Quantum Max Flow Analysis

Moreno Giussani

September 2, 2019

Abstract

This document will describe the analysis I have performed in the last months about the quantum implementation of the Max Flow algorithm. The max flow problem involves finding a feasible flow through a single-source, single-sink flow network that is maximum. I have been working for finding a practical solution to this problem using a quantum algorithm which could be at least as efficient as a classical one in a general case. I achieved in implementing a quantum algorithm based on Grover's algorithm which can solve the max flow problem, but its complexity is much worse than a classical algorithm.

Contents

1 Introduction

Before considering the algorithm, I have tried to understand most of the concepts which lies behind a quantum algorithm.

For what regards quantum computing, the standard model of computation is the quantum circuit. A quantum circuit is a scheme composed of some elementary blocks, which are qubits and quantum logic gates. Rows of this scheme represents qubits, while in columns are inserted quantum logic gates.

Sometimes, the tensor network model is used, especially in quantum physics papers.

1.1 Qubit

Qubits are the quantum equivalent of bits. They are usually represented using the bra-ket notation. A single qubit $|Q_0\rangle$is usually described by a 2-dimensional column vector (Ket notation) which is a specific linear combination of its orthonormal bases $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. When the qubit is measured (an equivalent operation of reading a bit value in classical computing), its value collapses to either $|0\rangle$ or $|1\rangle$ (their orthonormal bases).

Suppose $|Q_0\rangle$is defined as

$$|Q_0\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \;\; \alpha \in \mathbb{C}, \beta \in \mathbb{C}$$

Then α and β must respect the rule $|\alpha|^2 + |\beta|^2 = 1$, because $|\alpha|^2$ represents the probability that a measurement outputs $|0\rangle$ and $|\beta|^2$ represents the probability that a measurement outputs $|1\rangle$. During the computation the qubit can assume an "overlapped" state (both state 0 and state 1), but when measured, its expressivity power reduces to a classical bit. When both α and β are different from 0 ,Q_0 is said to be in superposition.

Qubits have also another interesting property: they cannot be copied. There is no way to create an identical copy of an arbitrary unknown quantum state (*no cloning theorem*).

Now, things get a bit tricky when considering a N-qubit quantum computer. If there are two or more qubits, their representation is made as the Kronecker product of all of the qbits, so they often cannot be considered as separated qubits. Suppose to have a 3-qubit quantum computer which uses qubits Q_a, Q_b, Q_c. The representation of the state of the quantum system becomes:

$$|Q_x\rangle = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \;\; x \in \{a, b, c\} \tag{1}$$

$$|Q_{ab}\rangle = |Q_a\rangle \otimes |Q_b\rangle = \begin{bmatrix} a_1 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ a_2 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_2b_1 \\ a_2b_2 \end{bmatrix} \tag{2}$$

$$|Q_{ab}\rangle = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_2b_1 \\ a_2b_2 \end{bmatrix} = a_1b_1(|0\rangle\otimes|0\rangle)+a_1b_2(|0\rangle\otimes|1\rangle)+a_2b_1(|1\rangle\otimes|0\rangle)+a_2b_2(|1\rangle\otimes|1\rangle) \tag{3}$$

$$|Q_{abc}\rangle = |Q_a\rangle \otimes |Q_b\rangle \otimes |Q_c\rangle = |Q_{ab}\rangle \otimes |Q_c\rangle = \begin{bmatrix} a_1b_1 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \\ a_1b_2 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \\ a_2b_1 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \\ a_2b_2 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1b_1c_1 \\ a_1b_1c_2 \\ a_1b_2c_1 \\ a_1b_2c_2 \\ a_2b_1c_1 \\ a_2b_1c_2 \\ a_2b_2c_1 \\ a_2b_2c_2 \end{bmatrix} \tag{4}$$

In many cases it is impossible to consider $Q_a, Q_b$and $Q_c$separately, because in a quantum system some quantum logic gates may cause to obtain a “mixed” state from which is not possible to find some suitable $Q_a, Q_b$and Q_c which satisfies (4). This concept, which is called *entanglement*, will be described in detail later. The quadratic sum of all elements of a Ket must be 1, like said before for a single qubit.

1.2 Quantum logic gates

In a quantum circuit model, quantum logic gates are transformation matrices which describes the behaviour of the physical quantum logic gates. Quantum logic gates are represented by means of unitary square matrices of size 2^n,where n is the number of qubits to which a gate can be applied. A matrix U is said unitary if

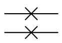
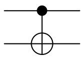
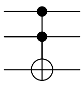
$$UU^\dagger = U^\dagger U = I$$

where $U^\dagger$ is the Hermitian transpose of U. The Hermitian transpose can be obtained by transposing U and then calculating the complex conjugate of all elements in U^T matrix.

The description of the state obtained from the application of a generic quantum gate $G$from quantum state$\left|S^0\right\rangle$ can be calculated as:

$$\left|S^1\right\rangle = G\left|S^0\right\rangle$$

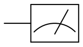
Some of the most known unitary logic gates are:

Name	Symbol	Matrix	Circuit
Hadamard	H	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$	$-\boxed{H}-$
Pauli-X (Not)	X(or NOT)	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$	$-\boxed{X}-$
Pauli-Y	Y	$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$	$-\boxed{Y}-$
Pauli-Z	Z or R_π	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$	$-\boxed{Z}-$
Swap	$SWAP$	$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$	
Controlled Not	$CNOT$	$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$	
Toffoli	$CCNOT$ or T	$\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}$	
Identity	I	$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$	$-$ or $-\boxed{I}-$

When applied to a single qubit in one of its bases ($|0\rangle$ or $|1\rangle$), an H gate will put the qubit in a superstate.

There are many more controlled gates which are represented using a C prefix, like for the CNOT gate. Their structure is

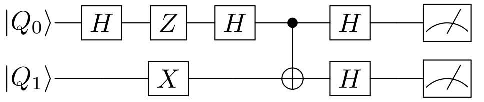
$$U = \begin{bmatrix} u_{11} & u_{12} & ... & u_{1m}\\ u_{21} & u_{22} & ... & u_{2m}\\ ... & ... & ... & ...\\ u_{m1} & u_{m2} & ... & u_{mm}\end{bmatrix} \quad CU = \begin{bmatrix} 1 & 0 & 0 & ... & 0\\ 0 & 1 & 0 & ... & 0\\ 0 & 0 & u_{11} & ... & u_{1m}\\ ... & ... & ... & ... & ...\\ 0 & 0 & u_{m1} & ... & u_{mm}\end{bmatrix}$$

The "measurement" operator has symbol 

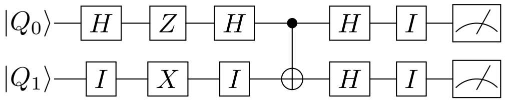
1.3 Quantum circuits

Quantum circuits can be represented via a ladder-like scheme. Each row (horizontal lines) represents a distinct qubit, and the gates which have to be applied to the given qubit are inserted on that line, from left to right. Gates on the same column has to be applied at the same time.

Here's an example with a 2-qubit circuit:



which is equivalent to the given circuit:



Identity gates columns will not change the state, they can be ignored, because

$$I_{n\,x\,n} \otimes I_{m\,x\,m} = I_{nm\,x\,nm}$$

As said before, in a multi qubit computer, considering $Q_0$and Q_1as independent qubits would often lead to mistakes, because the application of a gate to a qubit would cause some side effects on other qubits.

If not specified, like in this case, every qubit is conventionally initialized to state $|0\rangle$.

Knowing that the above circuit is a 2-qubit circuit, the initial state $\left|S^0\right\rangle$is described as $|Q_0\rangle \otimes |Q_1\rangle = |00\rangle$.

Then, the applied gate is

$$H \otimes I = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

.

So, the state of the circuit after the application of $H \otimes I$ is:

$$\left|S^1\right\rangle = (H \otimes I)\left|S^0\right\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

Now, simulating the whole execution:

$$\left|S^2\right\rangle = (Z \otimes X)\left|S^1\right\rangle = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\left|S^3\right\rangle = (H \otimes I)\left|S^2\right\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

$$\left|S^4\right\rangle = CNOT\left|S^3\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

$$\left|S^5\right\rangle = (H \otimes H)\left|S^4\right\rangle = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\left|S^6\right\rangle = (I \otimes I)\left|S^5\right\rangle = \left|S^5\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Then, the measurement would show $|01\rangle$as final result ($Q_0 = 0, Q_1 = 1$).

By definition, all the quantum gates that have been applied to this circuit are unitary. For this, by replacing recursively S^i, $0 < i < 6$ with S^{i-1} in the example circuit we obtain:

$$\left|S^6\right\rangle = (I \otimes I)(H \otimes H)CNOT(H \otimes I)(Z \otimes X)(H \otimes I)\left|S^0\right\rangle$$

So, if the circuit is unitary, then we could reverse the execution from the final state to the initial one. This property is called reversibility. In the example:

$$[(I \otimes I)(H \otimes H)CNOT(H \otimes I)(Z \otimes X)(H \otimes I)]^{-1}\left|S^6\right\rangle = \left|S^0\right\rangle$$

$$[(I \otimes I)(H \otimes H)CNOT(H \otimes I)(Z \otimes X)(H \otimes I)]^{\dagger}\left|S^6\right\rangle = \left|S^0\right\rangle$$

$$\left|S^0\right\rangle = (H \otimes I)^{\dagger}(Z \otimes X)^{\dagger}(H \otimes I)^{\dagger}CNOT^{\dagger}(H \otimes H)^{\dagger}(I \otimes I)^{\dagger}\left|S^6\right\rangle$$

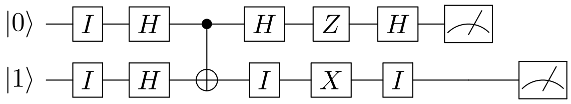
In our example, there are no complex gates, so

$$\left|S^0\right\rangle = (H \otimes I)'(Z \otimes X)'(H \otimes I)'CNOT'(H \otimes H)'(I \otimes I)'\left|S^6\right\rangle$$

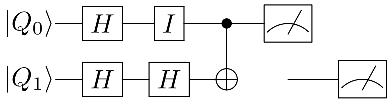
All the applied matrices are symmetrical, so

$$\left|S^0\right\rangle = (H \otimes I)(Z \otimes X)(H \otimes I)CNOT(H \otimes H)(I \otimes I)\left|S^6\right\rangle$$

This means that from every state of the system, we could rewind the execution flow. This property is called *reversible computing* and it is a property of every quantum computing algorithm. In this case, we can also build a circuit that from the final state S^6could reobtain the initial state S^0. The circuit that allows to do that is the original circuit flipped horizontally, that is:



Sometimes, considering qubits as independent from others would cause mistakes. Consider the following simple circuit:



$$\left|S^0\right\rangle = |00\rangle$$

$$\left|S^1\right\rangle = H \otimes H\left|S^0\right\rangle = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

$$\left|S^2\right\rangle = I \otimes H\left|S^1\right\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

$$\left|S^3\right\rangle = CNOT\left|S^2\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

In state S^3, the two qubits are no more independent from eachother. In state S^1, both $|Q_0\rangle$ and $|Q_1\rangle$ are in state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, then in S^2, $|Q_1\rangle =$

$|0\rangle$and $|Q_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. In S^3 it is nonsensical considering the two qubits individually. They can be both 0 or both 1 with equal probability. When two or more qubits are dependent on eachother, it is said that they are *entangled*. If we measure Q_0 as first, its measurement will influence Q_1's measurement, because if and only if Q_0 collapses to $|0\rangle$, then Q_1 will collapse to $|0\rangle$.

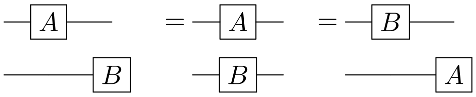
In the *CHSH game* is proven that when one of two entangled quantum particles (which are modeled as qubits in circuits) is measured, the other particle will assume the same state that the first particle had assumed when collapsing, at a faster-than-light speed.

1.4 Tensor Network Diagrams

Tensor network diagrams are models used to better represent complex entangled circuits, because they allow to define circuits with less parameters. They are usually lossy data compression methods used for describing the most important properties of the quantum state of a system. Quantum circuits are submodels of tensor network diagrams.

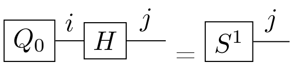
A tensor is usually a box, oval or triangle with some wires pointing up and down. Wires pointing up are called arms, while wires pointing down are called legs. Arms are upper indices while legs are lower indices. The number of arms and legs determines the underlying algebraic structure (scalar, vector or matrix in our context). They are often tilted by 90 degrees clockwise. Vectors are single arm tensors, while matrices has one arm and one leg.

Disconnected tensors are multiplied using the tensor product (Kronecker product in our case). Tensors can be freely moved past eachother (plannar deformation) without changing the semantic of the content they model, as in the example:



Unlike quantum circuits, wires can cross tensor symbols and other wires as long as the wire endpoints do not change.

Two tensors connected together means that they are multiplied (contracted or summed over). The following image



Is equivalent to

$$S^1 = Q_0 H$$

Tensor network diagrams were used in [1].

A more complete and better explanation can be found in [2].

2 The classical algorithm

Classical maximum flow algorithms are based of the Ford-Fulkerson method via Edmonds-Karp algorithm and Dinic's algorithm.

A maximum flow algorithm is an algorithm which attempts to find a feasible flow which is maximum.

2.1 Ford-Fulkerson method / Edmonds-Karp algorithm

Ford-Fulkerson method[3] requires as input:

- A network graph $G(V, E)$ where V is the vertices set and E is the edges set
- $\forall (u, v) \in E : \exists f(u, v)$ where $f(u, v)$ is the flow on the edge from vertex u to vertex v
- $\forall (u, v) \in E : \exists c(u, v)$ where $c(u, v)$ is the available capacity of the edge from vertex u to vertex v
- A source node $s \in V$ and a sink node $t \in V$
- Some constraints:
  - Capacity constraints: $\forall (u, v) \in E : f(u, v) \leq c(u, v)$
  - Skew symmetry: $\forall (u, v) \in E : f(u, v) = -f(v, u)$
  - Flow conservation: $\forall u \in V : u \neq s and u \neq t \implies \sum_{w \in V} f(u, w) = 0$
  - Source and sink flow constraint: $\sum_{(s,u) \in E} f(s, u) = \sum_{(v,t) \in E} f(v, t)$
- A residual network graph $G_f(V, E_f)$ with (residual) capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow. A residual graph has some important properties. If $f(u, v) > 0$ and $c(v, u) = 0$, then $c_f(v, u) = c(v, u) - f(u, v) = f(u, v) > 0$, so a possible flow can be present in a residual graph but not in the original graph G

Ford-Fulkerson method is composed of the following steps:

1. $\forall (u, v) \in E :\ f(u, v) = 0$
2. While there is a path $P \subseteq E : P \neq \emptyset$ from s to t in $G_f$with $c_f(u, v) > 0 \; \forall (u, v) \in P$
   (a) Find $c_f(P) = min\{c_f(u, v) : (u, v) \in P\}$
   (b) $\forall (u, v) \in P :$
      i. $f(u, v) = f(u, v) + c_f(P)$ (send the flow along the path)
      ii. $f(v, u) = f(v, u) - c_f(P)$ (reduce the flow in the "reversed" path)

Step 2 looks for an augmenting path (by using a search algorithm), then step 2.a finds the maximum admissible flow correction that can be done in path P , then step 2.b updates the flows.

Edmond-Karps algorithm applies the Ford-Fulkerson method by implementing a breadth-first search to find the path in step 2. By using breadth first, the complexity of the algorithm is $\mathcal{O}(VE^2)$

2.2 Dinitz's algorithm

Dinitz's algorithm (better known as Dinic's algorithm) is very similar to Edmond-Karps, it uses too a residual graph and looks for an augmenting path by using a level graph. A *level graph* is a graph that assigns to each vertex its distance to the root node. In some paper, it is called *layer graph.*

Dinitz's algorithm steps are:

1. Building the residual graph
2. Building the layer graph from the residual graph, by discarding edges connecting vertices at the same level.
3. All the paths in the layer graph contain augmenting paths. If there are no paths, the flow is maximum.
4. Update the flows and the residual graph
5. Go to step 2

3 Quantum Implementation

Before analyzing the proposed quantum implementations, there are some important details that has to be described about the actual state of the art of quantum computers.

For what regards the chosen quantum algorithm, I have considered many options. The most important specification that I have considered is that the quantum max flow algorithm implementation must work on a generic directed weighted graph, because classical computers can still provide excellent results for more specific cases.

3.1 Quantum Computers

Quantum computers at the actual state of the technology cannot provide a memory or either a "qRAM", because there is no way to reproduce an arbitrary unknown state (no cloning theorem) and there is no known way to store and load qubits (at least, not confirmed from reliable sources). All the qubits has to be allocated at the beginning of the computation and there is no "discard" option for a qubit.

At the moment, the most powerful quantum computer has more or less 70 qubits available (Google Bristlecone)

The whole circuit has to be available at compile time, and there is a unique flow of execution. Measurement cannot be used to change the behaviour of the quantum system at runtime.

Quantum states are very unstable, so this makes the process of creating a qRAM even more complex.

From a programmer's point of view, a quantum computer programming is very similar to FPGA programming.

Quantum computers have a structure very similar to vector machines and GPUs, but without any buffer or memory. For this reason, they have been successfully applied in neural networks, an application where GPUs excels at.

Quantum computers provides a probabilistic result, circuits are run many times and can be used for critical systems only if their results can be checked by classical computers or other more reliable methods.

Quantum gates has a reliability percentage, for this reason long circuits tend to fail in producing correct results.

In most quantum computers, not all qubits lines are interconnected. Most qubits are interconnected with 2 or more other qubits.

3.2 Tensor Networks

In [1], it is proposed to represent flow graphs as tensor networks, where the capacity of each link in the flow network is represented as the dimension of a Hilbert space. This implementation looked very complex, and also relies on qudits (quantum digits), so the required size of the circuit for representing a simple graph could explode.

A qudit is a quantum information unit with n orthonormal bases, and the superposition is between all n states. For n=2, a qudit is exactly equivalent to a qubit. Qudits may have more expressive power than qubits due to the availability of more superposition states.

Also, the paper does not provide an implementation of the actual circuit, which led me to look for other options.

3.3 Other attempts

I have thought about algorithms which could exploit at most as possible the quantum potential, instead of "porting" classical algorithms to quantum. The first critical problems which arised were:

1. How to represent graphs?

2. How to represent weights?

How to represent graphs I tried to find a smart way to represent graphs in quantum circuits. I took inspiration from [1] in which they attempt to represent a graph as a tensor network. Knowing that quantum circuits are submodels of tensor networks, I attempted to represent a simple tree architecture (without weights) in the circuit. I used qubits as nodes and entanglement to represent edges between nodes. I accomplished in representing a single simple path without repeating nodes (which is indeed a trivial problem), which would have been either taken or not taken, but I failed in representing more complex structures, due to the fact that i haven't been able to entangle 3 qubits together. This attempt seemed promising to me because with entanglement i would have obtained the flow constraint, so that if a node is "taken", then another following node should be taken too. If we consider for example the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, if the first qubit is taken, then the following one would be taken too or none of those would be taken.

How to represent weights In the upper representation there is no weight representation. Weights are hard to be represented in a quantum environment, especially because comparisons in quantum circuits (eg toffoli) are hard to implement and cause an explosion in size of the circuit. What I have considered is to represent only the layer graph built on the residual graph in the quantum circuit. Then, I would have used the quantum circuit to find a path in the classical residual graph to obtain a (small) speedup.

3.4 Quantum Edmond-Karps Algorithm

In [5] is shown an approach for building a quantum max flow algorithm, which is better explained in [4]. They propose to build a layered subgraph (also called level subgraph, the same that is used in Dinitz's algorithm). This is the version that iI have implemented. The procedure is:

1. Set the level l to ∞ to all nodes except the source s, whose level is set to 0.

2. Create a one entry queue Q =$\{s\}$

3. While $Q \neq \emptyset$

    (a) Take the first node n from Q

    (b) Find by Grover's search all its neighbors y with $l(y) = \infty$, set $l(y) = l(n) + 1$, and append y into Q

    (c) Remove n from Q

But there is a problem with this algorithm, at step 3.b. It can become clear after Grover's algorithm is being explained.

3.4.1 Grover's algorithm

Before defining what does Grover's algorithm , it is necessary defining what is a quantum oracle.

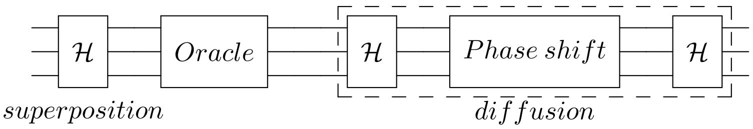
A *quantum oracle* of a function $f(C)$ is a quantum circuit which given an input S, it outputs $|1\rangle$ on a specific qubit if $S \in f(C)$, $|0\rangle$otherwise. Grover's algorithm has complexity $O(\sqrt{N})$ times the oracle complexity.

Grover's algorithm [6](sometimes called Grover's search algorithm) is a quantum algorithm which, given as input a oracle of $f(C)$ and the expected output of the function S, it attempts to find an input $C \in f^{-1}(S)$ which could have generated S. This has some important consequences in cryptography, because many algorithms for cryptography could possibly be broken if a quantum representation of the cryptographic one way function is provided.

Originally, Grover's algorithm was called Grover's search, because it allows to do a search in an unsorted database of possible inputs. But knowing that building (classically) such a oracle for a unsorted database requires at least linear complexity, it is not suggested to be used practically for searches. This is the main reason which blocks the provided quantum algorithm to outcome the classical one.

Grover's algorithm outputs one result per run. It could be scaled both horizontally (another quantum computer running the same circuit) and vertically (double the quantum circuit, like in a distinct quantum computer).

Grover's circuit is composed of four parts, superposition initialization, oracle, phase shift and diffusion.



Of those parts, the oracle and the diffusion part are iterated many times. The number of times that they have to be repeated has to be set as $\frac{\pi}{4}\sqrt{2^n}$, where n is the number of qubits used ($2^n$is the size of the "database"). The problem is that $\frac{\pi}{4}\sqrt{2^n}$ is often not an integer. Rounding the number of iterations could cause to not find a solution even when it exists.

> "Grover's quantum searching technique is like cooking a soufflé. You put the state obtained by quantum parallelism in a "quantum oven" and let the desired answer rise slowly. Success is almost guaranteed if you open the oven at just the right time. But the soufflé is very likely to fall—the amplitude of the correct answer drops to zero—if you open the oven too early." – Kristen Fuchs [6]

3.4.2 Finding neighbors by Grover's algorithm

> 3.b) Find by Grover's search all its neighbors y with $l(y) = \infty$, set $l(y) = l(n) + 1$, and append y into Q

This is the key point. Grover's algorithm for searches returns one result per run, which is chosen casually between all the possible results. Finding all the infinite-distanced neighbors of a single node is a task that can be done in $O(\frac{\pi}{4}\sqrt{2^{\lceil\log_2 \bar{b}\rceil}}\bar{b}) \implies O(\bar{b}\sqrt{\bar{b}})$, where $\bar{b}$ is the average number of neighbors in the graph ("branching factor").

The explanation behind $O(\frac{\pi}{4}\sqrt{2^{\lceil\log_2 \bar{b}\rceil}}\bar{b})$ lies in the fact that the algorithm has to be rerun at least $\bar{b}$ (neighboring nodes) time to find all infinite distance neighbors, and each run has to do $\frac{\pi}{4}\sqrt{2^{\lceil\log_2 \bar{b}\rceil}}$ iterations, where $2^{\lceil\log_2 \bar{b}\rceil}$ is the size of the database ("search space"). The number of database entries has to be a power of 2. The assumption behind this estimation is that the best "lucky" case is considered, no unexpected outcomes are considered.

The overall quantum complexity is then

$$O(\bar{b}^{\frac{3}{2}}N)$$

where $N$is the number of nodes in the graph. This complexity can be reduced heavily depending on the topology of the network. Considering the average branching factor as $\bar{b} = \frac{E}{N}$ where E is the number of edges in the network flow graph, we get:

$$O(\frac{E^{\frac{3}{2}}}{\sqrt{N}})$$

This complexity has to be multiplied with the complexity of the oracle. A boolean oracle (the only one which could work in a general network flow graph) has almost constant complexity. The problem is that the oracle has to be built classically. This can be done in $O(\bar{b}) = O(\frac{E}{N})$. Unfortunately, when a node is found, a layer is assigned to it, the database changes and so the oracle has to be changed (recomputed classically) too. Each element can be found at most once (once it is found, it gets its layer number updated), so the overall complexity (both classical and quantum) is:

$$O(\frac{E^{\frac{3}{2}}}{\sqrt{N}} \cdot \frac{E}{N} \cdot N) = O(\frac{E^{\frac{5}{2}}}{\sqrt{N}})$$

This result is still optimistic, because there is a strong assumption behind this estimation. Quantum unexpected outcomes happens very often, especially in large sparse search spaces. This is due to the intrinsic nature of quantum computers and it is due to the problem of having many values to be searched. Quantum unexpected outcomes happen because the probability of some values which shouldn't be found is not exactly zero, and tend to increase as the size of the search space increases.

Suppose that we have a dataset of 200 elements where 50 elements are being searched. Then, we would have an 8 qubit quantum circuit in which 56 entries are out of bound, 100 elements have 1% each chance to be extracted and 100 elements shouldn't be found. Noise in a quantum algorithm can substantially increase the chances to outcome erroneous values, also because each iteration in Grover's algorithm amplifies the most significant possible outcomes, which could be an erroneous one (noise).

The complexity considered in [4] is based on an “adjacency model”. The adjacency model I suppose is based on an adjacency matrix, which could used to store information about the network.

Implementing such a matrix in a quantum circuit would cause to have huge circuits. Let’s start by a graph without network capacities. Then we would have a circuit with a size of N^2 qubits. Actually, this would be unfeasible for most networks due to the fact that the most powerful quantum computer can work on at most 100 qubits.

3.4.3 Building the oracle

Building the oracle requires to know where the elements are in the classical dataset. This means that to know where they are, a linear search (if the set is unordered) has to be performed. Once the elements have been found, it is nonsensical insert them into an oracle and then let the quantum algorithm find the values that have been inserted in the oracle, because we still know what we have to search. It could have sense only for testing how a quantum computer behaves, so this is the reason why I have kept writing the implementation.

3.5 Actual implementation

The actual implementation sticks as most as possible to the quantum Edmond-Karps Algorithm version. The algorithm must have a classical part and a quantum part which must work together. The classical part is written in C#, while the quantum part is written in Q# and they work together by using Microsoft Quantum Development Kit.

The quantum implementation uses Grover’s algorithm to find neighbors whose distance is infinite. The Grover’s algorithm implementation is the one provided with the quantum templates of Microsoft QDK. Neighbors are represented as indices, which has to be found in the implicit database. An *implicit database* is a database in which all the possible elements are present. The elements which has to be found are marked. Non-marked elements should never be outcomed.

To improve slightly the algorithm’s performance, the algorithm stops layering the graph once the sink is found. It also excludes the nodes whose flow cannot be increased from the layering phase.

Then, from the sink, the path is rewinded to the source (it is a spanning tree) and then the flow is augmented on the path. The algorithm stops when the layered graph cannot include the sink.

It also includes a random graph generator for generating samples. To provide a graphical output of the result, the graph is being outputted as .dot and .gexf. The README.md file contains the technical details for building and testing the program.

On Linux, a .png file is also generated from the .dot.

The program outputs and saves to file also the number of unexpected outcomes found during the execution.

Project structure The project is composed of 3 C# files, a .csproj configuration XML file and a Q# file, that is the Microsoft Quantum's Grover Algorithm implementation.

The "main" C# file is Driver.cs, which contains two classes, Utilities and Driver.

Utilities contains code for generating a graph schema, while Driver contains the main function and the code that calls the quantum Edmonds-Karp algorithm.

Graph.cs contains the code for generating and handling graphs. It has 3 classes, Edge, Graph and Node.

Graph.addLayer is the layering function, while Graph.findInnerPath is the recursive function that looks for an augmenting path. The quantum function is Graph.findLayerNeighbors, which is used both to find infinite distanced neighbors (infinite value is stored as -1) and to find the augmenting path.

4 Programming environment

The chosen programming enviroment for the algorithm implementation is Microsoft Quantum Development Kit (QDK) on Visual Studio Code.

Visual Studio Code is one of the most popular development environments, due to its simple design. It is composed of a simple but powerful lightweight editor, with many extensions available for download. This allows the developer to install packages as soon as they are required, which is a great advantage. It offers an intuitive UI similar to Atom IDE but much lighter and reactive.

QDK requires C# code for the classical parts and Q# (which is similar to F# and C#) code for the quantum parts. The QDK has many useful libraries, a compiler and a simulator which could be run very easily from Visual Studio Code.

The QDK is still under development, it is updated with a new version almost monthly but updates often break older versions' code. It is based on Microsoft .Net and it is cross-platform.

QDK is documented, but the documentation is sparse in many Microsoft websites, and not all articles are updated. Anyway, the Microsoft Quantum team is always available on Github and updates the outdated documentation as soon as they have been notified of.

There are also many Microsoft based examples/libraries, such as an Grover's algorithm implementation which i used in the implemented project.

There are also some Q# community integrations which allow to integrate the Q# code to other quantum languages, such as Qiskit and OpenQasm.

Conclusion

The quantum algorithm implementation does not show an advantage with respect to a classical algorithm such as Dinitz's or Edmonds-Karp. The quantum implementation can be useful to see how often unexpected outcomes do happen in a much simplified quantum algorithm (the quantum oracle implementation is the simplest possible). The implemented algorithm is the implementation of the analysed algorithm described on [5]. It is very inefficient, due to the complexity required to encode a quantum oracle and the scarcity of qubits in quantum computers (storing a complete graph in a quantum oracle requires too many qubits). To obtain a quantum speedup, it is required to not convert classical algorithms into quantum ones, because classical computation has different properties with respect to quantum computation, because operations which are simple in classical computing (such as comparing two bits) can be very complex in quantum and viceversa. Exploiting quantum circuits' capabilities is a very challenging task, due to the logic behind the quantum gates and the qubits' properties.

References

[1] Shawn X. Cui, Michael H. Freedman, Or Sattath, Richard Stong: “Quantum Max-flow/Min-cut”, J. Math. Phys. 57, 062206 (2016); arXiv:1508.04644, 2015. 1.4, 3.2, 3.3

[2] Jacob Biamonte: “Tensor Networks in a Nutshell”, arXiv:1708.00006, 2017. 1.4

[3] Lester R. Ford Jr. , Delbert R. Fulkerson: “Maximal flow through a network” , Canadian Journal of Mathematics 8: 399–404, 1956. 2.1

[4] Andris Ambainis: “Quantum Algorithms for Matching and Network Flows”, arXiv:quant-ph/0508205, 2005. 3.4, 3.4.2

[5] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan Mniszewski, Balu Nadiga, Dan O’Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Phil Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard Zamora: “Quantum Algorithm Implementations for Beginners”, arXiv:1804.03719, 2018. 3.4, 4

[6] Colin P. Williams: “Explorations in Quantum Computing”, Springer-Verlag London, second edition, p 241-261, 2011.

3.4.1