Analysis of Grover's Algorithm
Introduction to Quantum Mechanics for ICT, A.Y. 2020–2021, Semester: 1

Giacomo Lancellotti - Matteo Lodi

November 2020

Abstract

We present an overview of Grover's search algorithm. The problem concerns an unsorted database containing N items, out of which one or more items satisfies a given condition. The algorithm is an example of quantum advantage over classical computing in terms of complexity. An implementation of the algorithm is then presented solving a 2 x 2 Sudoku puzzle using the Qiskit framework.

Contents

1 History & Quantum Phenomena

In the early '80s, Benioff showed for the first time that quantum mechanical computers were possible. He basically proved that for each Turing machine and for each time step, it is possible to build an Hamiltonian capable of describing the correct computation step result. It is also worth mentioning that the models proposed did require energy to work but only at times when the operator has to check whether the computation has finished. An important remark is that it was first shown the fact that quantum computing was at least as powerful as classical computing.

Later works from the mid '80s and early '90s conceptualized and formalized better this paradigm. At the base of computer theory, actually the very concept of computation as we mean it, lays the Church-Turing hypothesis (1936), which states that any 'function that would normally be regarded as computable' can be computed by the universal Turing machine. Deutsch, starting with this sentence, extended the concept concluding that it not only was possible to build an Hamiltonian, thus an equivalent quantum circuit, from a Turing machine, but that these new machines were actually more powerful than the classic ones. Still, this was true as long as the task the machine is performing can be considered "specialized".

Feynman in 1985 was the first to formalize how the computing engine of a quantum machine works. By stating four primitive elements (NOT, AND, FANOUT and EXCHANGE) he managed to build quantum gates, the quantum correspondence of the classic minimal components of a logic unit (more on this later). An important example of this machine performing better than a classic one in a specialized task was given by Shor in 1994: a quantum factoring algorithm could, for the first time, reach polynomial complexity.

Following similar steps, Grover's algorithm from the 1996 paper "A fast quantum mechanical algorithm for database search" gives a solution which is, complexity-wise, within a small constant factor the fastest possible over a search problem.

The main difference between classical computing and quantum computing is also the reason for this exponential advantage: the **Superposition**. Classic probabilistic algorithms do exist, and quantum computing follows the same concept of distribution of states: instead of having the system in a specified state, it is in a distribution over various states with a certain probability of being in each one of them. However, unlike classical systems, the probability vector does not completely describe the system in this new computing method. In order to completely describe the system we need the amplitude in each state. The evolution of the system is obtained by pre-multiplying this amplitude vector (that describes the distribution of amplitudes over various states) by a transition matrix, the entries of which are complex in general. So now the probabilities in any state are given by the square of the absolute values of the amplitude in that state.

Gates are the core logical elements of a quantum circuit. Following Feynman research, it is possible to consider just two primitive elements: the NOT gate and

the AND gate and extend the logic from them. It was thus discovered that those same operations that defined classical computing could be transposed directly in this new quantum world. The evolution of a quantum system can now be directly influenced by these gates. Schrödinger equation physically describes a quantum system. Quantum gates are **Unitary**, because they are implemented via the action of a Hamiltonian for a specific time, which gives a unitary time evolution according to the Schrödinger equation. During computation we want unitary operators because we would like transformations to act linearly on the state (so that a matrix representation is possible) and to preserve probabilities (meaning to preserve the inner product). In other words, the kind of transformations we want to apply to the system have to be **Reversible**, other than the necessary measurements on the system.

2 Complexity Theory

The main goal of complexity theory is to understand and find efficient algorithms to solve specific problems. It tries to categorize computational problems by looking at how many resources it needs, typically time (computational steps) and space (memory) needed.

The definition of efficiency is strictly related to the Church-Turing hypothesis, which states that all physical implementations of computing engines can be simulated in polynomial time by a probabilistic Turing machine. Using this hypothesis we can wonder about efficiency in a very general way.

A first class of complexity is the **P** class. An algorithm belongs to this class if a corresponding deterministic Turing machine that ends computation in polynomial time exists. If we extend a little this class, we can introduce the probabilistic algorithms: those algorithms whose corresponding probabilistic Turing machine ends computation in polynomial time, and the results has to be correct at least $2/3$ of the times. Now we are in the **BBP** class. The **BBP** class together with the **P** class of computational complexity include all the efficiently solvable algorithms, according to the definition of efficiency explained above.

The class of complexity that contains all non-efficiently solvable problems is the **NP** class. Informally, this class contains all the problems whose solution is efficiently verifiable, but its computation is "hard". The corresponding Turing machine is non-deterministic.

The **P** vs. **NP** relationship has always been the subject of many studies, even today. We can easily demonstrate the inclusion $\mathbf{P} \subseteq \mathbf{NP}$, yet the opposite inclusion is still today an unsolved problem. There are many reasons to believe that $\mathbf{P} \neq \mathbf{NP}$, but a formal demonstration doesn't exist yet, not even by taking into account the new quantum paradigm.

With the development of Quantum mechanics, there was a need to add this new paradigm into complexity theory, in order to understand if the classes of problems solvable in this field were different than the classic Turing-related ones. As it turns out, it is possible to formalize a Quantum Turing machine which behaves analogously to the classic Turing machine (Yao, 1993).

The quantum class corresponding to the **BPP** complexity class is the **BQP** class (bounded error quantum polynomial time). If a problem belongs to this class, its corresponding Quantum Turing Machine accepts an input, if it is a solution, in polynomial time with probability at least 2/3, and rejects it when it is not a solution with the same probability. Since it is easily demonstrated that a quantum Turing machine can simulate a probabilistic Turing machine, we have that $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{BQP}$.

The key aspect here, is that there are no results, up until now, that indicate that $\mathbf{BPP} \neq \mathbf{BQP}$, thus there is no sign that the quantum model is strictly superior to the classic model.

Grover's algorithm solving the database search problem is an example of a sub-exponential advantage. A NP-complete problem cannot achieve an exponential speed-up when approached using the new quantum paradigm: Grover's speed-up is "just" quadratic, yet it is almost as fast as a these kind of algorithms can be in this complexity class.

3 Quantum computation

In a classical machine the goal is to implement a function that taken as input a set of bit in a generic state $\mathbf{x}$ produces an output $\mathbf{y}$ which may, in general, use a different number of bits from the input. In other terms a circuit computes the value of the function f

$$f : \{0,1\}^n \mapsto \{0,1\}^m \tag{1}$$

$$x \mapsto f(x) = y \tag{2}$$

Similarly what a quantum circuit does can be summarized as follow:

$$U : |z, 0\rangle \mapsto |z, f(z)\rangle \tag{3}$$

where $|z\rangle$ is a generic state of a n qubit system and $|0\rangle$ is the state initialized to $|00...0\rangle$ of a system of m qubit. This has advantages over the generic classic function specified above. To understand this concept, suppose we have a single qubit register as input and we build the generic quantum circuit that implements the operator U:

$$U : |0, 0\rangle \mapsto |0, f(0)\rangle \tag{4}$$

$$U : |1, 0\rangle \mapsto |1, f(1)\rangle \tag{5}$$

This is what a classical function could do as well. However,the starting qubit could be also in a superposition of states:

$$|z\rangle = \alpha|0\rangle + \beta|1\rangle \tag{6}$$

and if we apply the circuit to this state, we obtain:

$$U : |z, 0\rangle = \alpha|0, f(0)\rangle + \beta|1, f(1)\rangle \tag{7}$$

The final result contains the information both on $f(0)$ and on $f(1)$, it is like we have evaluated $f(z)$ simultaneously for two values of z. This is due to the **intrinsic quantum parallelism** that derives from the superposition principle and from the linearity of the quantum mechanics. If we consider that the Hilbert space of a n qubit system has a dimension of 2^n we arrive at the conclusion that we are able to compute simultaneously a function over 2^n different inputs.

If α_x is the amplitude of the generic state x

$$U \sum_{x=00..0}^{11..1} \alpha_x|x, 0\rangle = \sum_{x=00..0}^{11..1} \alpha_x U|x, 0\rangle = \sum_{x=00..0}^{11..1} \alpha_x|x, f(x)\rangle \tag{8}$$

However, this type of parallelism is not immediately useful, as to obtain some results we need to measure the system making it collapse into a single eigenstate. Even if we seem to have reached a dead end, by accessing only a single value of the function, the aim of a quantum algorithm is to take advantages from the intrinsic parallelism. The nature of the quantum parallelism is different from the classic version in which we divide the problem to make it compute from different circuits simultaneously. In the quantum world we use a single circuit to evaluate a function f over different values exploiting the superposition.

4 Problem

In computer science the search problems constitute a large class of different problems like the search of an element in a database, solving a SAT or finding the coloring scheme of a graph. In the most general form a search problem is structured as: "find an element **x** in a set of possible solutions such that a certain condition **P(x)** is true".

An unstructured search problem is a search problem where the structure of the solution space is unknown. On the other hand for a structured one this information can be used to build efficient algorithms like the binary search algorithm for problems where the solution space has a binary tree structure.

In the general case for an unstructured problem the best classical strategy is to check the condition **P(x)** over all the possible elements of the set, if the latter has a dimension **N** the search will require **O(N)** evaluations of the condition **P**. Thus, following an average reasoning, **N/2** accesses are necessary to solve the problem.

On a quantum machine this type of problems can be solved with a small probability error in $\mathbf{O}(\sqrt{N})$ steps using the algorithm described in this paper.

5 Algorithm

5.1 Setting up the algorithm

As anticipated before the problem we want to solve is that of linear search. Giving an unstructured set of elements and a certain condition the problem is to find the elements that satisfy the given condition. More generally instead of searching directly on the objects we associate an index to them refactoring the problem. Giving the set $X = \{00..0, ..., 11..1\}$ and a function $f(x)$ with $x \in X$ such that $f(x) = 1$ if x is the solution to the problem and $f(x) = 0$ otherwise. The problem is now to find the x that satisfies $f(x) = 1$.

The function f also called the oracle is given by the specific problem and it is something able to identify the solution, later on we will see its global effect.

Quantistically speaking, we can match the element of X with the base state of an Hilbert space. Therefore if X contains N elements the Hilbert space has dimension N and our computational base must have $n = \lfloor \log_2 N \rfloor + 1$ qubit. The oracle can be expressed like an unitary operator $\hat{O}$ that applies on the base state $|x\rangle$ the transformation

$$|x\rangle|q\rangle \xrightarrow{\hat{O}} |x\rangle|q \oplus f(x)\rangle \tag{9}$$

intuitively if x is the solution the qubit $|q\rangle$ is flipped. At this point we could find the solution applying $\hat{O}$ to the state $|x\rangle|0\rangle \forall x$ and sensing as long as the auxiliary qubit becomes $|1\rangle$. Things get more interesting if we initialize the auxiliary qubit in superposition $(|0\rangle - |1\rangle)/\sqrt{2}$ if x is not the solution nothing change, if it is the qubit becomes $-(|0\rangle - |1\rangle)/\sqrt{2}$, in other word the oracle acts like a phase shift of π radians.

$$|x\rangle \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \xrightarrow{\hat{O}} (-1)^{f(x)}|x\rangle \frac{(|0\rangle - |1\rangle)}{\sqrt{2}} \tag{10}$$

Notice that the state of the auxiliary qubit does not change, this bring us to simplify the formula into a more compact representation

$$|x\rangle \xrightarrow{\hat{O}} (-1)^{f(x)}|x\rangle \tag{11}$$

5.2 Algorithm description

The general idea of Grover algorithm is to amplify the amplitude of the solutions in order to make the system collapse to those values when it is measured. At first we have to initialize the system to the uniform distribution $(\frac{1}{N}, \frac{1}{N}, ... \frac{1}{N})$ so as to have the same amplitude in every of the N states. This can be achieved taking an n qubit register initialized in the state $|0\rangle^{\otimes n}$ and applying an Hadamard gate over all qubit with the operator $H^{\otimes n}$. The corresponding state is now a superposition

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \tag{12}$$

Now if we measure the system the probability of getting the right answer would be $1/N$. To increase the amplitude of a certain state we have to design a new operator $\hat{D}$ that we will call Diffuser whose effect is

$$\hat{D} = 2\,|s\rangle\,\langle s| - \hat{I} \tag{13}$$

This operator can be implemented as $D = H^{\otimes n} R H^{\otimes n}$ where H is the Hadamard operator and R is the rotation operator, which can be described by a transform matrix $R = diag(1, -1, -1, .., -1)$. In practice $\hat{R}$ makes a phase shift of π over all computational states other than $|0\rangle$,that is

$$R : |x\rangle \mapsto \begin{cases} |x\rangle & \text{if } x = 0 \\ -|x\rangle & \text{if } x > 0 \end{cases} \tag{14}$$

It easy to verify that $R = 2\,|0\rangle\,\langle 0| - I$. The global effect of the diffuser is what is called "inversion about average", formally this operation correspond to

$$H^{\otimes n} R H^{\otimes n} = H^{\otimes n}(2\,|0\rangle\,\langle 0| - I)H^{\otimes n} = 2\,|s\rangle\,\langle s| - I \tag{15}$$

Applying the operator D over a generic state $\sum_N \alpha_x\,|x\rangle$ the result is

$$(2\,|s\rangle\,\langle s| - I)(\sum_N \alpha_x\,|x\rangle) = \sum_N (2A - \alpha_x)\,|x\rangle \tag{16}$$

where $A = \sum_N \alpha_x\,|x\rangle\,/N$ is the average of all amplitudes of the generic state.
This concept becomes more intuitively if we observe the matrix definition of D

$$D = \begin{bmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \dots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \dots & \frac{2}{N} \\ \dots & \dots & \dots & \dots \\ \frac{2}{N} & \frac{2}{N} & \dots & \frac{2}{N} - 1 \end{bmatrix} \tag{17}$$

and applying it on $\sum_N \alpha_x\,|x\rangle$ and develop the usual rows-columns product

$$\begin{bmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \dots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \dots & \frac{2}{N} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \frac{2}{N} & \frac{2}{N} & \dots & \frac{2}{N} - 1 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \dots \\ \alpha_x \\ \dots \\ \alpha_{N-1} \end{bmatrix} = \begin{bmatrix} \frac{2}{N}\sum_{i=0}^{N-1} \alpha_i - \alpha_0 \\ \dots \\ \frac{2}{N}\sum_{i=0}^{N-1} \alpha_i - \alpha_x \\ \dots \\ \frac{2}{N}\sum_{i=0}^{N-1} \alpha_i - \alpha_{N-1} \end{bmatrix} \tag{18}$$

we see how the double of amplitude' average $\frac{2}{N}\sum_{i=0}^{N-1} \alpha_i$ is added up with the inverse of all aplitudes. The last point we need to analyze is how to mark

with a negative sign the amplitude associated with the solution states, in this the aforementioned oracle operator $\hat{O}$ is useful. Its effect to a generic state is

$$O : \sum_{i=0}^{N-1} \alpha_i |x\rangle \mapsto \sum_{i=0}^{N-1} (-1)^{f(x)} \alpha_i |x\rangle \tag{19}$$

To recap what we have done so far the oracle is used to encode the information about the specific problem translating it into a phase shift over the system state, while the diffuser amplifies this marked state amplitude over all the others. This bring us to the formulation of the Grover operator

$$\hat{G} = \hat{D}\hat{O} \tag{20}$$

the whole algorithm can now be seen as the iteratively application of this new operator for a specific number of times.

The circuit that realize Grover's algorithm is reported in [Figure 1].

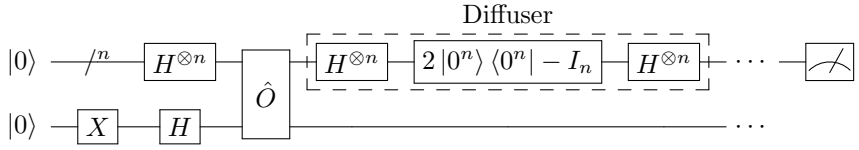

Figure 1: Grover's algorithm circuit

These are the steps of the algorithm:

1. Initialization.
$$|0\rangle^{\otimes n} |0\rangle \tag{21}$$

2. Application of $H^{\otimes n}$ to first n qubit and HX to the last ancilla.
$$\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \alpha_i |x\rangle \frac{[|0\rangle - |1\rangle]}{\sqrt{2}} \tag{22}$$

3. The Oracle inverts the amplitudes of $|x\rangle$ that are solutions, leaving unchanged the rest.
$$\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} (-1)^{f(x)} \alpha_i |x\rangle \frac{[|0\rangle - |1\rangle]}{\sqrt{2}} \tag{23}$$

4. The Diffuser amplifies those solution's amplitudes by inverting them over the average of all the amplitudes.
$$\sum_{i=0}^{N-1} (2A - \alpha_i) |x\rangle \frac{[|0\rangle - |1\rangle]}{\sqrt{2}} \tag{24}$$
where $A = \sum_N \alpha_x |x\rangle / N$ is the average of all the amplitudes of the generic state.

5. Finally the measurement yields the solution (or one of the possible ones if more than one solution is present).

5.3 Inversion about Average

As we said, at the heart of the algorithm there are two elements: the Oracle inverts the amplitudes of the states that can be considered solutions to the search problem, whereas the Diffuser applies what's called the "Inversion about Average".

The objective of this procedure is to amplify the amplitudes of the solutions states, which were multiplied by -1 by the Oracle, by inverting them on the average of all the possible states (actually, the inversion is applied to the whole state). What follows is that what had, at the beginning of the diffuse operation, a negative amplitude has now not only a positive amplitude but it is much higher than the non-solution states, specifically twice as higher as the average. The successive iterations of the Grover operator will enhance more and more the solution states: the Oracle will once again flip negatively their amplitudes and the Diffuser will amplify them positively.

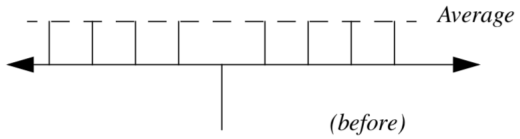

Figure 2: What is given as input to the Diffuser by the Oracle: the solution state is negative but its absolute value is the same of the non-solution state.

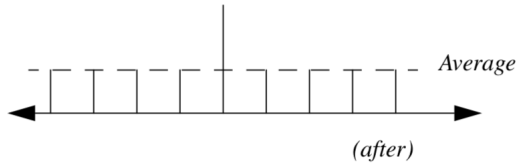

Figure 3: The output of the Diffuser: the negative solutions are now inverted about average, as well as doubled in length.

From a more mathematical point of view, the Diffuser transformation D can be seen as $D \equiv -I + 2P$ where I is the identity matrix and P is a projection matrix $P_{ij} = 1/N$ for all i and j. It is easy to prove that D is unitary. Since the amplitudes of the non-solution states are equal to approximately $1/\sqrt{N}$, if we have one solution state the average will be approximately still $1/\sqrt{N}$ so the transformation will influence only that state, which now is increased by $2/\sqrt{N}$.

Lastly, since we want a certain result, the Grover operator cannot be applied only once because the amplitudes of the solution vectors are simply not big enough to obtain it. Each iteration of this loop increases the amplitude of the solution state by $\mathbf{O}(1/\sqrt{N})$; as a result we will need $\mathbf{O}(\sqrt{N})$ repetitions of the loop to make it so that those amplitudes reach approximately the measurement certainty $\mathbf{O(1)}$.

A separate consideration has to be done for a situation in which the algorithm has $M > 1$ elements that satisfy the search criteria (thus multiple solution states). As it turns out, the only alteration we have to apply is the number of iterations of the algorithm itself: instead of $\mathbf{O}(\sqrt{N})$ repetitions if we take into account the M solutions we will have to loop it a total of $\mathbf{O}(\sqrt{N/M})$ times. Intuitively, having more solutions brings the average of the whole state to a lower value (again, those solutions are negative). This means that the Inversion about Average works faster with regards to the increase of amplitude of those states of interest, thus the repetitions need not to be done as often.

Lastly, executing more iterations than the optimal ones does not improve the result, it only diverges faster and faster. The closer we are to the asymptotic optimal number, the better.

6 Sudoku problem

Sudoku is a logic-based, combinatorial number-placement puzzle. The general problem of solving Sudoku puzzles of N x N blocks is known to be NP-complete. Algorithms such as backtracking and annealing can solve most of the 9 x 9 puzzles. However, for a large value of n, a combinatorial explosion occurs which limits the properties of the Sudoku that can be constructed, analyzed and solved. Over the time, many variants of Sudoku have been introduced. There are two main aspects of problem difficulty. The first is the complexity of individual steps (logical operations) involved in solving the problem and the second aspect being the structure of dependency among individual steps, i.e., whether steps are independent (can be applied in parallel) or they are dependent (must be applied sequentially). One is yet to come up with a quantum algorithm which can deterministically achieve this feat for any N x N Sudoku.

Our case study is based on a 2 x 2 Sudoku to show how a quantum algorithm can be implemented to solve a real problem and how to take advantage in terms of computation.

A 2 x 2 Sudoku is a grid of four blocks which contains a value between 0 or 1, where every row and column can not have repeated values.

The only two valid assignments out of 16 combinations are reported in [Figure 4].

To figure out how to solve such problem, first we have to encode it into a 'quantum representation', using a quantum register of four qubit we assign to every cell a qubit in this register that we will call the state register, namely $|A\rangle |B\rangle |C\rangle |D\rangle$ [Figure 5].

The next step is to configure the constraints on the values contained in the

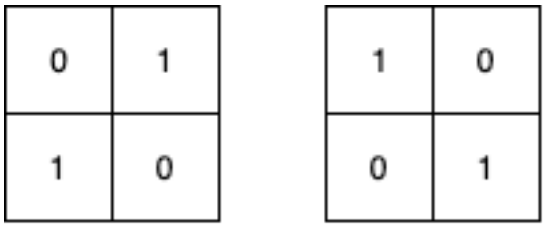

Figure 4: 2x2 valid solutions

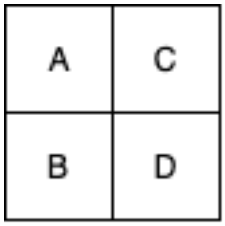

Figure 5: Variables assignment

row and column. We adopt a logical approach checking that the values on the same row are different (same reasoning for columns), in order to do this it was used a Quantum Bit String Comparator (QBSC) that takes as input two N qubit strings $|a\rangle = |a_1\rangle |a_2\rangle ... |a_n\rangle$ and $|b\rangle = |b_1\rangle |b_2\rangle ... |b_n\rangle$ and it returns which of the two strings is greater (in binary representation) or if they are equal. Basically, the quantum circuit compare the strings bit-to-bit from the left (most significant bit) to the right (less significant bit). Measuring the output that we will call O_1 and O_2 we obtain this table of values:

a, b	O_1	O_2
$a > b$	1	0
$b > a$	0	1
$a = b$	0	0

Initially, the comparison between the first bit of each string is dominant, that is, if they are different, then the outputs will be $O1 = a0$ and $O2 = b0$. If they are equal the comparison between the second bit of each string will be dominant and so on, obviously only the less significant bit does not have the dominion transfer circuit that is realized with a Toffoli gate with 0 activation and two standard CCNOT.

Even if the QBSC [Figure 6] seems too powerful for our purposes because in a 2 x 2 case it is used only the QtQ part, its usefulness becomes clear if we consider cases in which the variables are coded whit multiple qubit or more complex controls are required like in 'Sudoku city' version.

The next step is the construction of the Oracle that is made up of four qbit-to-qbit comparators that check the logical constraints: $A \neq B$,$A \neq C$,$C \neq D$

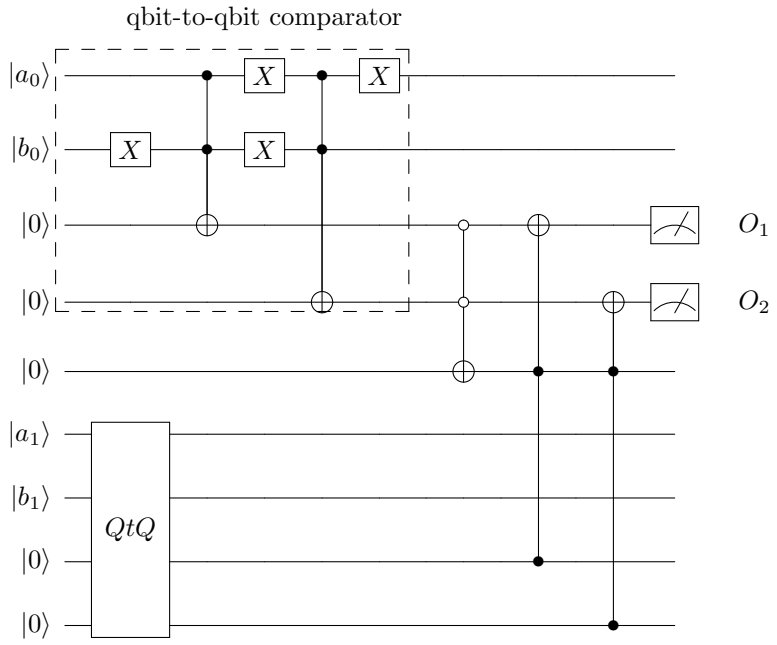

Figure 6: QBSC for 2 qubit string

and $B \neq D$. The second ancilla qubit of every comparator (O_2) is set to 1 if the constraint is true [Figure 7] .

At this point we have everything we need to solve the 2 x 2 Sudoku. Analyzing the final circuit [Figure 8]: as a first step we apply the superposition to the whole state register and the last ancilla is set to $(|0\rangle - |1\rangle)/\sqrt{2} = |-\rangle$. Using the Oracle we are able to mark the correct states and with the application of the Diffuser operator to amplify their amplitude. The second Oracle application is used for uncomputing the ancillas, a necessary move in order to have a repeatable iteration of the same gates over and over that does not need new states every time. The last ancilla is a key element to the Oracle functioning since it checks that all the states, amplified subsequently by the Diffuser [Figure 9], are only the correct ones: those that follow the the $O_1 \oplus O_2$ condition. Finally the measurement gives us a single possible solution to the problem, thus multiple measurements (meaning multiple iterations of the whole algorithm) are needed in order to find all the possible solutions.

It is clear now how even with the help of quantum computing the difficulty of the problem explodes very quickly, this type of approach can be easly extended to every Sudoku of dimension n increasing the number of comparators in the Oracle with consequently a high number of ancilla, far beyond the present day hardware power.

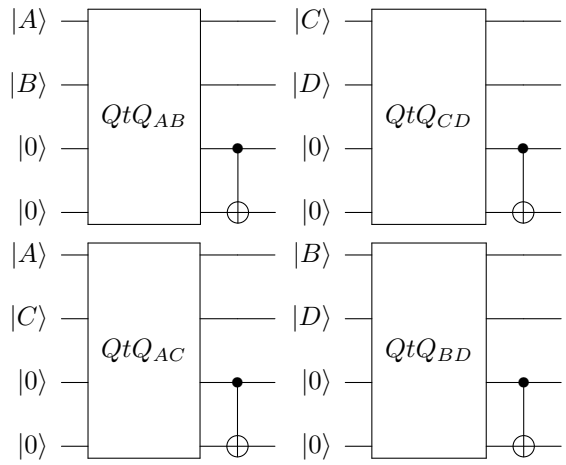

Figure 7: Oracle

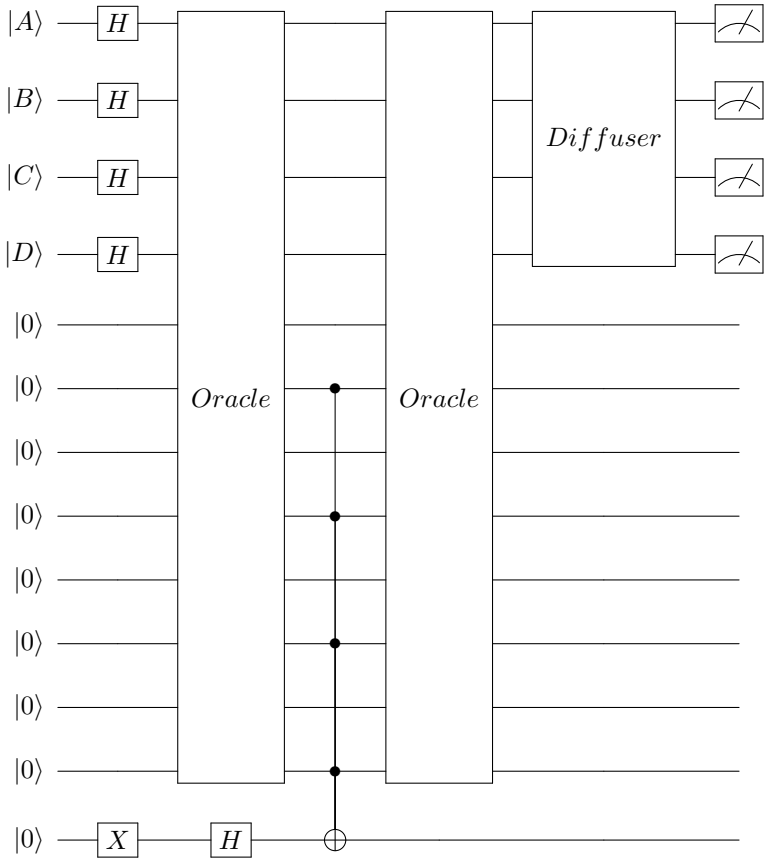

Figure 8: 2 x 2 Sudoku

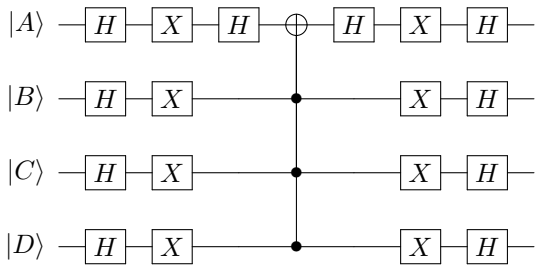

Figure 9: Diffuser

7 Code and experimental results

```
from qiskit import *
import numpy as np
from qiskit.providers.aer import QasmSimulator
from qiskit.visualization import plot_histogram
from operator import itemgetter


a_string = QuantumRegister(1, name='a string')
b_string = QuantumRegister(1, name='b string')
c_string = QuantumRegister(1, name='c string')
d_string = QuantumRegister(1, name='d string')

all_0_ancilla_ab = QuantumRegister(2, name='all 0  ancilla ab')
all_0_ancilla_cd = QuantumRegister(2, name='all 0  ancilla cd')
all_0_ancilla_ac = QuantumRegister(2, name='all 0  ancilla ac')
all_0_ancilla_bd = QuantumRegister(2, name='all 0  ancilla bd')

out = QuantumRegister(1,name='output')

classical = ClassicalRegister(4, name='measure')

qc = QuantumCircuit(a_string, b_string, c_string, d_string, all_0_ancilla_ab,
    all_0_ancilla_cd, all_0_ancilla_ac, all_0_ancilla_bd, out, classical)

#out in -
qc.x(out)
qc.h(out)

#superposition of input
qc.h(a_string[0])
qc.h(b_string[0])
```

```
qc.h(c_string[0])
qc.h(d_string[0])

qc.barrier()


def oracle(qc):

    # comparator ab
    qc.x(b_string[0])
    qc.mct([a_string[0], b_string[0]], all_0_ancilla_ab[0])
    qc.x(a_string[0])
    qc.x(b_string[0])
    qc.mct([a_string[0], b_string[0]], all_0_ancilla_ab[1])
    qc.x(a_string[0])
    qc.cnot(all_0_ancilla_ab[0], all_0_ancilla_ab[1])
    qc.barrier()

    # comparator ac
    qc.x(c_string[0])
    qc.mct([a_string[0], c_string[0]], all_0_ancilla_ac[0])
    qc.x(a_string[0])
    qc.x(c_string[0])
    qc.mct([a_string[0], c_string[0]], all_0_ancilla_ac[1])
    qc.x(a_string[0])
    qc.cnot(all_0_ancilla_ac[0], all_0_ancilla_ac[1])
    qc.barrier()

    # comparator cd
    qc.x(d_string[0])
    qc.mct([d_string[0], c_string[0]], all_0_ancilla_cd[0])
    qc.x(c_string[0])
    qc.x(d_string[0])
    qc.mct([d_string[0], c_string[0]], all_0_ancilla_cd[1])
    qc.x(c_string[0])
    qc.cnot(all_0_ancilla_cd[0], all_0_ancilla_cd[1])
    qc.barrier()

    # comparator bd
    qc.x(b_string[0])
    qc.mct([d_string[0], b_string[0]], all_0_ancilla_bd[0])
    qc.x(d_string[0])
    qc.x(b_string[0])
    qc.mct([d_string[0], b_string[0]], all_0_ancilla_bd[1])
    qc.x(d_string[0])
    qc.cnot(all_0_ancilla_bd[0], all_0_ancilla_bd[1])
```

```
    qc.barrier()


def diffuser(qc):

    qc.h(a_string[0])
    qc.h(b_string[0])
    qc.h(c_string[0])
    qc.h(d_string[0])

    qc.x(a_string[0])
    qc.x(b_string[0])
    qc.x(c_string[0])
    qc.x(d_string[0])

    qc.h(a_string[0])
    qc.mct([ b_string[0], c_string[0], d_string[0]], a_string[0])
    qc.h(a_string[0])

    qc.x(a_string[0])
    qc.x(b_string[0])
    qc.x(c_string[0])
    qc.x(d_string[0])

    qc.h(a_string[0])
    qc.h(b_string[0])
    qc.h(c_string[0])
    qc.h(d_string[0])

    qc.barrier()


#n is the number of iteration of Grover operator
n = 2

for i in range (n):

    oracle(qc)
    qc.mct([all_0_ancilla_ab[1], all_0_ancilla_ac[1],all_0_ancilla_cd[1],
        all_0_ancilla_bd[1]], out)
    qc.barrier()

    oracle(qc)
    diffuser(qc)

qc.measure(a_string[0],classical[0])
```

```
qc.measure(b_string[0],classical[1])
qc.measure(c_string[0],classical[2])
qc.measure(d_string[0],classical[3])

print(qc)

def print_sudoku(dict):
    list=[]
    for item in dict:
        for a in item:
            list.append(a)
        data=np.array(list)
        shape=(2,2)
        sudoku=data.reshape(shape)
        print(sudoku)
        print('\n')
        list.clear()

#start simulation
backend = QasmSimulator()
result = execute(qc, backend=backend, shots=1024).result()
answer = result.get_counts()
plot_histogram(answer).show()

#return N max values
res = dict(sorted(answer.items(), key=itemgetter(1), reverse=True)[:2])

#print the sudoku schemas
print('the solutions are:')
print_sudoku(res)
```

We show the result of the simulation: the two states whose amplitudes are pointed up in the histogram [Figure 10] correspond to the only two possible solutions in a 2 x 2 Sudoku puzzle, $ABCD = 1001$ or 0110 . Each other state is a possible but non-valid assignment to the Sudoku, hence the Oracle and the Diffuser do not enhance their amplitude, leaving them unchanged.

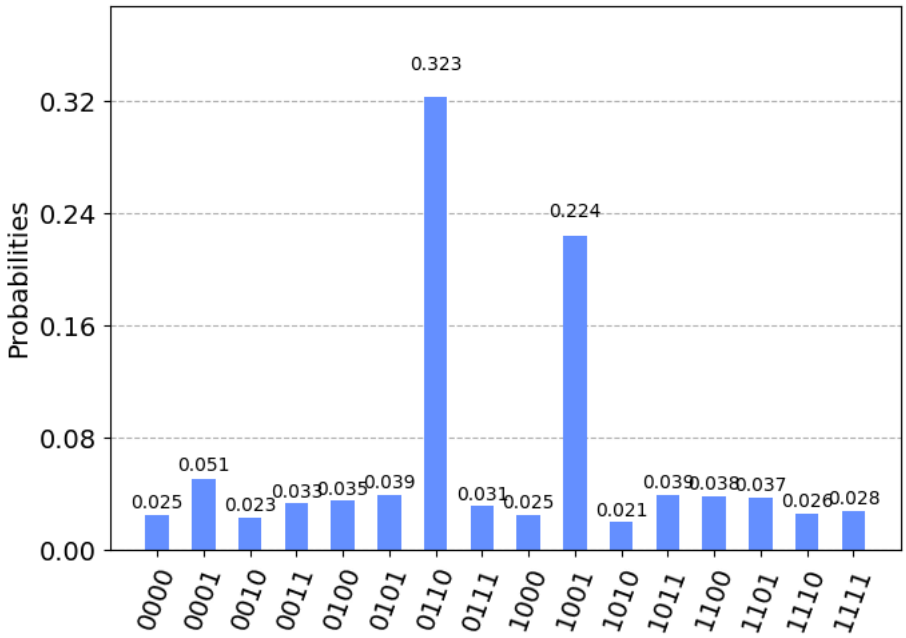

Figure 10: Amplitude distribution

8 Conclusion

The aim of this work is to show the advantages of quantum computing over the classical one and its limits. In 1994 Peter Shor showed how a quantum computer could factorize a number in polynomial time launching the quantum computing into a new era. To create his algorithm Shor used certain mathematical properties of composite numbers that are well suited to be exploited on quantum hardware. NP-complete problems do not seem to share those special properties. The question thus remains unanswered: is there an efficient quantum algorithm to solve NP-complete problems? Until today no such algorithm has been found.

We can say that a quantum algorithm capable of efficiently solving NP-complete problems (with an exponential speedup) has to exploit the problem's inner structure in a new way far beyond the present day techniques. You can not achieve an exponential speedup by treating the problem as a structure-less 'black boxes' because in this inefficient way an exponential number of solutions have to be tested in parallel. The Grover algorithm comes in our aid in this terms giving us a quadratic speedup in this class of problems that is a useful advantage in some scenarios, but a square root does not transform an exponential time into a polynomial one, it just produces a smaller exponential and this is proven to be the best result possible in a black box search. This highlights the limitations of today's quantum computers but not rule out the possibility that efficient

algorithm for NP-complete problems could be discovered. If such algorithms existed, however, they would have to exploit the problems' structures in ways that are unlike anything we have seen, quantum mechanics by itself is not going to do the job.

A new computing paradigm other than the Turing Machine should be discovered or broadly revise our knowledge base. This is what should inspire computer scientists in designing new quantum algorithms.

References

[1] **Lov K. Grover**, *A fast quantum mechanical algorithm for database search*
Bell Labs, *1996*

[2] Qiskit's Grover's algorithm web-page
`https://qiskit.org/textbook/ch-algorithms/grover.html`

[3] Wikipedia's Grover's algorithm web-page
`https://en.wikipedia.org/wiki/Grover%27s_algorithm`

[4] **Daan Sprenkels**, *Grover's algorithm*
Radboud University Nijmegen, The Netherlands

[5] **Richard P. Feynman**, *Quantum Mechanical Computers*
Princeton University, *1985*

[6] **Peter W. Shor**, *Algorithms for quantum computation: discrete logarithms and factoring*
Massachusetts Institute of Technology, *1994*

[7] **Riccardo Rende, Carlo Mastroianni, Francesco Plastina**, *Quantum Algorithm for the Boolean Satisfiability Problem*
ICAR CNR, *2019*

[8] **David Sena Oliveira, Rubens Viana Ramos**, *Quantum bit string comparator: circuits and applications*
Departamento de Engenharia de Teleinformática Universidade Federal do Ceará, *2007*

[9] **Alessandra Di Pierro**, *Quantum Computing*
University of Verona

[10] **Andrew Chi-Chih Yao**, *Quantum Circuit Complexity*
Princeton University

[11] **Scott Aaronson**, *The limits of quantum*
Scientific American, *2008*