Supplementary Material for "Quantum Algorithms for Weighted Constrained Sampling and Weighted Model Counting"

Fabrizio Riguzzi

Department of Mathematics and Computer Science, University of Ferrara, Via Machiavelli 30, Ferrara, 44121, Italy.

Corresponding author(s). E-mail(s): fabrizio.riguzzi@unife.it;

Abstract

We present the proofs of the lemmas in Section 4, a brief introduction to the quantum computing concepts used in the paper, the code for the quantum algorithms in Q# and Qiskit, and the result of a run of the algorithms applied to the problem of WMC, MPE and MAP.

1 Proofs of Lemmas of Section 4

Lemma 5. *Let $N = 2^n$ and ϕ be a black box function. Assume that A_ϕ is a probabilistic algorithm that makes queries to ϕ and returns an element $x \in \mathbb{B}^n$. If, for any non-constant ϕ, the probability that $\phi(x) = 1$ is at least $p > 0$, then there is a function ϕ' such $A_{\phi'}$ makes at least pN queries*[1].

Proof. Let ϕ_y be a Boolean function such that

$$\phi_y(z) = \begin{cases} 1 \text{ if } z = y \\ 0 \text{ otherwise} \end{cases}$$

Let $P_y(s)$ be the probability that A_{ϕ_y} returns y using s queries. Suppose the algorithm samples the configurations to query with a probability distribution that assigns probability q_z to z. We have that $\sum_{z \in \mathbb{B}^n} q_z = 1$

We show by induction that

$$\sum_{y \in \mathbb{B}^n} P_y(s) \leq s$$

For $s = 1$, we have

$$\sum_{y \in \mathbb{B}^n} P_y(1) = \sum_{y \in \mathbb{B}^n} q_y = 1$$

Now suppose

$$\sum_{y \in \mathbb{B}^n} P_y(s-1) \leq s - 1$$

and consider the s-th query. A_{ϕ_y} queries $\phi(y)$ with probability q_y so $P_y(s) \leq P_y(s-1) + q_y$. So

$$\sum_{y \in \mathbb{B}^n} P_y(s) \leq \sum_{y \in \mathbb{B}^n} P_y(s-1) + \sum_{y \in \mathbb{B}^n} q_y \leq s - 1 + 1 = s$$

There are N different choices for y so there must exist one with

$$P_y(s) \leq \frac{s}{N}$$

because otherwise $P_y(s) > \frac{s}{N}$ for all y implies that $\sum_{y \in \mathbb{B}^n} P_y(s) > N \frac{s}{N} = s$.

By assumption $P_y(s) \geq p$, so $\frac{s}{N} \geq P_y(s) \geq p$ implies that $s \geq pN$. □

Lemma 6. *Let $N = 2^n$ and ϕ be a black box function with M configurations x for which $\phi(x) = 1$. Assume that A_ϕ is a probabilistic algorithm that makes queries to ϕ and returns an element $x \in \mathbb{B}^n$. If, for any ϕ with M solutions, the probability that $\phi(x) = 1$ is at least $p > 0$, then there is a function ϕ' such $A_{\phi'}$ makes at least $p\frac{N}{M}$ queries.*

Proof. Let ϕ_S be a Boolean function such that

$$\phi_S(z) = \begin{cases} 1 \text{ if } z \in S \\ 0 \text{ otherwise} \end{cases}$$

[1] This lemma differs from Lemma 5.1.1 in (Hirvensalo, 2013) because the bound on the number of queries here is pN rather than $pN - 1$, thus providing a tighter bound.

where S is a subset of $\mathbb{B}^n$ such that $|S| = M$. Let $P_S(s)$ be the probability that A_{ϕ_S} returns x such that $x \in S$ using s queries. Suppose the algorithm samples the configurations to query with a probability distribution that assign probability q_z to z. We have that $\sum_{z\in\mathbb{B}^n} q_z = 1$

We show by induction that

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq s\binom{N-1}{M-1}$$

For $s = 1$, we have

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(1) = \sum_{S\subseteq\mathbb{B}^n,|S|=M} \sum_{z\in S} q_z$$

Each z appears in a number of subsets that is $\binom{N-1}{M-1}$ so

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(1) = \sum_{z\in\mathbb{B}^n} \binom{N-1}{M-1} q_z = \binom{N-1}{M-1} \sum_{z\in\mathbb{B}^n} q_z = \binom{N-1}{M-1}$$

Now suppose

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s-1) \leq (s-1)\binom{N-1}{M-1}$$

and consider the s-th query. A_{ϕ_S} queries $\phi(z)$ with probability q_z so $P_S(s) \leq P_S(s-1) + \sum_{z\in S} q_z$. So

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq \sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s-1) + \sum_{S\subseteq\mathbb{B}^n,|S|=M} \sum_{z\in S} q_z$$

Again,

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq \sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s-1) + \sum_{z\in\mathbb{B}^n} \binom{N-1}{M-1} q_z$$

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq \sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s-1) + \binom{N-1}{M-1} \sum_{z\in\mathbb{B}^n} q_z$$

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq \sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s-1) + \binom{N-1}{M-1}$$

$$\sum_{S\subseteq\mathbb{B}^n,|S|=M} P_S(s) \leq s\binom{N-1}{M-1}$$

There are $\binom{N}{M}$ different choices for S so there must exist one with

$$P_S(s) \leq \frac{s}{\frac{N}{M}}$$

because otherwise $P_S(s) > \frac{s}{\frac{N}{M}}$ for all S implies that

$$\sum_{S \subseteq \mathbb{B}^n, |S|=M} P_S(s) > \binom{N}{M} \frac{sM}{N}$$

$$\sum_{S \subseteq \mathbb{B}^n, |S|=M} P_S(s) > \frac{N!}{(N-M)!M!} \frac{sM}{N} = s \frac{(N-1)!}{(N-M)!(M-1)!} = s \binom{N-1}{M-1}$$

By assumption $P_S(s) \geq p$, so $\frac{s}{\frac{N}{M}} \geq P_S(s) \geq p$ implies that $s \geq p\frac{N}{M}$. □

2 Introduction to Quantum Computing

Here we provide a brief introduction to quantum computing following (Nielsen and Chuang, 2010). The bit is at the basis of classical computing and has a single value, either 0 or 1. The *quantum bit* or *qubit* is a generalization of the bit and is at the basis of quantum computing. A qubit represents a state defined by a pair of complex numbers that can have various physical implementations. From a mathematical point of view, a qubit is a unit vector in the $\mathbf{C}^2$ space, where $\mathbf{C}$ is the set of complex numbers, i.e.,

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

where the normalization requirement is that $|\alpha|^2 + |\beta|^2 = 1$.

Qubits are represented using the Dirac notation, where $|\psi\rangle$ (read "ket psi") is a two dimensional column vector and $\langle\psi|$ (read "bra psi") is a two dimensional complex conjugate row vector. The notation $\langle\psi|\phi\rangle$ (read "braket") is the inner product of the $\mathbf{C}^2$ space, i.e., it is the dot product between $|\phi\rangle$ and the complex conjugate $|\psi\rangle^*$. The special states $|0\rangle$ and $|1\rangle$ are called *computational basis states* and form the orthonormal basis

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

for $\mathbf{C}^2$. Any qubit state $|\psi\rangle$ can be expressed as a linear combination of the computational basis states:

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

In this case we say that $|\psi\rangle$ is in a *superposition* of states $|0\rangle$ and $|1\rangle$. Note that computational basis state are orthonormal: $\langle 0|1\rangle = \langle 1|0\rangle = 0$.

2.1 Composite Systems

Whenever we have more than one qubit, we have a composite physical system (also called a *quantum register*) and the state space expands accordingly: for n qubits, their state is a unit vector in the $\mathbf{C}^{2^n}$ space and there are 2^n computational basis states,

e.g., if $n = 2$, the basis states are $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, and the state of the qubits can be written as

$$|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

where $\alpha_{00}, \ldots, \alpha_{11}$ are complex numbers that form a unit length vector, i.e., such that $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$. The state space of a composite physical system given the component physical systems can be obtained using the tensor product of the states of the components.

The tensor product of two qubits

$$|a\rangle = a_0\,|0\rangle + a_1\,|1\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

$$|b\rangle = b_0\,|0\rangle + b_1\,|1\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

is

$$|a\rangle \otimes |b\rangle = \begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{bmatrix} =$$

$$a_0 b_0\,|00\rangle + a_0 b_1\,|01\rangle + a_1 b_0\,|10\rangle + a_1 b_1\,|11\rangle$$

The tensor product $|a\rangle \otimes |b\rangle$ is also written succinctly as $|a\rangle\,|b\rangle$ or also as $|ab\rangle$, so for example $|0\rangle \otimes |0\rangle = |0\rangle\,|0\rangle = |00\rangle$.

The Dirac notation is also useful for representing outer products $|a\rangle\,\langle b|$:

$$|a\rangle\,\langle b| = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \begin{bmatrix} b_0^* & b_1^* \end{bmatrix} = \begin{bmatrix} a_0 b_0^* & a_0 b_1^* \\ a_1 b_0^* & a_1 b_1^* \end{bmatrix}$$

2.2 Measurement

One of the operations that can be performed on a quantum system is *measurement*. There are various types of measurements, the simplest is measurement in the computational basis. When we have a qubit in the state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ and we measure it in the computational basis, we obtain a classical bit as a result: 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Since the states of qubits are unit vectors, then $|\alpha|^2 + |\beta|^2 = 1$ and the probabilities of the outcomes are well-defined. We can also measure multi-qubit systems and in that case we obtain a vector of classical bits.

2.3 Density Operators

A qubit in a superposition state encodes uncertainty on the result of its measurement. In this case, the state is known with certainty, it is only the result of measurement that is uncertain. Sometimes we want to represent uncertainty also on the state of the system. For example, we may know that the system is in one of several states $|\psi_i\rangle$,

where i is an index, with respective probabilities p_i. In this case we can represent the state of the system using a *density operator* ρ defined by the equation

$$\rho = \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right|$$

so density operators are matrices. If the state $|\psi\rangle$ of a quantum system is known exactly, the system is said to be in a *pure state* and the density operator is simply $\rho = |\psi\rangle \langle\psi|$. Otherwise, the system is said to be in a *mixed state* and that it is in a *mixture* of different pure states in the ensemble $\{(p_i, |\psi_i\rangle)\}$ for ρ.

Quantum mechanics can be formulated in terms of density operators as well as in terms of state vectors. Each technique is more convenient in certain cases, for example, density operators are convenient when we want to identify the state of a subsystem of a quantum system.

Consider a composite system AB obtained by composing two systems A and B with density operators ρ^A and ρ^B. The joint state is described by a density operator $\rho^{AB} = \rho^A \otimes \rho^B$. The state of system ρ^A is also called *reduced density operator* and is obtained as

$$\rho^A = \operatorname*{tr}_B(\rho^{AB})$$

where tr_B is a map called the *partial trace over system* B and is defined to apply to a tensor product as

$$\operatorname*{tr}_B(|a_1\rangle \langle a_2| \otimes |b_1\rangle \langle b_2|) = |a_1\rangle \langle a_2| \operatorname{tr}(|b_1\rangle \langle b_2|) \tag{1}$$

where $|a_1\rangle$ and $|a_2\rangle$ are any two vectors in the state space of A, $|b_1\rangle$ and $|b_2\rangle$ are any two vectors in the state space of B, and tr is the trace operator on matrices defined as the sum of its diagonal elements

$$\operatorname{tr}(A) = \sum_i A_{ii}.$$

The trace operator has the following important property

$$\operatorname{tr}(|b_1\rangle \langle b_2|) = \langle b_2|b_1\rangle \tag{2}$$

When we extract the state of a subsystem of a system that is in a pure state, we can obtain a mixed state, meaning that we have uncertainty in the state of the subsystem. Consider a system with two qubits in the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$. Its density operator is

$$\begin{aligned}\rho^{AB} &= \left(\frac{|00\rangle + |11\rangle}{\sqrt{2}}\right)\left(\frac{\langle 00| + \langle 11|}{\sqrt{2}}\right) \\ &= \frac{|00\rangle \langle 00| + |11\rangle \langle 00| + |00\rangle \langle 11| + |11\rangle \langle 11|}{2}\end{aligned}$$

We want to compute the reduced density operator for the first qubit so

$$\begin{aligned}\rho^A &= \operatorname*{tr}_B(\rho) \\ &= \frac{\operatorname{tr}_B(|00\rangle\langle 00|) + \operatorname{tr}_B(|11\rangle\langle 00|) + \operatorname{tr}_B(|00\rangle\langle 11|) + \operatorname{tr}_B(|11\rangle\langle 11|)}{2} \\ &= \frac{|0\rangle\langle 0|\langle 0|0\rangle + |1\rangle\langle 0|\langle 0|1\rangle + |0\rangle\langle 1|\langle 1|0\rangle + |1\rangle\langle 1|\langle 1|1\rangle}{2} \\ &= \frac{|0\rangle\langle 0| + |1\rangle\langle 1|}{2}\end{aligned}$$

As you can see, the system is in a mixture of the ensemble $\{(1/2, |0\rangle), (1/2, |1\rangle)\}$. So, even if we have perfect knowledge of the system, we have uncertainty on the state of a subsystem.

2.4 Quantum Circuits

To present quantum algorithms, we use the quantum circuit model of computation, where each qubit corresponds to a wire and quantum gates are applied to sets of wires.

Quantum gates are linear operators represented by matrices with complex elements that must be unitary. A matrix is *unitary* if $MM^\dagger = M^\dagger M = I$, where $M^\dagger$ is the *adjoint* or *Hermitian conjugate* of a matrix M, i.e., the complex conjugate transpose matrix $M^\dagger = (M^*)^T$. We start from gates operating on single qubits that are described by matrices belonging to $\mathbf{C}^{2\times 2}$. For example, the quantum counterpart of the NOT Boolean gate for classical bits is the X gate, defined as

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

and drawn in circuits as in Figure 15a. Quantum gates can also be defined by the effect they have on an orthonormal basis. For example, applying the X gate to the computational basis produces:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

and

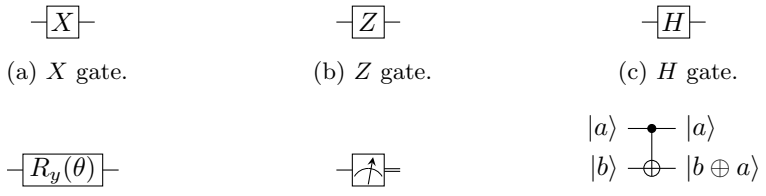
$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

so X exchanges $|0\rangle$ and $|1\rangle$, justifying its role as the quantum counterpart of the NOT Boolean gate.

The Z gate (see Figure 15b) is defined as

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

i.e, it transforms $|0\rangle$ to $Z|0\rangle = |0\rangle$ and $|1\rangle$ to $Z|1\rangle = -|1\rangle$.



(a) X gate. (b) Z gate. (c) H gate.

(d) R_y gate. (e) Measurement gate. (f) CNOT gate.

Fig. 15: Examples of quantum gates and measurements.

The *Hadamard* gate (see Figure 15c) is

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

and transforms $|0\rangle$ to $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Another useful gate is the parameterized R_y *rotation* gate (see Figure 15d)

$$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

Measurement can be seen as another type of operator applied to quantum states. In general, a measurement is described by a collection of measurement operators $\{M_m\}$. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement, then the probability that result m occurs is

$$P(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle$$

The measurement operators must satisfy the condition

$$\sum_m M_m^\dagger M_m = I$$

This condition ensures that the probabilities add up to one. A measurement of a qubit in the computational basis is given by the operators $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$. It's easy to verify that $M_0^\dagger M_0 = M_0^2 = M_0$, $M_1^\dagger M_1 = M_1^2 = M_1$, and $M_0^\dagger M_0 + M_1^\dagger M_1 = M_0 + M_1 = I$. This holds for all measurements in the computational basis. Measurement is drawn in circuits as in Figure 15e.

The results of measurement can also be computed with density operators. Suppose we have a measurement composed of measurement operators $\{M_m\}$ and that the system is described by density operator ρ given by the ensemble $\{(p_i, |\psi_i\rangle)\}$. If the system is in pure state $|\psi_i\rangle$, the probability of obtaining result m is

$$P(m|i) = \langle\psi_i| M_m^\dagger M_m |\psi_i\rangle = \mathrm{tr}(M_m^\dagger M_m |\psi_i\rangle\langle\psi_i|)$$

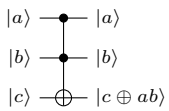


Fig. 16: CNOT with two control qubits.

where we applied the following property of the trace operator

$$\mathrm{tr}(A\,|\psi\rangle\,\langle\psi|) \;=\; \langle\psi|\,A\,|\psi\rangle$$

Then the marginal probability of result m is

$$\begin{aligned} P(m) \;&=\; \sum_i p(m|i)p_i \\ &=\; \sum_i p_i\,\mathrm{tr}(M_m^\dagger M_m\,|\psi_i\rangle\,\langle\psi_i|) \\ &=\; \mathrm{tr}(M_m^\dagger M_m \rho) \end{aligned}$$

Turning to composite systems, the most important two-qubit gate is *controlled-NOT* or *CNOT* that has two inputs, the control and the target qubits. The gate flips the target qubit if the control bit is set to 1 and does nothing if the control bit is set to 0. It can be seen as a gate that operates as $|ab\rangle \to |a, b \oplus a\rangle$, where $\oplus$ is the XOR operation, see Figure 15f.

Any multi-qubit logic gate can be composed from CNOT and single-qubit gates.

CNOT may be generalized to the case of more than two qubits: in this case, the extra qubits act as controls and the target is flipped if all controls are 1, see Figure 16. Moreover, given an operator U, it is possible to define a controlled-U operator defined as $|ab\rangle \to |a, U^a b\rangle$: if $a = 0$ it does nothing, otherwise it applies operator U to b.

Example 2. *Consider formula ϕ from Example 1:*

$$\phi = (S \to W) \wedge (R \to W) \wedge (S \wedge R \to).$$

Transforming the formula into conjunctive normal form[2] *we obtain*

$$\phi = (\neg S \vee W) \wedge (\neg R \vee W) \wedge (\neg S \vee \neg R)$$

The quantum circuit for computing the value of formula ϕ from Example 1 is shown in Figure 17.

Quantum circuits are read from left to right. Each line or wire corresponds to a qubit and starts in a computational basis state, usually $|0\rangle$ unless otherwise indicated. The circuit in Figure 17 contains one wire for each Boolean variable of Example 1, three wires that represent the so-called *ancilla qubits*, and one wire for the output

[2] A formula is in conjunctive normal form if it is a conjunction of clauses where a clause is a disjunction of literals and a literal is a propositional variable or its negation.

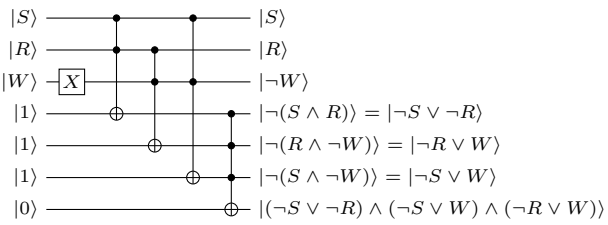


Fig. 17: Quantum circuit for computing ϕ

qubit. Ancilla qubits are used in order to make the circuit reversible, a requirement of quantum circuits. The bottom qubit will contain the truth value of the function ϕ at the end of the computation.

As can be seen, CNOT with more than one control qubit can be used to perform other Boolean functions. For example, the first CNOT with two control qubits computes the NAND of its control qubits because the target qubits is initially set to $|1\rangle$: in fact $|S, R, 1\rangle \rightarrow |S, R, (S \wedge R) \oplus 1\rangle = |S, R, \neg(S \wedge R)\rangle$. On the other hand, the last CNOT with three control qubits and a target qubit set initially set to $|0\rangle$ computes the AND of the control qubits.

3 Implementation

In this section we report the implementation of QWMC and QWCS applied to the problems of MPE and MAP for the sprinkler case of Example 1 using two languages for quantum computing: Q# by Microsoft (Svore et al, 2018) and Qiskit by IBM (Abraham and et al., 2019). The code is available at https://github.com/friguzzi/qwmc.

3.1 Q#

Q# is a domain-specific language for implementing quantum algorithms that is syntactically related to Python, C# and F#, see https://docs.microsoft.com/en-us/azure/quantum/user-guide/ for a complete user guide, here we will just present an introductory overview.

The main unit of Q# code is an *operation* which implements a sequence of quantum and classical operations. For example, the implementation of the sprinkler circuit of Example 2 is shown in Listing 1. It is applied to a query register and a target qubit and uses three ancilla qubits. With respect to Figure 17, it includes an extra qubit in the query register to implement the doubling of N. The operation flips the target qubit if the sprinkler formula takes value 1. `X` is the built-in X gate, `CCNOT` is the controlled X gate with two control qubits, and `Controlled X` is a multi-controlled X gate, used in line 13 with four control qubits. The use of the `within ... apply` construct allows to automate uncomputation - addition of the gates that revert the operations on the query register and ancilla bits in order to properly implement the classical computation as a quantum oracle.

Listing 1 Sprinkler circuit in Q#.

```
    operation Sprinkler(q : Qubit[],  target : Qubit) : Unit is Adj+Ctl
    {
        use a = Qubit[3];
        within {
            X(q[2]);
            X(a[0]);
            X(a[1]);
            X(a[2]);
            CCNOT(q[0], q[1], a[0]);
            CCNOT(q[1], q[2], a[1]);
            CCNOT(q[0], q[2], a[2]);
        } apply {
            Controlled X(a + [q[3]], target);
        }
    }
```

Operation `ApplyMarkingOracleAsPhaseOracle`, shown in Listing 2, takes as input a marking oracle (an oracle that flips the state of the target qubit) and a register of qubits, and transforms the marking oracle into a phase oracle (an oracle that flips the sign of the register). The X and H gates applied to the target qubit put it into the state $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$, then the marking oracle is applied. Since the target is in the $|-\rangle$ state, flipping the target if the register satisfies the oracle condition will bring the target to $-|-\rangle = -(|0\rangle - |1\rangle)\sqrt{2}$, applying a -1 factor to the whole register, effectively applying the phase oracle.

Listing 2 From a marking oracle to a phase oracle in Q#.

```
operation ApplyMarkingOracleAsPhaseOracle(markingOracle : ((Qubit[], Qubit) =>
    Unit is Adj+Ctl), q : Qubit[]) :  Unit is Adj+Ctl
{
    use target = Qubit();
    within {
        X(target);
        H(target);
    } apply {
        markingOracle(q, target);
    }
}
```

Listing 3 *Rot* operation in Q#.

```
operation Rot(q: Qubit[], weights : Double[]): Unit is Ctl+Adj
{
    for i in 0 .. Length(weights) - 1 {
        let theta = 2.0 * ArcSin(Sqrt(weights[i]));
        Ry(theta, q[i]);
    }
    H(q[3]);
}
```

Operation `Rot`, shown in Listing 3, implements the *Rot* gate of Figure 7, which applies rotation $R_y(\theta_i)$ to the ith qubit, where $\theta_i = 2 \arcsin \sqrt{w_i}$, and w_i the weight of the ith variable. The H gate is applied to the $n+1$th bit.

Operation `GroverIteration`, shown in Listing 4, takes as input a register and a marking oracle and implements the Weighted Grover operator WG of Figure 8. First it uses `ApplyMarkingOracleAsPhaseOracle` to apply the phase oracle to the register. Then it applies the adjoint (Hermitian conjugate) of gate *Rot* to all register qubits and multiplies by -1 all basis states except $|00\ldots0\rangle$. This is performed by multiplying by -1 only configuration $|00\ldots0\rangle$ (lines 7-13) and then rotating the register by π radians (line 14).

Listing 4 Grover iteration in Q#.

```
1  operation GroverIteration(q : Qubit[], oracle : ((Qubit[],Qubit) =>
2      Unit is Adj+Ctl), weights : Double[]) : Unit is Ctl+Adj
3  {
4      ApplyMarkingOracleAsPhaseOracle(oracle, q);
5      within {
6          Adjoint Rot(q, weights);
7          ApplyToEachCA(X, q);     // Brings |0..0> to |1..1>
8      } apply {
9          use a = Qubit();
10         Controlled X(q, a);   // Bit flips the a to |1> if register is |1...1>
11         Z(a);                 // The phase of a (and therefore the whole
12                               // register phase)
13                               // becomes -1 if above condition is satisfied
14         Controlled X(q, a);   // Puts a back in |0>
15         Ry(2.0 * PI(), q[0]);
16     }
17 }
```

Finally, operation `QWMC`, shown in Listing 5, implements the whole QWMC algorithm. First it defines a constant `t` – the number of bits to use to represent the weighted model count. It uses a four-qubit register for storing the state and a `t`-qubit register for storing the phase. It calls the `OracleToDiscrete` library function to convert the Grover operator to a library type `DiscreteOracle`, suitable for exponentiation. Then it initializes the qubits of the register according to the weights using `Rot`. Next, the code calls a library operation to perform quantum phase estimation (line 12). Then we measure the phase register to get an integer divided it by 2^t to obtain a number between 0 and 1. Finally, lines 19 and 20 compute the weighted model count from the phase.

Operation `QMPE`, shown in Listing 6, implements the MPE algorithm using QWCS. It uses a 4-qubit register for the state and initializes it according to the weights (line 6). The Grover operator is applied $\mathrm{CI}\left(\frac{\arccos \frac{\sum_{x;\phi(x)=1} W_x}{2}}{2 \arcsin \frac{\sum_{x;\phi(x)=1} W_x}{2}}\right) = \mathrm{CI}(0.763) = 1$ time (line 7) and finally the register is measured (lines 8, 9).

QMAP (Listing 7) differs from QMPE, because we measure only a subarray of the register, rather than the whole register (lines 8-9).

Listing 5 Quantum weighted model counting operator in Q#.

```
operation QWMC() : Double
{
    let t = 5;
    let weights = [0.55, 0.3, 0.7];
    let oracle = OracleToDiscrete(GroverIteration(_, Sprinkler(_, _),
      weights));

    use (q, p) = (Qubit[4], Qubit[t]);
    Rot(q, weights);
    // Allocate qubits to hold the eigenstate of U and the phase in a big
    // endian register
    let pBE = BigEndian(p);
    // Call library
    QuantumPhaseEstimation(oracle, q, pBE);
    // Read out the phase
    let phase = IntAsDouble(MeasureInteger(BigEndianAsLittleEndian(pBE)))
      / IntAsDouble(1 <<< (t));

    ResetAll(q + p);
    // The phase returned by QuantumPhaseEstimation is the value phi such
    // that e^{2pi phi} is an eigenvalue
    let angle = 2.0 * PI() * phase;
    let wmc = 2.0 * PowD(Sin(angle / 2.0), 2.0);
    // The 2.0 factor is added because there is an extra bit with weight 0.5
    // that is introduced to make the weighted count < 0.5

    return wmc;
}
```

Listing 6 Quantum MPE operator in Q#.

```
operation QMPE() : Result[]
{
    let weights = [0.55, 0.3, 0.7];
    use reg = Qubit[4];

    Rot(reg, weights);
    GroverIteration(reg, Sprinkler, weights);
    let query= reg[0 .. 2];
    let state = MultiM(query);

    ResetAll(reg);
    return state;
}
```

3.2 Qiskit

Qiskit is a Python library for implementing quantum algorithms. The code for performing QWMC presented in this section is based on the code for quantum counting available in (Asfaw and et al., 2020).

To use it, a number of import declaration are needed, see Listing 8.

Listing 9 shows the implementation of the sprinkler circuit by a function that takes as input a quantum circuit `qc` and two quantum registers, `q`, for the query qubits and `a`, for the ancilla qubits. Methods `x`, `ccx` and `mct` of `qc` implement respectively the X gate, the X gate with two control qubits and the X gate with a list of control qubits.

Listing 7 Quantum MAP operator. in Q#

```
operation QMAP() : Result[]
{
    let weights = [0.55, 0.3, 0.7];

    use reg = Qubit[4];
    Rot(reg, weights);
    GroverIteration(reg, Sprinkler, weights);
    let query = Subarray([0, 2],reg);
    let state = MultiM(query);

    ResetAll(reg);
    return state;
}
```

Listing 8 Importing components in Qiskit.

```
from qiskit import Aer
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute
from qiskit.tools.visualization import plot_histogram
from math import pi, sqrt, sin, asin
```

Listing 9 Sprinkler circuit in Qiskit.

```
def sprinkler(qc,q,a):
    qc.x(q[2])
    qc.x(a[0])
    qc.x(a[1])
    qc.x(a[2])
    qc.ccx(q[0],q[1],a[0])
    qc.ccx(q[1],q[2],a[1])
    qc.ccx(q[0],q[2],a[2])
    qc.mct([a[0],a[1],a[2],q[3]],a[3])
    qc.ccx(q[0],q[2],a[2])
    qc.ccx(q[1],q[2],a[1])
    qc.ccx(q[0],q[1],a[0])
    qc.x(a[2])
    qc.x(a[1])
    qc.x(a[0])
    qc.x(q[2])
```

Function `rotations` shown in Listing 10 uses a different approach to implement the *Rot* operator: it builds internally the query register and the quantum circuit and returns it. Then the method `to_gate` is used to convert the circuit into a new gate, `rot`, that implements *Rot*. Method `inverse` is used to obtain the inverse operation (the adjoint) `invrot` of *Rot*.

The weighted Grover operator WG is implemented by function `grover_circ` of Listing 11 that applies the same gates as operator `GroverIteration` of Listing 4. The circuit that is returned by `grover_circ` is then turned into a gate and converted into a controlled gate by method `control`.

Function `qft` of Listing 12 builds an n-qubit QFT circuit and is taken literally from (Asfaw and et al., 2020). The circuit is then transformed into a gate and inverted.

Listing 10 *Rot* circuit in Qiskit.

```
def rotations():
    q=QuantumRegister(4)
    qc=QuantumCircuit(q)
    weights=[0.55,0.3,0.7]
    for i in range(len(weights)):
       theta=2.0*asin(sqrt(weights[i]))
       qc.ry(theta,q[i])
    qc.h(q[3])
    return qc

rot=rotations().to_gate()
invrot=rot.inverse()
```

Listing 11 Weighted Grover circuit in Qiskit.

```
def grover_circ():
    q=QuantumRegister(4)
    a=QuantumRegister(5)
    qc=QuantumCircuit(q,a)
    qc.x(a[3])
    qc.h(a[3])
    sprinkler(qc,q,a)
    qc.h(a[3])
    qc.x(a[3])
    qc.append(invrot,range(4))
    for i in range(q.size):
        qc.x(q[i])
    qc.mct([q[0],q[1],q[2],q[3]],a[4])
    qc.z(a[4])
    qc.mct([q[0],q[1],q[2],q[3]],a[4])
    for i in range(q.size):
        qc.x(q[i])
    qc.ry(2*pi,q[0])
    qc.append(rot,range(4))

    return qc
grover = grover_circ().to_gate()
cgrover = grover.control()
```

Listing 13 builds the overall circuit with `t` counting qubits: it applies H to all counting qubits, it applies *Rot* to the searching qubits, it applies the series of controlled WG operations and finally the inverse QFT.

To simulate the quantum circuit we use the code in Listing 14 that executes the simulation 1000 times and collects the results in `result_sim`.

The code for performing MPE with QWCS is shown in Listing 15. The code for performing MAP differs because line 8 is replaced by `qc.measure([q[0],q[2]],c)`.

4 Example of Execution

In this section we present the results of the execution of QWMC, QWCS applied to MPE and MAP on the sprinkler case of Example **??**.

Here we report the results obtained with Q#. Those for Qiskit are similar. The circuit for QWMC was run 1000 times by simulation using ideal, noise free quantum

Listing 12 Quantum Fourier transform in Qiskit.

```
t = 5   # no. of counting qubits
n = 4   # no. of searching qubits

def qft(n):
    """Creates an n-qubit QFT circuit"""
    circuit = QuantumCircuit(n)
    def swap_registers(circuit, n):
        for qubit in range(n//2):
            circuit.swap(qubit, n-qubit-1)
        return circuit
    def qft_rotations(circuit, n):
        """Performs qft on the first n qubits in circuit (without swaps)"""
        if n == 0:
            return circuit
        n -= 1
        circuit.h(n)
        for qubit in range(n):
            circuit.cu1(pi/2**(n-qubit), qubit, n)
        qft_rotations(circuit, n)

    qft_rotations(circuit, n)
    swap_registers(circuit, n)
    return circuit

invqft = qft(t).to_gate().inverse()
```

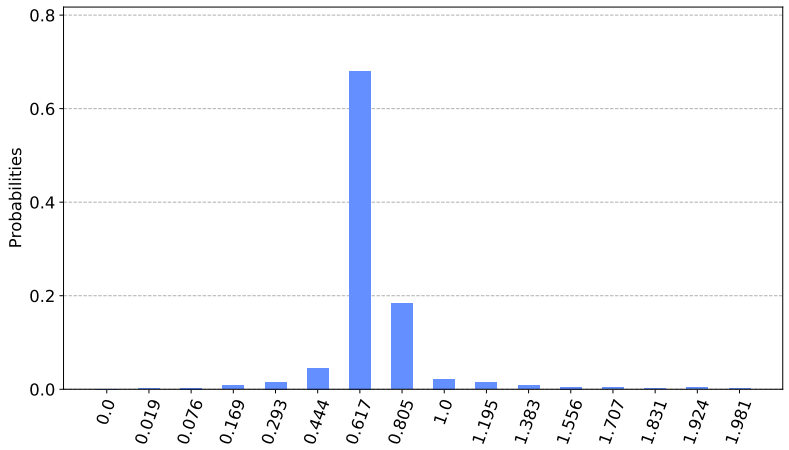


Fig. 18: Distribution of results for QWMC.

circuits. The distribution of results is shown in Figure 18: the peak at 0.617 shows that most simulations came very close to the true value of 0.679.

For MPE and MAP, we use the algorithm that repeatedly runs the circuits for QWCS and picks the most frequent result as the answer. QWCS was run by simulation 1000 times for MPE and MAP. Figures 19 and 20 show the distribution of results respectively for MPE and MAP: the highest peaks correspond to the solutions of the problem, 101 for MPE and 01 for MAP (see Section **??**).

Listing 13 Quantum weighted model counting operator in Qiskit.

```
q=QuantumRegister(n)
a=QuantumRegister(5)
s=QuantumRegister(t)
c=ClassicalRegister(t)

qc = QuantumCircuit(s,q,a,c) # Circuit with n+t qubits and t classical bits

for qubit in range(t):
    qc.h(qubit)

qc.append(rot, range(t,n+t))

# Begin controlled Grover iterations
iterations = 1
for qubit in range(t):
    for i in range(iterations):
        qc.append(cgrover, [qubit] + [*range(t, n+t+5)])
    iterations *= 2

# Do inverse QFT on counting qubits
qc.append(invqft, range(t))

# Measure counting qubits
qc.measure(range(t), range(t))
```

Listing 14 Simulation code in Qiskit.

```
backend = Aer.get_backend('qasm_simulator')
start = time.time()

job_sim = execute(qc, backend, shots=1000)
end = time.time()
print(end - start)

result_sim = job_sim.result()
```

Listing 15 Quantum MPE code in Qiskit.

```
q=QuantumRegister(4)
a=QuantumRegister(5)
c=ClassicalRegister(3)

qc=QuantumCircuit(q,a,c)
qc.append(rot, range(4))
qc.append(grover,range(9))
qc.measure([q[0],q[1],q[2]],c)
```

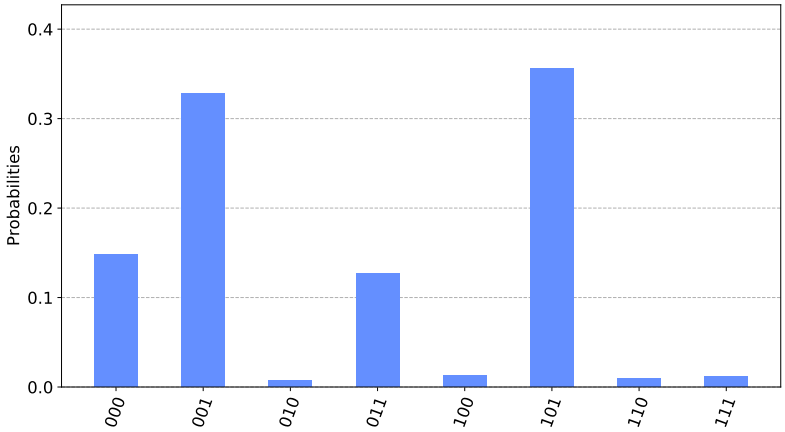


Fig. 19: Distribution of results for MPE.

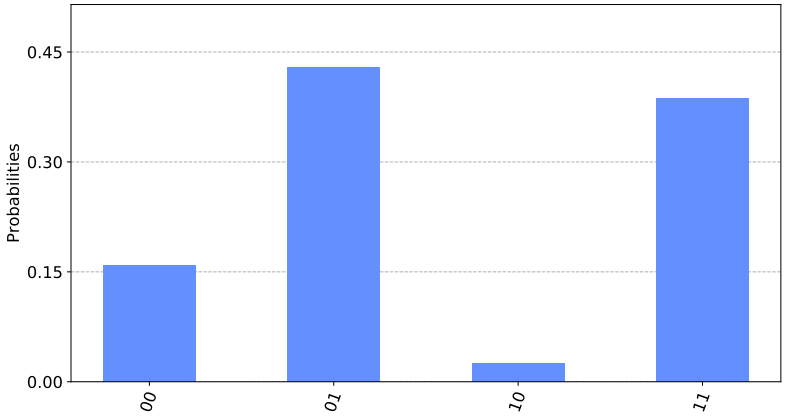


Fig. 20: Distribution of results for MAP.

References

Abraham H, et al. (2019) Qiskit: An open-source framework for quantum computing. https://doi.org/10.5281/zenodo.2562110, 10.5281/zenodo.2562110

Asfaw A, et al. (2020) Learn quantum computation using qiskit. http://community.qiskit.org/textbook

Hirvensalo M (2013) Quantum Computing. Natural Computing Series, Springer

Nielsen M, Chuang I (2010) Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press

Svore K, Geller A, Troyer M, et al (2018) Q#: Enabling scalable quantum computing and development with a high-level dsl. In: Real World Domain Specific Languages Workshop (RWDSL 2018). ACM, New York, NY, USA, pp 7:1–7:10, https://doi.org/10.1145/3183895.3183901