Report on Quantum Computing exploratory research

Moreno Giussani, Samuele Pino

Project for Software Engineering 2, Politecnico di Milano

June 15, 2019

Contents

Chapter 1

Introduction

1.1 Abstract

1.2 Quantum circuits

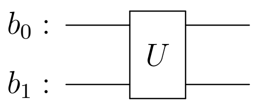
We assume the reader already knows the basic concepts about linear algebra and elementary quantum gates. For more information on these basic topics, we recommend [10, 15].

Here we propose a fast walk-through of some possible compositions of quantum gates in the context of a circuit. For further explanations and a more complete reading on the topic we recommend [15, p. 123–129].

1.2.1 Matrix representation of circuits

Operations that make sense in quantum computing are usually performed on more than 2 or 3 qubits and they often give as an output multiple qubits as well. Such computations can be performed by long and complex circuits, therefore we need to be able to decompose them into a sequence of simpler quantum gates. A circuit can be represented by a unitary matrix, which can mathematically describe the operations performed on an array of input qubits.

Consider a qubit array $|b\rangle = [b_0, b_1]^T$ where b_0 and b_1 are respectively the most and least significative qubits. Therefore a unitary matrix U applied on a $|b\rangle$ will have the following representation:



$$U|b\rangle = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

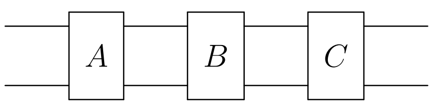
where on the right side we have a standard matrix multiplication. Please note that the order of this product is important, as it is *not commutative*.

1.2.2 Some elementary gates

A computation usually requires a combination of elementary gates in 3 main ways: sequentially, in parallel or conditionally. In Table 1.1 we show some gates (and their matrix form) that will be useful for the rest of this document.

1.2.3 Quantum gates in series

The series of quantum gates applied on a qubit line (or on a subset of qubit lines) in a circuit is equivalent to the *dot product* between the matrices of each gate in reverse order.



$$C \cdot B \cdot A$$

Matrices A, B and C must be of the same size (in this example, being applied on 2 bits, they must be 4×4). The result matrix will be obviously of the same size of A, B and C (4×4).

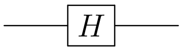
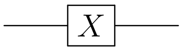
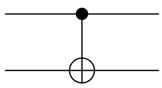
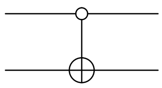
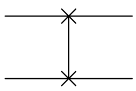
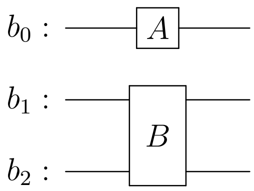
Hadamard gate		$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
NOT gate		$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
CNOT (on $b_0 = 1$)		$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$
CNOT (on $b_0 = 0$)		$\begin{bmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$
SWAP gate		$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$

Table 1.1: Some elementary quantum gates in circuit and matrix representation.

1.2.4 Quantum gates in parallel

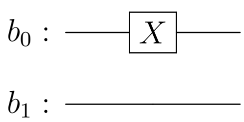
Applying distinct quantum gates to disjointed subsets of qubits is equivalent to the *direct product* (or *tensor product*, or *kronecker product*) between the matrices of each gate. Here the order is given by the position of the qubits to which gates are applied (most significative first).



$$A^{(2\times2)} \otimes B^{(4\times4)} = U^{(8\times8)}$$

Given $A \in M^{m\times m}$ and $B \in M^{n\times n}$, the result matrix will be $mn \times mn$ dimensional. We can easily notice that the matrix dimension grows fast with consecutive applications of direct product and the resulting dimension is always a power of 2 in quantum circuits.

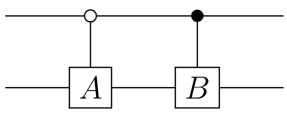
A special case is when some qubits have a gate applied, while others have nothing (that is equivalent to an identity matrix).



$$X \otimes I_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

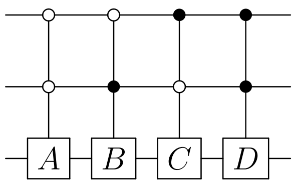
1.2.5 Controlled gates

The effect of a gate on a subset of qubits (“targets”) can be applied conditionally to the value of one or more other qubits (called “controls”). This operation is equivalent to a *direct sum* between matrices.



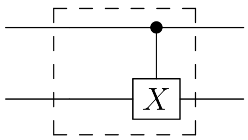
$$A \oplus B = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & b_{11} & b_{12} \\ 0 & 0 & b_{21} & b_{22} \end{bmatrix}$$

in this example $A \oplus B$ means that gate A is applied to the bottom qubit if the top one is in state $|0\rangle$, while B is applied if the top qubit is in state $|1\rangle$. It can be easily generalized to the case of 2 control qubits:

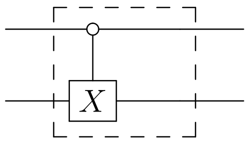


$$A \oplus B \oplus C \oplus D = \begin{bmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & 0 \\ 0 & 0 & 0 & D \end{bmatrix}$$

in general if we have n_c control lines and n_t target lines, we can obtain a direct sum of up to 2^{n_c} gates, each gate of dimension 2^{n_t}. If we have less than 2^{n_c} gates to control, the missing spots in the direct sum are filled by appropriate sized identity matrices:



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



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

Chapter 2

Microsoft Quantum Development Kit

QDK is Microsoft's open source developement kit for quantum computing. It is quite young, as it has been released in January 2018. Despite that, it has some interesting characteristics, like the use of a specific "quantum focused" language: Q#. Differently from other frameworks (like pyQuil, QISKit, ProjectQ...), QDK uses a technology based on Majorana fermions, which is also the reason why right now there are no hardware devices on which run the algorithms. [11]

It is updated very frequently with often drastic improvement in terms of usability and bug fixes. Of course it is not yet a mature enviroment, as it is still under development.

2.1 Software overview

2.1.1 Installation

Microsoft QDK can be installed on top of Visual Studio or Visual Studio Code (recommended). The setup is easy and the process it's well explained in the official page: `https://docs.microsoft.com/it-it/quantum/install-guide/vs-2017`.

After installation it is also possible to validate its correctness running a sample program whose aim is to check the possible absence of needed packages (linke NuGet).

2.1.2 Documentation

A complete documentation of the software and language can be found on the official website `https://docs.microsoft.com/it-it/quantum`. It contains tutorials on how to run the first quantum program, info about the simulator, Q# syntax and its libraries. Other part of the website also contain a good documentation on the theory about quantum computing.

Moreover the open source libraries are useful to learn the language. Last but not least, there is a vast number of verbosely commented examples (Teleportation, Grover's Search, Integer factorization, simulations...).

2.1.3 Language

Using QDK requires some basic knowledge of C# for the classical host computation, usually contained in a "driver.cs" file, that is used to call the simulator with the quantum program, optionally providing inputs.

The quantum part of the program uses Q# that, despite the name, is more similar to a hardware description language than to an object oriented one. We can define *operations*, callable routines with quantum instructions, that as functions take some input and return an output value. We can also define variables to values bindings (like integers and booleans), perform operations on single qubits (like gates, conditionals and controls).

In general the language is high level oriented: you do not have to design the spatial disposition of gates on qubit lines as you are not bound to a specific architecture, therefore the programmer can focus more on the algorithm than on the implementation details, thanks also to the available libraries.

2.1.4 Simulator

QDK can be used within a local run of Visual Studio, in this case it can simulate circuits of up to 30 qubits. If more power is needed, it can also be run in Microsoft Azure cloud (through a pais subscription) achieving simulations of more than 40 qubits.

It uses a locally deployed simulation environment based on dotnet. The language abstracts from the actual architecture to be deployed (it uses Qubit objects, not specific low level registers), in order to allow an better reusability and portability of the code.

It also implements a Toffoli simulator, a special-purpose simulator for quantum algorithms that are limited to X, CNOT, and multi-controlled X.

A trace simulator is also provided. It is useful for debugging classical code and estimating the resources required to run a given instance of a quantum program. Circuits of thousands of qubits can be tested, as the trace simulator executes the program without simulating the state of the quantum computer.

Hardware and noise analysis

The technology Microsoft is trying to use has not been implemented on hardware yet. Moreover QDK does not provide any functionality for noise analysis or simulation. This is probably connected to the fact that Microsoft is betting on topological qubits, that should be highly resilient to noise and decoherence.

2.2 Example: Grover Search implementation

Chapter 3

Insights on Grover Search Algorithm and its implementation

3.1 Introduction

3.1.1 Premise

Many quantum algorithms base their reason to exist in the fact that a small routine of the them (e.g. the search for the next arc to be considered among those coming out of a given node, in Max Flow Analysis) is done by a quantum computer. It is usually nothing more than a search in a list (or more generally in a database) of one or more elements that satisfy a certain condition (the arcs that have not been visited yet, i.e. having infinite weight value).

3.1.2 The problem

Let's start noticing that although we have presented in Section 2.2 an implementation of Grover Search in Q#, it actually works on what is known as "virtual database". Alike real databases, virtual (or implicit) ones are not really databases: given n as the number of bits, they are nothing more than the set of integer numbers $[0, 2^{n-1}]$.

Such a "database" can be easily implemented with a quantum register initialized with $H^{\oplus n}$. In this way, whenever the register is measured, it collapses to one of all the possible combinations of its bits (i.e. $[0, 2^{n-1}]$), being all these combinations all equally probable.

Actually the implementation described in Section 2.2 complies with most of the available literature [9, 12]. It is evident that currently most of the works someway related to Grover Search Algorithm are devoted to quantum search on virtual databases. [7]

Apparently some people agree that Grover is limited to implicit databases, therefore not convenient or even not useful at all for real databases [14, 16, 3, 1, 2]. On the other hand, someone had a deeper study on the algorithm, understanding the mechanism and implementing (at least mathematically) the encoding and the search on a real database. [5]

3.2 The phone book implementation

The work [5] actually finds a way to encode some elements into a real database. It is done setting a register to an entangled state, as sum of the states corresponding to the elements that we want to encode into the database. This database-register is created by applying a particular matrix

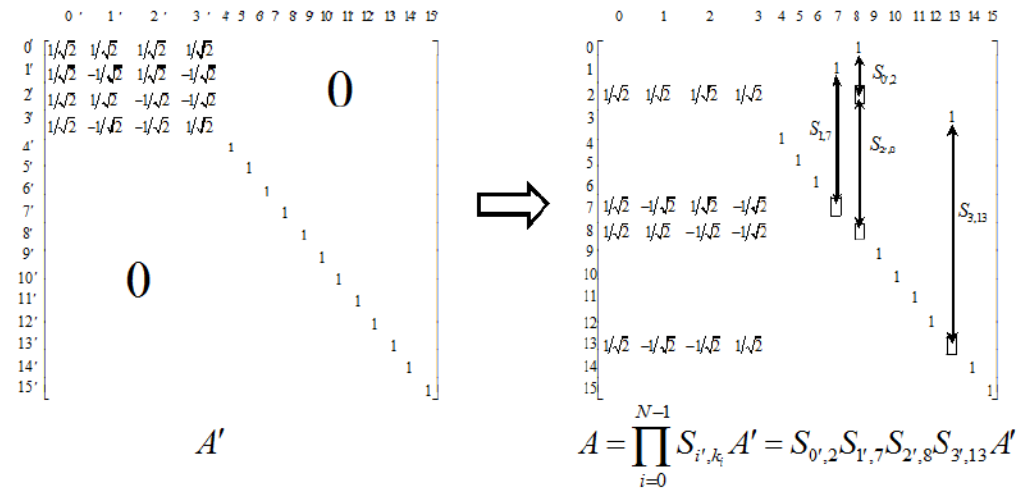


Figure 3.1: Successive row swapping operations to transform A' to A in for the specific telephone database example. (credits to [5])

to it, in which the database elements are encoded within the rows order. An example is shown in Figure 3.1.

Calling n the number of bits of the primary key (i.e. the contact *name*) and m the number of bits of the data field (i.e. the contact *number*), the square matrix A' will have a size of $K = 2^n \cdot 2^m = 2^{n+m}$ rows. Matrix A' can be obtained as a direct sum $H^{\otimes n} \oplus I_{K-2^n}$ (see Section 1.2.5).

Matrix A can be obtained by applying to A' a series of swap operators S_{ij} that perform a swapping between rows i and j of a matrix. This operation is not described in the proposed paper, therefore we will try to give an algorithm to perform it (see Section 3.3). This is the key passage that let us prepare an entangled register, ready to be used for the subsequent Grover iterations, as shown in the mentioned work.

This is a remarkable result, as it demonstrates the theoretical consistency of Grover's Algorithm for searching purposes. Critics can be raised against the performance or the convenience of the entire process with respect to the classical one, but these topics have already been discussed elsewhere [14].

3.3 Permutation of rows in a matrix

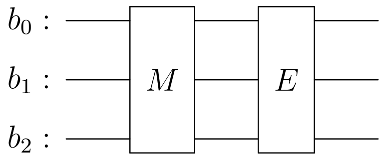
The main issue that we want to address now is how to perform an arbitrary permutation of the rows of a "quantum" matrix. This is a fundamental algorithm passage for the correct implementation of Grover iteration. Despite that, we were not able to find any hint in literature on how to perform such permutations, so here we present some ideas that can be a starting point for a future improved and more general solution of the problem.

As it is well known in linear algebra, given a square matrix M we can obtain M' (a version of it where i-th and j-th rows are swapped) multiplying M by a matrix E, where E is the identity with i-th and j-th rows swapped:

$$EM = M'$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{41} & a_{42} & a_{43} & a_{44} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

Therefore our problem is to find a circuit that implements matrix E. This technique is consistent with the fact that a circuit applied on an array of qubit can be represented with a matrix multiplying the vector from the left side. Multiplying E from the left of M is equivalent to placing the circuit of E after (on the right of) the circuit of M.



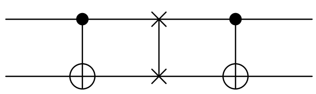
3.3.1 Simplified case: 2 qubits

If we have a circuit of only 2 qubits, swapping 2 rows can be relatively easy.

Rows to swap	Matrices to apply
1, 2	ICNOT (*NOT gate* controlled on first qubit = 0)
1, 3	SWAP · ICNOT · SWAP
1, 4	CNOT · SWAP · ICNOT · SWAP · CNOT
2, 3	SWAP
2, 4	CNOT · SWAP · CNOT
3, 4	CNOT

For example:

$$CNOT \cdot SWAP \cdot CNOT = swap(2,4)$$



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

3.3.2 General case: shift and control

A possible extension of the algorithm to the case of n qubits would make an intensive use of controlled gates. Control through a qubit equal to 1 is equivalent to the direct sum of an identity matrix and the controlled gate itself. This means that, if we can control a gate, we are able to replicate the behavior of its 4×4 matrix in the bottom half of a larger 8×8 matrix (in terms of rows swapping). It could be interesting if we could temporarily "move down" the rows of a big matrix, perform our swaps and then move them up again. To easier describe this process we will make some definitions.

Definition. Let N denote the number of rows in the matrix representing an operation G. The number N is obviously 2^n, where n is the number of bits on which gate G is applied. Let's consider only a gate G which is decomposable as a direct product of a matrix M and an identity matrix I_k. We will call k the *grade* of operator G.

Example:

- CNOT and SWAP have *grade* 0
- CNOT3, SWAP3 and SHIFT3 (Section 4.3) have *grade* 1
- SHIFT4 (Section 4.3) has *grade* 2
- CCNOT has grade 0

We can easily increase by h the degree of a matrix G by performing $kron(G, I_h)$. This is equivalent, in a register large $n + h$ qubits, to apply G to the first n qubits and nothing to the remaining h.

Algorithm

Let's take as an example the problem of permuting rows of an 8×8 matrix.

Using CNOT3, SWAP3 and ICNOT3 we can exchange matrix macroblocks (blocks of 2 contiguous lines). We can then operate on the two blocks (each 2x8) of the lower half-matrix using the CCNOT, CSWAP and controlled ICNOT ports, with granularity of the individual rows. Please note that the CSWAP allows us to exchange two rows of two different blocks (2x8), this can be useful in the generalization to more qubits. The same algorithm can be used for 16x16 matrices, increasing by one the degree of all previous ports and adding one more bit of control to the existing port (thus obtaining CCCNOT, CCICNOT, CCSWAP...).

Useful gates to perform the shift are SHIFT, QSD and QSU gates, together with their higher grade versions (Section 4.3).

Open issues

Probably the addition of new control qubits at each step of generalization implies an exponential growth in spatial complexity of the circuit.

3.3.3 General case: sorting algorithms

This approach, instead of focusing on single row swaps, treats the permutation from initial to final matrix as a single process. You can see the analogy with sorting algorithms applied to arrays (bubble sort, merge sort...).

If there was a way to swap 2 consecutive rows of a matrix, regardless of their position, the problem would be easily solved. In that case we could apply bubble sort as an algorithm to rearrange all the rows as we like.

The only general way that we could devise to exchange consecutive rows is to use X gates in direct sum with I_2 matrices all over the diagonal. The main drawback of this method is that this configuration is only able to swap rows $2i + 1$ and $2i + 2$, with $i \geq 0$. Therefore if we want to swap for example rows 2 and 3 we need to use a SWAP gate in direct sum with an appropriate number of I matrices down the diagonal. The problem in using SWAP in this configuration is that it works only for swapping rows $4i + 2$ and $4i + 3$, with $i \geq 0$. Therefore if we want to swap rows 4 and 5 we need a new different gate (possibly 8×8 or bigger) and so on.

Chapter 4

Quantum gates in Octave

Octave is a free software and a scientific programming language whose syntax is largely compatible with Matlab.

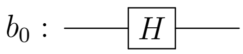
To fill the gap between some theoretical papers (which perform calculations on matrices) and quantum gates (that are eventually how those matrices are implemented) we modeled some quantum matrices as combination of known gates. In this way it was possible to investigate on how such matrices could be really implemented. Moreover some of these gates are referred in other chapters of this document, so this chapter has also the purpose of being in some way an appendix.

4.1 Elementary (existing) gates

4.1.1 Hadamard and X, Y, Z

We will show only H as an example, but the same applies for X, Y and Z and in general for 2×2 gates.

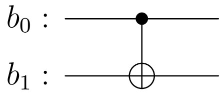
Circuit	Octave code



```
H = [
1,  1;
1,  -1
]./sqrt(2);
```

4.1.2 CNOT

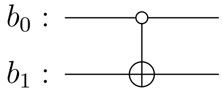
Circuit	Octave code



```
CNOT = C(X);
```

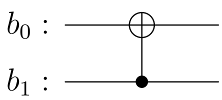
4.1.3 ICNOT: inverted CNOT

Circuit	Octave code



```
ICNOT = IC(X);
```

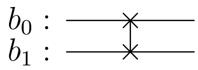
Note that **it is not equivalent** to this circuit:



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

4.1.4 SWAP

Circuit | Octave code

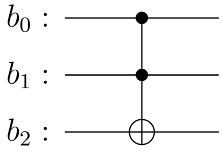



```
SWAP = [
1, 0, 0, 0;
0, 0, 1, 0;
0, 1, 0, 0;
0, 0, 0, 1;
];
```

4.1.5 CCNOT

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

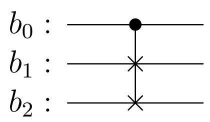
Circuit | Octave code




```
CCNOT = C(CNOT);
```

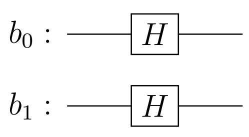
4.1.6 CSWAP

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

Circuit	Octave code
	`CSWAP = C(SWAP);`

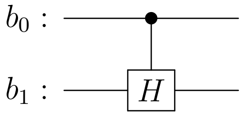
4.2 Operations between gates

4.2.1 Kronecker product (or direct product)

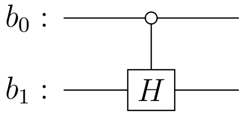
Circuit	Octave code
	`kron(H, H);`

4.2.2 Gate control (direct sum)

Circuit	Octave code




```
function out = C(gate)
size = rows(gate);
out = blkdiag(eye(size), gate);
endfunction;
```




```
function out = IC(gate)
size = rows(gate);
out = blkdiag(gate, eye(size));
endfunction;
```

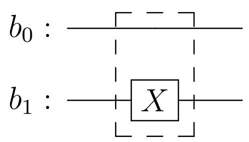
Note that CNOT is a “controlled X”.

4.3 New (derivated) gates

4.3.1 DSWAP

Performs a swap of the first 2 and the last 2 rows of the matrix, i.e. flips the least significative qubit.

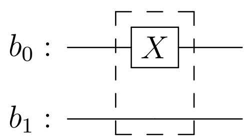
$$DSWAP = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Circuit	Octave code
	`DSWAP = kron(eye(2), X);`

4.3.2 SHIFT

This gate shifts rows of half the size of the matrix, i.e. flips the most significative qubit.

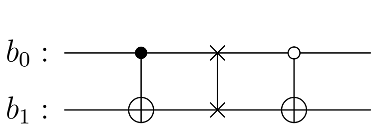
$$SHIFT = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Circuit	Octave code
	`DSWAP = kron(X, eye(2));`

4.3.3 QSD: Quarter Shift Down

This gate shifts rows down of a quarter the matrix.

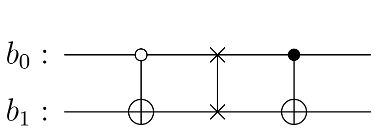
$$QSD = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Circuit	Octave code
	`QSD = ICNOT*SWAP*CNOT;`

4.3.4 QSU: Quarter Shift Up

This gate shifts rows down of a quarter the matrix.

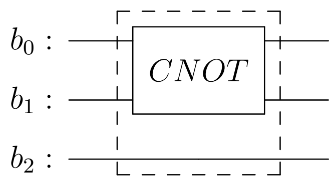
$$QSU = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Circuit	Octave code
	`QSU = CNOT*SWAP*ICNOT;`

4.3.5 CNOT3

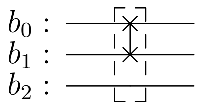
CNOT of grade 1. Please note that it is different from CCNOT.

$$CNOT3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Circuit	Octave code
	`CNOT3 = kron(CNOT, eye(2));`

4.3.6 SWAP3

$$SWAP3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Circuit	Octave code
	`SWAP3 = kron(SWAP, eye(2));`

4.3.7 SHIFT3

$$SHIFT3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

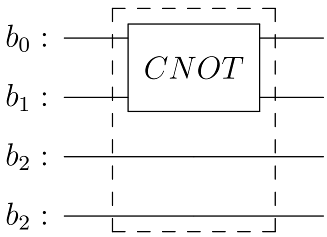
Octave code

```
SHIFT3 = kron(SHIFT, eye(2));
```

4.3.8 CNOT4

CNOT of grade 2. Please note that it is different from CCCNOT.

$$CNOT4 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Circuit	Octave code
	`CNOT4 = kron(CNOT, eye(4));`

Chapter 5

Other ways explored during the research

Grover's Algorithm was originally devised to work with functions that are satisfied by a single input. Actually it has also been "generalized to search in the presence of multiple *winners*". [6]

It turns out that Grover's Algorithm can be more useful in "speeding up the solution to NP-complete problems such as 3-SAT" than actual search. [13]

We also considered spatial database search as a possible way of exploiting Grover's Algorithm, in the specific case of graphs with costs on arcs [4, 8].

Although there are some points of contact with Grover, none of them seemed to me of any use for our Max Flow Analysis problem.

Chapter 6

Conclusions

Bibliography

[1] Grover's algorithm: what to input to oracle? `https://quantumcomputing.stackexchange.com/questions/2149/grovers-algorithm-what-to-input-to-oracle`, 2018.

[2] Grover's algorithm: where is the list? `https://quantumcomputing.stackexchange.com/questions/2110/grovers-algorithm-where-is-the-list`, 2018.

[3] How is the oracle in grover's search algorithm implemented? `https://quantumcomputing.stackexchange.com/questions/175/how-is-the-oracle-in-grovers-search-algorithm-implemented`, 2018.

[4] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 50–59, New York, NY, USA, 2001. ACM.

[5] P. Alsing and N. McDonald. Grover's search algorithm with an entangled database state. *Proceedings of SPIE - The International Society for Optical Engineering*, 05 2011.

[6] M. Boyer, G. Brassard, P. Hyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(45):493–505, 1998.

[7] B. Broda. Quantum search of a real unstructured database. *The European Physical Journal Plus*, 131(2):38, Feb 2016.

[8] A. M. Childs and J. Goldstone. Spatial search by quantum walk. *Physical Review A*, 70(2):022314, 2004.

[9] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.

[10] A. Helwer. Quantum computing for computer scientists. Available at `https://www.microsoft.com/en-us/research/video/quantum-computing-computer-scientists/`, 2018.

[11] R. LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019.

[12] C. Lavor, L. Manssur, and R. Portugal. Grover's algorithm: quantum database search. *arXiv preprint quant-ph/0301079*, 2003.

[13] A. Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[14] G. F. Viamontes, I. L. Markov, and J. P. Hayes. Is quantum search practical? *Computing in Science Engineering*, 7(3):62–70, May 2005.

[15] C. P. Williams. *Explorations in Quantum Computing*. Texts in Computer Science. Springer, 2nd edition, 2011.

[16] C. Zalka. Using grover's quantum algorithm for searching actual databases. *Physical Review A*, 62:52305, 10 2000.