the part of the pattern we know, applying a controlled Z gate with controls on all the qubits corresponding to bits we know onto one of the qubits corresponding to the qubits we don’t know, and then applying the same NOT gates again. This method can be seen in the function *constr-search-phaser* in my code (Appendix A). We will notate this oracle function with .



**Figure 3** This example would correspond to the partial sequence (0 1 ? ?). We apply a NOT gate to the first qubit because we know it is a zero, then a Z with controls on the known qubits with a target on one of the unknowns, and then re-apply the NOT.

**Recalling pattern given partial sequence**

1. Apply to the qubits.
2. Repeat times:
   1. Apply to the qubits.
3. Measure register.

Where , , , , and where is the number of patterns, is the number of marked states (states involving the unknown qubits that the controlled Z gates in did not act on, or, in the matrix representation, all the states corresponding to the *−*1s in the matrix) that do not correspond to learned patterns, and all the marked states that do correspond to learned states. In my code, I assumed that there was only one learned state that the recall should return, so I assumed is 1 and is equal to where is the number of unknowns in the partial sequence.[[4]](#footnote-4)

Though I have no understanding of , V&M assert that the second algorithm is still and with the same time complexity as Grover’s algorithm. This makes sense in that though it was modified slightly, the algorithm remains essentially the same as Grover’s.

**Usage of Code**

The code that I wrote to implement these algorithms has been included in Appendix A. It is written in Scheme and uses quetzal.rkt, a quantum computer simulator also written Scheme (Bernoudy, 2015). The learning algorithm is called *learn*, and the recall algorithm is called *Grover-part*. You can use them like so:

> (**define** P ’((0 0 0 0) (0 0 1 1) (0 1 1 0) (1 0 0 1) (1 1 0 0) (1 1 1 1)))

> (**learn** P)

> (**Grover-part** ’(0 1 1 ?) P)

> (**measure-register**)

The most likely result is |0110> with a probability of 0.8437500000000002

**Discussion and Conclusion**

Though in general I was able to successfully implement most parts of the algorithms outlined by V&M, there was one notable issue that I had: The calculation of often gave me the wrong value. As a result, the code would often apply the oracle and Grover diffusion operator too many times and thus the final superposition was not likely to return the correct value. For example, when 6 patterns of length 4 are learned, and a partial sequence with two unknowns is used to recall, V&M’s definition of gives . However, repeating the Grover step results in a final superposition with chance of returning the wrong pattern, and only a chance of retrieving the correct pattern. If we set , however, the final system state has a chance of returning the correct pattern. While this may be due to a mistake in my code or in my implementation of it, I have always been able to achieve a high rate of getting back the correct pattern using different values of , which suggests there is an issue with my (or V&M’s) calculation of .

**Appendix A**

This code may also be found on GitHub at https://github.com/rhyzomatic/qassoc.rkt.

#lang racket

(**require** math)

(**require** racket/vector)

(**require** 2htdp/batch-io)

(**define-namespace-anchor** a)

(**define** ns (**namespace-anchor->namespace** a))

(**require** "quetzal.rkt") ; note that quetzal.rkt must be present in current directory

;-----------------Start qassoc.rkt--------------------;

(**define** (**S-con** p)

(**matrix** [

[1 0 0 0]

[0 1 0 0]

[0 0 (**sqrt** (**/** (**sub1** p) p)) (**/** 1 (**sqrt** p))]

[0 0 (**/** 1 (**sqrt** p)) (**sqrt** (**/** (**sub1** p) p))]

]))

(**define** (**learn** patterns)

(**letrec** (**[bits** (**length** (**car** patterns))]

[qubits (**+** (**length** (**car** patterns)) 3)] ; n qubits for storage, 1 intermediate qubit, 2 control qubits

[C0NOT (**matrix\*** (**G-nqubit-constructor** 4 '(**0**) Pauli-X-gate) ; C0NOT is the same as CNOT surrounded by nots on the control qubit

CNOT-gate

(**G-nqubit-constructor** 4 '(**0**) Pauli-X-gate))]

[big-control-X (**matrix-stack** (**list** (**submatrix** (**identity-matrix** (**expt** 2 (**+** 1 bits))) (**-** (**expt** 2 (**+** 1 bits)) 2) (**expt** 2 (**+** 1 bits)))

(**matrix-row** (**identity-matrix** (**expt** 2 (**+** 1 bits))) (**-** (**expt** 2 (**+** 1 bits)) 1))

(**matrix-row** (**identity-matrix** (**expt** 2 (**+** 1 bits))) (**-** (**expt** 2 (**+** 1 bits)) 2))))])

(**initialize-register** (**build-list** qubits (**lambda** (**x**) 0))) ; Initialize all qubits to |0>

(**for** (**[pattern** patterns] [p-index (**length** patterns)])

(**for** (**[bit** pattern] [b-index bits])

(**if** (**=** bit 1)

(**apply-gate** register (**list** (**sub1** qubits) b-index) C0NOT) ; flip the corresponding qubit if the bit in the pattern is a 1

(**void**)))

(**apply-gate** register (**list** (**-** qubits 1) (**-** qubits 2)) C0NOT) ; flip the c1 control qubit

(**apply-gate** register (**list** (**-** qubits 2) (**-** qubits 1)) (**S-con** (**-** (**length** patterns) p-index))) ; apply the "save" gate to control qubits

(**for** (**[bit** pattern] [b-index bits])

(**if** (**=** bit 0)

(**apply-gate** register (**list** b-index) Pauli-X-gate) ; apply not gates on all qubits whose corresponding bit is 0

(**void**)))

(**apply-gate** register (**build-list** (**-** qubits 2) values) big-control-X) ; apply not gate with controls on all storage qubits on intermediate bit

(**apply-gate** register (**list** (**-** qubits 3) (**-** qubits 2)) CNOT-gate) ; CNOT with control on intermediate qubit, target on c1

(**apply-gate** register (**build-list** (**-** qubits 2) values) big-control-X) ; reverse the controlled not

(**for** (**[bit** pattern] [b-index bits]) ; reverse all the not gates to the storage qubits

(**if** (**=** bit 0)

(**apply-gate** register (**list** b-index) Pauli-X-gate)

(**apply-gate** register (**list** (**sub1** qubits) b-index) C0NOT))))

(**apply-gate** register (**list** (**-** qubits 1)) Pauli-X-gate) ; the algorithm will leave all states with the c2 bit as |1>, so not all these

(**remove-unentangled-qubit** register) ; We can now ignore the last three qubits as they are not entangled

(**remove-unentangled-qubit** register)

(**remove-unentangled-qubit** register)))

(**define** (**remove-unentangled-qubit** reg) ; removes the last qubit from the system (assumes it is unentangled)

(**let** [(**new-reg** (**submatrix** reg 1 (**build-list** (**/** (**matrix-num-cols** reg) 2) (**lambda** (**x**) (**\*** x 2)))))]

(**set-register** new-reg)

new-reg))

(**define** (**constr-big-control-Z** N)

(**diagonal-matrix** (**build-list** N (**lambda** (**x**) (**if** (**=** x (**-** N 1)) -1 1)))))

(**define** (**get-not-qubits** qubits mtau) ; gets all the qubits which we know should be flipped

(**cond** [(**null?** mtau) '()]

[(**eq?** (**car** mtau) 0) (**cons** (**-** qubits (**length** mtau)) (**get-not-qubits** qubits (**cdr** mtau)))]

[else (**get-not-qubits** qubits (**cdr** mtau))]))

(**define** (**constr-search-phaser** tau)

(**letrec** (**[qubits** (**length** tau)]

[N (**expt** 2 qubits)]

[find-unknown (**lambda** (**mtau**) ; get an unknown to apply a controlled Z gate to

(**if** (**eq?** (**car** mtau) '?) (**-** qubits (**length** mtau))

(**find-unknown** (**cdr** mtau))))]

[known-bits (**filter** (**lambda** (**x**) (**not** (**eq?** (**list-ref** tau x) '?))) (**build-list** qubits values))]

[big-control-Z (**constr-big-control-Z** (**expt** 2 (**+** (**length** known-bits) 1)))]

[to-not-gate (**lambda** (**q**) ; applies an X gate

(**G-nqubit-constructor** N (**list** q) Pauli-X-gate))]

[to-Z-gate (**lambda** (**q**) ; applies the Z gate to the two possible states of the unknown qubit

(**matrix\*** (**G-nqubit-constructor** N (**append** known-bits (**list** q)) big-control-Z) ; first a Z on the unknown

(**to-not-gate** q) ; flip the unknown

(**G-nqubit-constructor** N (**append** known-bits (**list** q)) big-control-Z) ; another Z

(**to-not-gate** q) ; flip it back to preserve the state

))])

(**eval** (**cons** matrix\* (**append**

(**map** to-not-gate (**get-not-qubits** qubits tau)) ; first apply X to all the qubits that should be flipped (so that if we have the right pattern, all qubits are 1)

(**list** (**to-Z-gate** (**find-unknown** tau))) ; controlled Z gate on on of the unknowns

(**map** to-not-gate (**reverse** (**get-not-qubits** qubits tau))))) ns) ; apply the same Xs in reverse order to preserve state

))

(**define** (**constr-patterns-phaser** patterns)

(**letrec** (**[qubits** (**length** (**car** patterns))]

[N (**expt** 2 qubits)]

[big-control-Z (**constr-big-control-Z** N)]

[to-not-gate (**lambda** (**q**) ; applies an X gate

(**G-nqubit-constructor** N (**list** q) Pauli-X-gate))]

[constr-helper (**lambda** (**pats**)

(**cond** [(**null?** pats) (**list** (**identity-matrix** N))]

[else (**append** (**map** to-not-gate (**get-not-qubits** qubits (**car** pats)))

(**list** big-control-Z)

(**map** to-not-gate (**get-not-qubits** qubits (**car** pats)))

(**constr-helper** (**cdr** pats)))]))])

(**eval** (**cons** matrix\* (**constr-helper** patterns)) ns)

))

(**define** make-Hadamard (**lambda** (**N**)

(**let** (**[constant** (**exact->inexact** (**/** 1 (**expt** 2 (**/** (**/** (**log** N) (**log** 2)) 2))))])

(**build-matrix** N N (**lambda** (**i** j)

(**\*** constant (**expt** -1 (**matrix-dot** (**bits->row-matrix** (**bits** i N)) (**bits->row-matrix** (**bits** j N))))))))))

(**define** (**calc-T** p r0 r1 N) ; p: total patterns, r0: # of states that don't correspond to patterns, r1: # that do

(**letrec** (**[a** (**/** (**\*** 2 (**-** p (**\*** 2 r1))) N)]

[b (**/** (**\*** 4 (**+** p r0)) N)]

[k (**+** (**-** (**\*** 4 a) (**\*** a b)) (**/** r1 (**+** r0 r1)))]

[l (**-** (**/** (**\*** 2 a (**+** N (**-** p r0 (**\*** 2 r1)))) (**-** N r0 r1)) (**\*** a b) (**/** (**-** p r1) (**-** N r0 r1)))])

(**exact-round** (**/** (**-** (**/** pi 2) (**atan** (**/** (**\*** k (**sqrt** (**/** (**+** r0 r1) (**-** N r0 r1)))) l))) (**acos** (**-** 1 (**/** (**\*** 2 (**+** r1 r0)) N)))))))

(**define** (**Grover-part** tau patterns)

(**letrec** (**[qubits** (**length** tau)]

[N (**expt** 2 qubits)]

[all-qubits (**build-list** qubits values)]

[full-Hadamard (**make-Hadamard** N)]

[Hadamard (**lambda** (**reg**)

(**apply-gate** register all-qubits full-Hadamard))]

[search-phase-flip-gates (**constr-search-phaser** tau)]

[patterns-phase-flip-gates (**constr-patterns-phaser** patterns)]

[T (**calc-T** (**length** patterns) (**-** (**expt** 2 (**count** (**lambda** (**x**) (**eq?** x '?)) tau)) 1) 1 N)] ; Assume that there is only one match in the learned patterns

[big-control-Z (**constr-big-control-Z** N)])

(**apply-gate** register all-qubits search-phase-flip-gates) ; Apply the phase flip for the one we're searching for

(**Hadamard** (**apply-gate** (**Hadamard** register) (**reverse** all-qubits) big-control-Z)) ; One Grover diffusion

(**apply-gate** register all-qubits patterns-phase-flip-gates) ; Apply gate that flips the phase of all patterns

(**Hadamard** (**apply-gate** (**Hadamard** register) (**reverse** all-qubits) big-control-Z)) ; A second Grover diffusion

(**for** (**[i** T]) ; We apply the search phase flipper and the Gover diffusion T times

(**apply-gate** register all-qubits search-phase-flip-gates) ; Apply the phase flip for the one we're searching for

(**Hadamard** (**apply-gate** (**Hadamard** register) (**reverse** all-qubits) big-control-Z)) ; Apply a Grover diffusion

)))

**References**

Bernoudy, W. (2015, October 16). Quetzal. Retrieved January 31, 2016, from <https://github.com/rhyzomatic/quetzal>

Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. *In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (pp. 212-219). ACM.

Ventura, D., & Martinez, T. (2000). Quantum associative memory. *Information Sciences*, 124(1), 273-296.

1. The used here is called big O notation. This notation allows us to roughly classify the time (or space) needed by an algorithm to run as a function of the size of the input. For example, if we have an algorithm that runs in time, it means that if we give the algorithm a large input of size , it will take approximately operations for some constant to output the result. [↑](#footnote-ref-1)
2. Any single qubit gate with an arbitrary number of controls can be achieved by using a series of CNOT gates and twice the number of qubits. Note that this means my change is not really an improvement. [↑](#footnote-ref-2)
3. A better and more in depth explanation of how this gate works can be found in V&M’s paper. [↑](#footnote-ref-3)
4. If you play around with a few more examples, you might notice that the code will not always produce the correctly matched string. This is covered in the conclusion. [↑](#footnote-ref-4)