A general quantum method to solve the graph K-colouring problem

Maurice Clerc

A general quantum method to solve the graph K-colouring problem

Maurice Clerc

10th July 2020

Abstract

For a N nodes graph, the complexity of the proposed method, called NK-method, is $O\left(N^2\right)$ in number of qubits and $O\left(N^4\right)$ in number of gates. It works for any K, but needs $N^2 - NK + 1$ qubits (and $NK + 1$ classical bits). As this number is quite high, we also propose a Q-method that requires significantly less qubits, but whose quantum circuit is far more complicated to build when K increases. For the NK-method we give a general Qiskit code that builds and simulates the quantum circuit. For the Q-method we just give a code for the 3-coloring. In these codes we use only gates whose unitary matrices are real. These codes can find all solutions that require at most K colours.

1 Introduction

Quantum computers manipulate qubits, so it is necessary to find a binary coding of the problem at hand. Some previous works suggest codings and quantum circuits that work only for 2 or 3 colours ([3]). Here we propose one that is a pure, general, constructive and complete[1] quantum approach to solve the graph colouring problem.

2 Two colours

This is of course just to present some concepts, for in this particular case a simple deterministic greedy algorithm is more efficient than a quantum one. Let us consider the graph of the figure 1. It is easy to see that a colouring is possible, for there is no cycle.

2.1 Method 1

A binary coding of the colours of the nodes is straightforward : 0 for one colour, 1 for the other. Therefore, to describe the quantum state of each node, we need just one qubit. A quantum circuit that generates all possible solutions (here only two) is given on the figure 2[2]. It works, let us examine in details how, but try to guess why the reasoning to build it is nevertheless erroneous.

1. Not hybrid, valid for any kind of graph, and finding all solutions with at most K colours. Not always complete, to be fair, because of the intrinsic random nature of the quantum approach. If the number of shots is too small you may not find all solutions. However, usually, we just want to find *one*.
2. https://quantum-computing.ibm.com/composer/

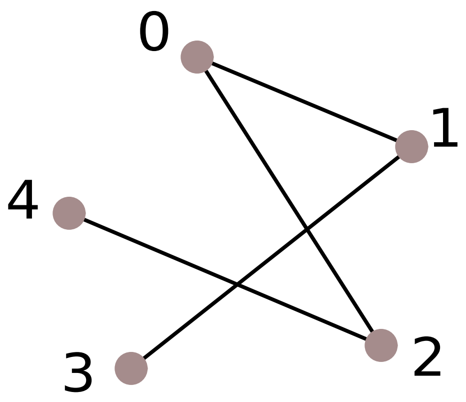


Figure 1: A 5-nodes graph for which a 2-coloring is possible.

Step 1

The Hadamard gate (symbol H) is applied to one qubit (one node), so that it potentially can have the colour 0 or 1. The others are set to |1> thanks to the NOT gates (symbol X), so that they later control some qubits.

Step 2

There is an edge between nodes 0 and 1. So, thanks to a CNOT gate we ensure that qubit 0 can not have the same colour than the qubit 1.

Step 3

There is an edge between nodes 0 and 2. So, thanks to a CNOT gate we ensure that qubit 0 can not have the same colour than the qubit 2.

Step 4

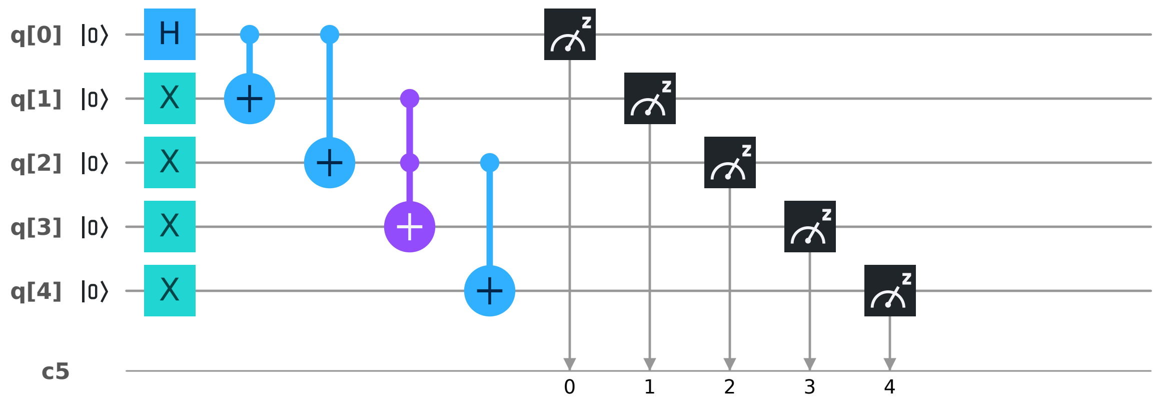
There is no edge between 1 and 2, so they can be of the same colour, but the edges 0-1 and 0-2 exist. A CCNOT gate ensures that 0 does not have the colour of 1 and 2.

Step 5

There is an edge between nodes 2 and 4. A CNOT gate ensures that they can not have the same colour.

What is wrong with this circuit[3]? Well, not exactly wrong, for the result is indeed correct, but it is a kind of cheating: it just simulates a greedy algorithm with the two possible initial colours on the node 0. Therefore it is inelegant and moreover not general, for we have to carefully examine the structure of the graph to build it. We can do better.

3. I used the IBM site https://quantum-computing.ibm.com/composer/



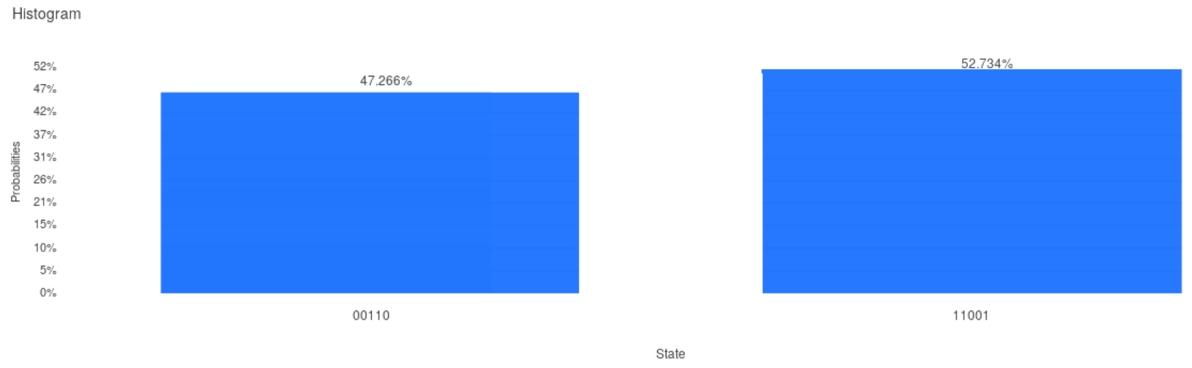
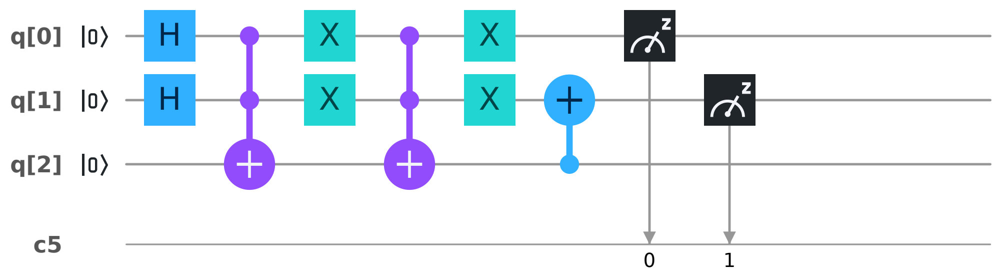


Figure 2: A possible circuit for 2-coloring and the two solutions, after 1024 shots. Colour 0 for nodes 0, 4, 5, or colour 0 for nodes 1 and 2.



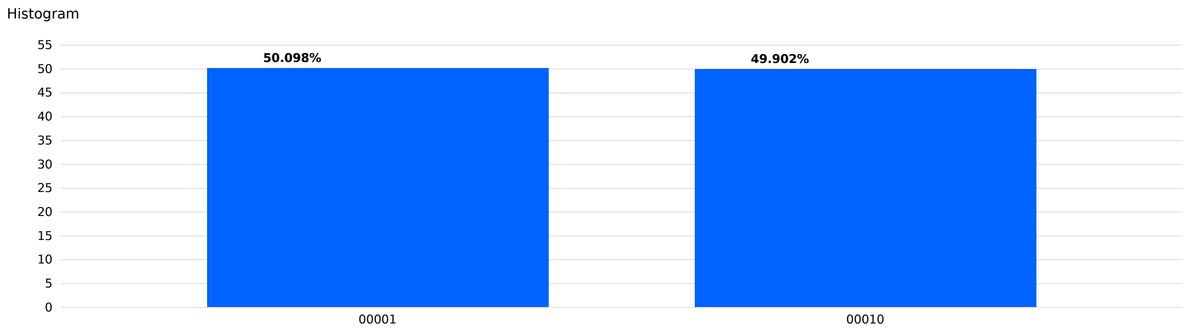


Figure 3: A circuit for the 2-nodes 1-edge graph, and the two solutions after 1000 shots. Colour 0 for node 0, or colour 0 for node 1.

2.2 Method 2

We do not assume 0 or 1 to initialise the nodes/qubits, and we just embed in the circuit the existence of each edge. Let us see first how we can do that for the simplistic graph that has only two nodes and one edge. Our basic circuit have to ensure the nodes can not have the same colour, and designing it is quite straightforward. As we can see on the figure3 it needs an ancillary bit, though. Note that some systems directly accept the negative CCNOT gate. Here, it is simulated by applying X, X, CCNOT, X, X.

We can then apply a similar "block" for the ends of each edge. The figure 4 shows how we can assemble two such blocks[4]. A result after 1000 shots is {'101': 488, '010': 512}.

And for our 5-node graph, the circuit of the figure 5 can be automatically generated, one block for each edge. The result after 1000 shots is {'11001': 511, '00110': 489}.

Looks nice. But isn't the graph too simple? What about if there are cycles? The good news is that it works perfectly. Let us consider a small graph with two cycles (figure 6).

In the Appendix 4.3 there is a Qiskit code simulating a quantum computer in order to find a 2-coloring. As expected it finds all the possible solutions (two, here).

Last question : what happens if there is no solution at all? It is easy to see with the 'triangle' graph (3 nodes, 3 edges). In our Qiskit code, just define the graph by

4. Here I used Qiskit, a specific Python extension, to simulate the quantum computer

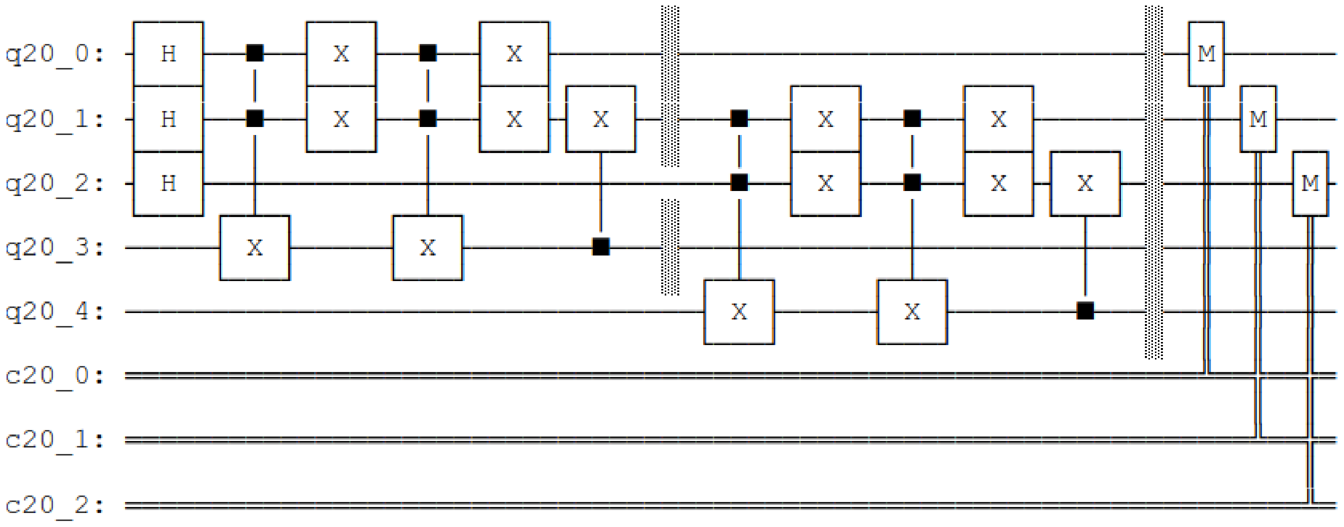


Figure 4: Circuit for the 3-nodes graph $0 \leftrightarrow 1 \leftrightarrow 2$.

```
end1=[0,1,2]
end2=[1,2,0]
```

and run it. The given 'solution' is {'100': 498, '011': 526}, which is of course wrong.

So, when there is no solution our quantum circuit proposes an erroneous one. It means we have to always check. It can be easily done in quadratic time or even less with a simple algorithm on a classical computer, but it is not very satisfying and, more important, this method is quite difficult to generalise to more than two colours (see the Q-method section 4 though).

3 *NK* method for *K* colours

The method used for two qubits could be easily generalised by using qudits (qunits), for such objects can directly represent K values. ([2]). However we do assume here you do not have access to such simulators, not even speaking of real computers using this technology.

So we use a coding that requires more qubits than strictly necessary but is valid for any K-colouring and for which it is not very complicated to automatically build a quantum circuit. The idea is to assign a set of K qubits to each node. So, we define a colouring matrix C of N lines and K columns. We assume that $N \geq K$. The graph is defined by its adjacency matrix G. Clearly there is enough information in these two binary matrices to decide if a colouring is valid or not. Actually we have to ensure two criteria before measure:

1. On each line of C there is one 1 and $K-1$ zeros, for obviously a given node can have just one colour.
2. Thanks to manipulations on C and G we can keep only valid colourings (i.e. valid C matrix).

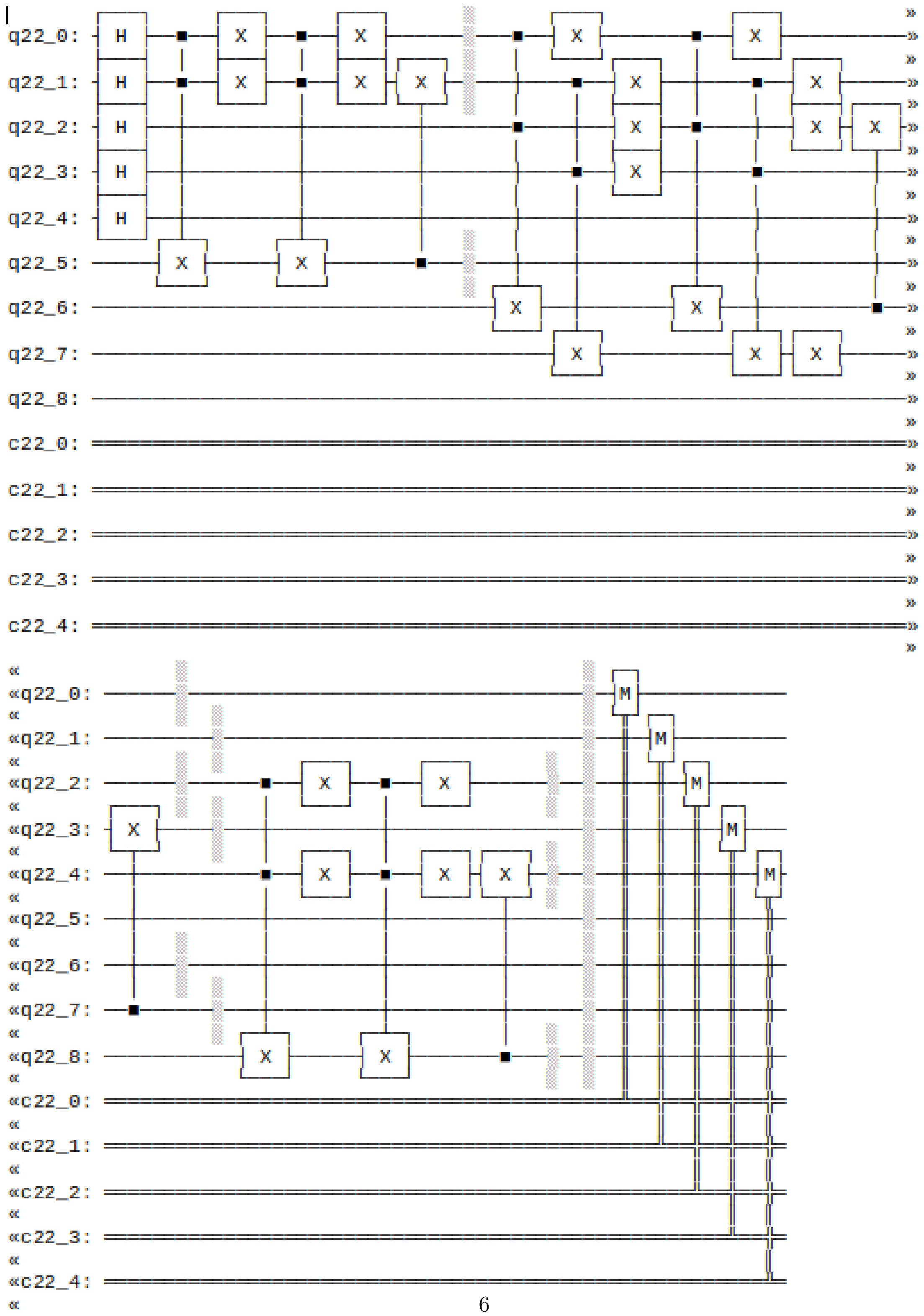


Figure 5: Circuit for the graph of the figure 1. It can be automatically generated.

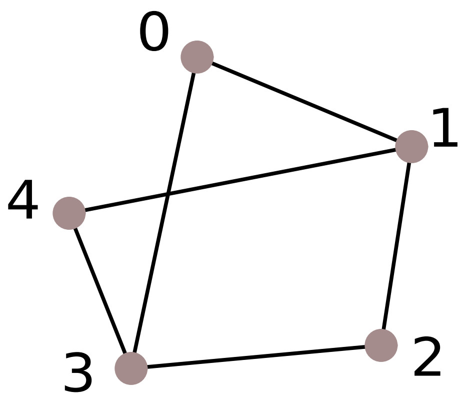


Figure 6: A 5 nodes graph with 2 cycles.

3.1 A bit of algebra

If you do not like maths, you can directly skip to the Quantum approach section 3.2. This mathematical part is here just for 'historical' reason, for it helped me to find a way to build the circuits, but is not really necessary to understand them.

We define

$$\Sigma = C \oplus C \tag{1}$$

$$G_K = \sqcup_K G \tag{2}$$

$$\Gamma = G_K \odot \Sigma \tag{3}$$

where $\oplus$ is the outer sum, $\sqcup_K$ the "horizontal" concatenation operator (K times), and $\odot$ the element-wise product.

Then the indicator is

$$v_{C,G} = \max(\Gamma) \tag{4}$$

The validity of the colouring C for the graph G is given by the condition,

$$v_{C,G} < 2 \tag{5}$$

This condition is equivalent to

$$v_{C,G} = 0 \vee v_{C,G} = 1 \tag{6}$$

which is easier to check when manipulating qubits. One can also use a modified outer sum in order to ensure Γ is binary. The condition is then simply

$$\Gamma = \mathbf{0} \tag{7}$$

where $\mathbf{0}$ is a null matrix.

The algorithm in Octave/Matlab© language

```
Sigma=outerSum(C);
% Variant:
% Sigma=max(0,Sigma-1);
GK=repmat(G, K,1);
Gamma=GK.*Sigma;
valid=max(Gamma(:))<2 ;
% Variants:
% valid=max(Gamma(:))<1; % valid=Gamma=zeros(size(C));
...
function Sigma=outerSum(C)
 % This is a simplified version.
 % For the normal complete version there are two matrices as inputs.
[~,K]=size(C);

    Sigma=[];
        for k=1:K
           Ck=meshgrid(C(:,k));
           Ck=Ck+Ck';
           Sigma=[Sigma Ck];
        end
end
```

Proof

1. The only possible values in Σ are 0, 1 and 2.
2. The only possible values in Γ are 0, 1 and 2.
3. Let us consider $\Sigma(n,m)$. We have $m=(k-1)N+j$, with $j\in[1,2,\cdots,N]$.
4. If $\Sigma(n,m)=0$ it means neither node n nor node j have the colour k.
5. If $\Sigma(n,m)=1$ it means node n has the colour k, and node j has another colour, or vice versa.
6. If $\Sigma(n,m)=2$ it means nodes n and j have the colour k (and this is of course the case for $n=j$).
7. This value 2 is "eliminated" in Γ iif $\Gamma(n,m)=0$, i.e. $G(n,j)=0$, which means "no edge between nodes n and j".
8. So, if there is no value 2 in Γ the colouring C is valid.

Example of valid 3-colouring (see the figure 7)

$$G=\begin{pmatrix}0&0&1&1\\0&0&1&1\\1&1&0&1\\1&1&1&0\end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

$$\sqcup \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\sqcup \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

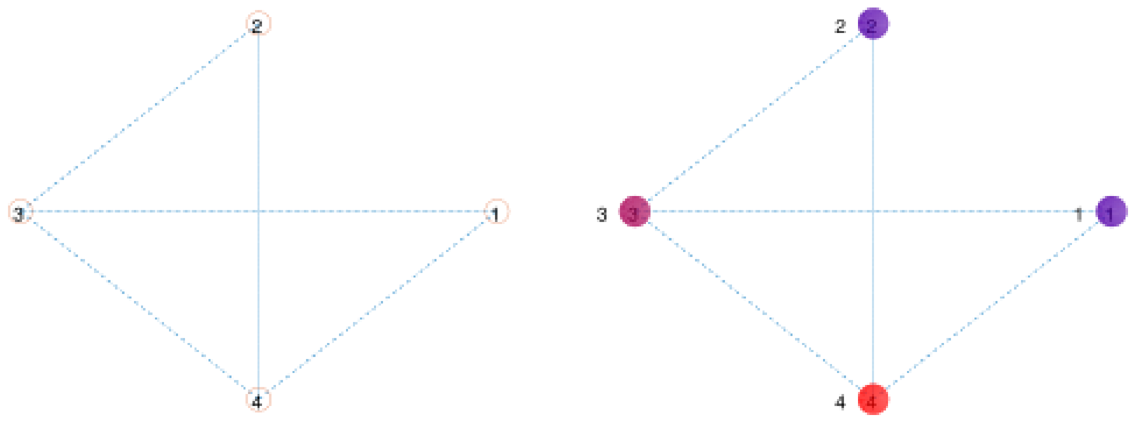
$$= \begin{pmatrix} 2 & 2 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 2 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 2 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$

$$G_K = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Note how the 2 values in Σ are "eliminated" in Γ thanks to the element wise product by G_K. Also it is worth carefully examining how Σ is built in order to find the indices of the values 2 in it. For each colour k:

- Duplicate "horizontally" N times the corresponding column of the matrix C, in order to build a square matrix Σ_k.



(a) The graph

(b) A possible 3-coloring

Figure 7: Colouring a 4 nodes 5 edges graph.

- Add it to its transpose.
- If $k > 1$,concatenate "horizontally" the result with the previous one.

So, a 2 is generated iif we have

$$\Sigma_k\left(i,j\right) = \Sigma_k\left(j,i\right) = 1 \tag{8}$$

But $\Sigma_k\left(i,j\right) = \Sigma_k\left(i,1\right)$ and $\Sigma_k\left(j,i\right) = \Sigma_k\left(j,1\right)$, which means that the same colour k is assigned to the nodes i and j. So, after a long detour the rule is very simple:

Theorem 1. *There is a value 2 in Σ iff two nodes have the same colour.*

It will be useful to build a quantum circuit. Note that it implies there are at least N values 2, because the N cases $i = j$.

Example of invalid 3-colouring

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

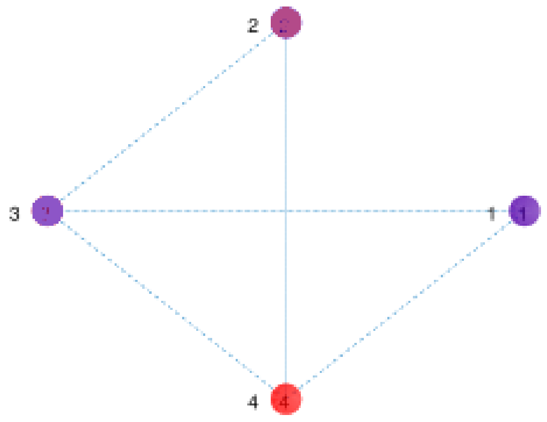


Figure 8: An invalid 3-colouring. Nodes 1 and 3 have the same colour, but there is an edge between them.

3.2 Quantum approach, *NK* method

The number of qubits to define a colouring matrix is NK (hence the name of this general method). In practice, to build a quantum circuit the algebraic approach is just a guideline and we do not have to really follow it. Actually the steps can be:

- Define the graph thanks to qubits[5].
- Generate a superposition of all possible colourings.
- Apply a filter to keep just the superposition of all colouring matrices, as defined above (one 1, and only one, in each line).
- Identify the pairs of nodes that have the same colour.
- Compare to the graph. If there is an edge between two nodes of same colour, "destroy" the colouring matrix (there is a trick here, see below)
- Measure (and display/save solutions).

Each of these steps can be performed by a quantum sub-circuit. A complete Qiskit code is given in the Appendix 4.3

Remark 1. The solutions found may contain some with less than K colours. So it is worth checking them. It can easily be done on a classical computer.

3.2.1 Define the graph

The $\frac{N(N-1)}{2}$ possible edges of the graph (supposed undirected, but it could easily be generalised) are represented by the same number of qubits, $|0>$ (no edge) or $|1>$ (edge).

5. We use qubits to define the graph just to have a pure quantum representation, and to be able to apply CX gates on graph elements. But actually, with a hybrid approach, we could use classical binary values.

3.2.2 Generate a superposition of all possible colorings

We just have to apply the Hadamard gate to NK qubits q_i. We can suppose they represent a $N \times K$ matrix.

$$Q = \begin{pmatrix} q_0 & q_1 & \cdots & q_{K-1} \\ q_K & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ q_{(N-1)K} & \cdots & \cdots & q_{NK-1} \end{pmatrix}$$

3.2.3 Apply a filter

Keeping just a superposition of all coloring matrices. We do not care of the probabilities of these matrices. The number of possible matrices is obviously K^N.

The Qiskit code below gives for example the following 27 results after 1000 shots for 3 nodes et 3 colours:

{'010100100': 99, '010001010': 6, '001100100': 36, '010010001': 6, '010010010': 9, '001001010': 4, '100001010': 24, '001100010': 22, '001001100': 7, '100010001': 17, '100001001': 11, '100010100': 99, '001010100': 22, '100001100': 52, '001100001': 4, '010001100': 23, '100010010': 47, '100100001': 49, '100100100': 255, '001010001': 2, '010010100': 41, '100100010': 100, '010001001': 4, '001001001': 2, '010100001': 17, '001010010': 8, '010100010': 34}

Each binary string represents a colouring matrix. For example 010100100 is

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

meaning[6]

- color 2 for node 1
- color 1 for node 2
- color 1 for node 3

Note that to simplify the circuit (see the figure 9) we reinitialise many times K ancillary qubits. And of course, when put at the beginning of a more complete code, the Measure and Execute sections have to be removed.

6. We should in fact read the binary string from right to left but it does not matter here.

Qiskit code

```
#---------------------------- Generate all colouring matrices
'''
1 line for each node
1 column for each colour
one and only one 1 in each line, other values 0
'''
from qiskit import *
from qiskit import Aer
backend_sim = Aer.get_backend('qasm_simulator')
nNodes=3
nColors=3 # nColors <= nNodes
nc=nColors+1
nqbits=nc*nNodes
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
c=ClassicalRegister(nColors*nNodes)
qc = QuantumCircuit(q,c)
# Initialisation
s=0
for n in range(nNodes):
    for k in range(nColors):
        qc.h(s+k)
    s=s+nc
# Constraints
s=0
for n in range(nNodes):
for k in range(nColors-1):
    for l in range(k+1,nColors):
        # Eliminate 11
            qc.ccx (s+k,s+l,s+nColors)
            qc.cx (s+nColors,s+k)
    qc.reset(s+nColors)
    # Eliminate 0* (no colour assigned to the node n)
    for k in range(nColors):
        qc.x(s+k)
    cb=list(range(s,s+nColors))
    qc.mcx (cb,s+nColors)
    for k in range(nColors):
        qc.x(s+k)
        qc.cx (s+nColors,s+nColors-1)
        qc.reset(s+nColors)
    s=s+nc
    # Measure
    cb=0
    for n in range(nNodes):

        s=n*(nColors+1)
        for k in range(nColors):

            qb=s+k
            qc.measure(qb,cb)
            cb=cb+1

    # Execute the circuit on the qasm simulator.
    job = execute(qc, backend_sim, shots=1000)
    # Grab the results from the job.
    result_sim = job.result()
    counts = result_sim.get_counts(qc)
    print(counts)
    qc.draw()
```

3.2.4 Identify the pairs of nodes that have the same colour

There are $\frac{N(N-1)}{2}$ possible pairs so we need this number of ancillary qubits. Thanks to CCX gates, each ancillary qubit is set to $|1>$ if the two nodes have the same colour, and kept to $|0>$ else.

3.2.5 Compare to the graph

Then we apply 4-qubits MCX gates. One qubit represents the state of a pair of nodes (same colour or not), another one the state of the corresponding edge (exists or not), and a third one is just used as intermediate. This is a trick to "eliminate" the invalid colourings: the forth one (that is a part of the colouring matrix) is set to $|0>$ in case the first two qubits are in state $|1>$. See the principle on the figure 10. Of course it could be simplified if you have a customised gate like a 3-controlled reset.

3.2.6 Measure and propose solutionsn

We measure the qubits representing the colouring matrix, and we keep only the bit strings that are not 0^*. Then we can reshape the bit strings into colouring matrices.

3.3 Examples

3.3.1 3 nodes, 3 colours

The graph is just a "triangle": three nodes and an edge between each pair of nodes. We are looking for a 3-colouring.

As a result a Qiskit simulation gives

{'0 100001010': 30, '0 010001100': 23, '0 001100010': 26, '1 000000000': 463, '0 100010001': 19, '0 000000000': 394, '0 001010100': 23, '0 010100001': 22}

We ignore the null binary strings, so we have, as expected, six valid colorings. For example 100001010 is in fact the coloring matrix

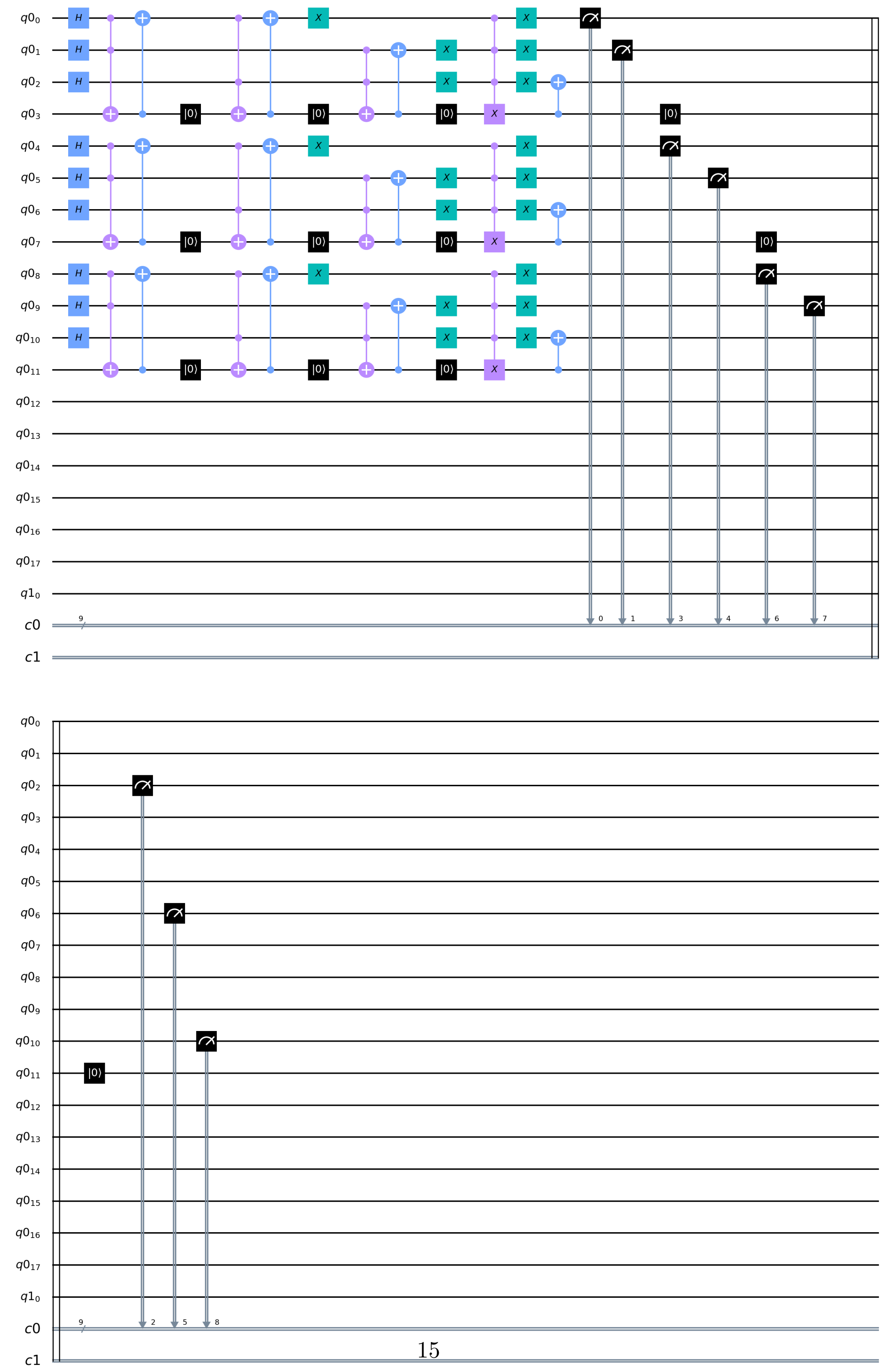


Figure 9: Circuit to generate all coloring matrices (3 nodes, 3 colours)

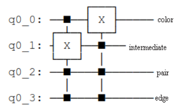


Figure 10: “Destroying” a colouring if two nodes have the same colour (q0_2=|1>) and an edge between them (q0_3=|1>).

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

that means

- color 1 for node 1
- color 3 for node 2
- color 2 for node 3

3.3.2 3 nodes, 2 colours

We keep the same triangular graph, but try to find a 2-coloring. It is impossible, the simulation, after 1000 shots, gives simply two 0* binary strings: {’0 000000’: 350, ’1 000000’: 650}

3.4 Complexity

The worst-case complexity of any general k-colouring algorithm on a classical (Türing) machine is exponential in the size of the graph, except for the cases $k \in \{0, 1, 2\}$. And of course we have the same problem when just using simulations of quantum computing. Actually, on my laptop our Qiskit program generates an error message (Insufficient memory for 29-qubits circuit) for the 3-coloring of a 4 nodes 5 edges graph. Nevertheless we can estimate the theoretical complexity.

3.4.1 Number of qubits

We need:

- $\frac{N(N-1)}{2}$ qubits for the graph. As explained in 3.2, this could be not taken into account in a hybrid approach.
- NK qubits for the coloring matrix
- $\frac{N(N-1)}{2}$ qubits for the pairs of nodes
- $N + 1$ ancillary qubits
- $NK + 1$ classical bits for measures and conditional operations

As $K \leq N$ the complexity in terms of number of qubits is $O\left(N^2\right)$, even if some ancillary qubits are reset and reused.

Table 1: Gates used in the NK- method. For each gate the size of the equivalent unitary matrix is $2^w \times 2^w$.

	Number	Gate	Weight w
Defining the graph	$\frac{N(N-1)}{2}$	X	1
Initialisation	NK	H	1
Colouring matrix	$N\frac{K(K-1)}{2}$	C_2X	3
	$N\frac{K(K-1)}{2}$	CX	2
	NK	X	1
	NK	C_KX	K+1
	NK	X	1
	NK	CX	2
Check if same colour	$K\frac{N(N-1)}{2}$	C_2X	3
Compare to the graph	$2NK\frac{N(N-1)}{2}$	C_3X	4
Total worst case ($K = N$)	$N^4 + N\frac{N(N-1)}{2} + 8N^2$		

3.4.2 Number of gates

By reading the Qiskit code 4.3, we can build a table that counts the number of gates. We note C_iX the multicontrolled X gate with i control qubits (so the classical CCX gate is written C_2X). As we can see, the complexity in terms of number of gates is $O\left(N^4\right)$.

3.5 Discussion

Clearly we use too many qubits (NK) to describe a colouring. Theoretically, to code a K colouring we need only Q qubits so that

$$Q = N\left\lceil \frac{\ln(K)}{\ln(2)} \right\rceil$$

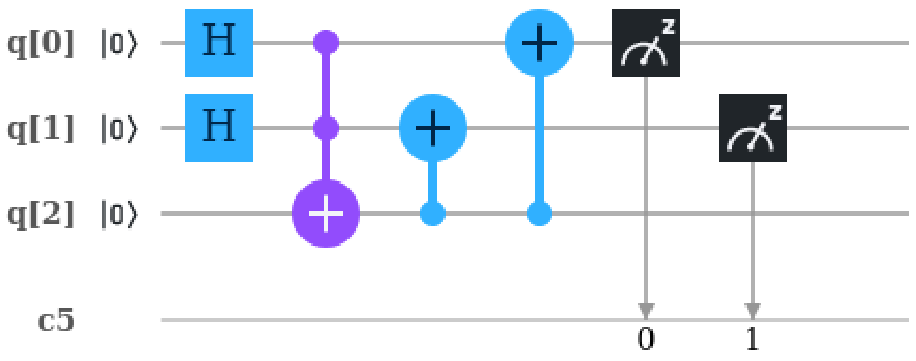
where $\lceil x \rceil$ means 'the smallest integer greater than or equal to x'. For $N = 4$ and $K = 3$ we have $NK = 12$ and $Q = 8$. And for the 3-colouring of a 10 nodes graph we have $NK = 30$ and $Q = 20$. For a given K the percentage gain is obviously constant (1/3 for $K = 3$), but grows rapidly in absolute value.

4 Using less qubits, Q method for 3-colouring

As discussed we could theoretically use less qubits than in the general method that requires a binary colouring matrix, but it would be more and more complicated when the number of colours increases. So we present here how to do just for a 3-coloring. First we will see how to use two qubits to code only three colours, and then a complete circuit (more precisely its Qiskit code).

4.1 Two qubits for three values

With two qubits, we have four possibilities, so before to build the complete circuit we have to use a sub-circuit that "eliminates" one of them. The four possible kets are |00>, |10>, |01> and |11>. Let us suppose that the three first ones are for coding the three colours.



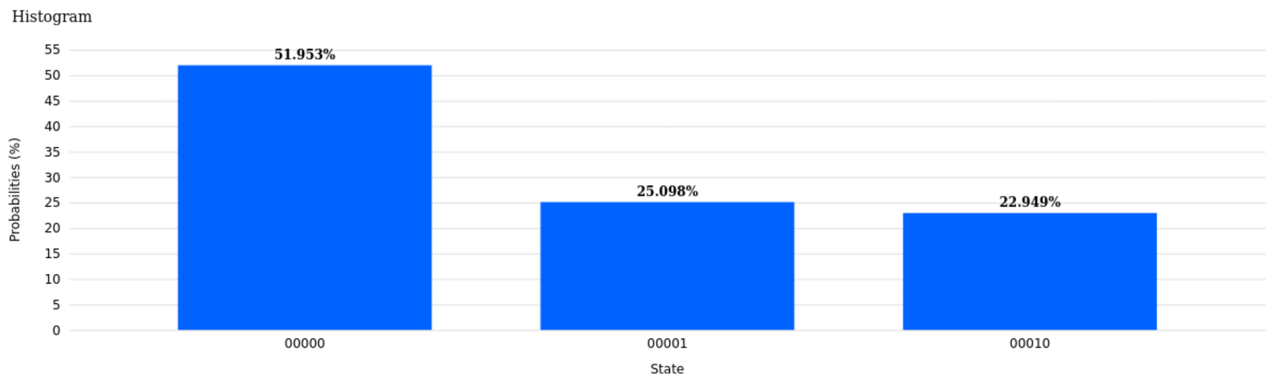


Figure 11: The output of this simple circuit is never |11 >. Probabilities estimated over 1000 shots, for the three possible outputs 00, 01 and 10.

We have to design a circuit whose input is the two qubits in any state and whose output (after measure) is never |11>. The simple one of the figure 11 does the job. Note that the probabilities of the three possible outputs are different. The scheme of the circuit can be summarised as follows:
|00> => |00>
|11> => |00>
|10> => |10>
|01> => |01>
As at the beginning the four probabilities are 1/4, it is clear that the three final ones will be 1/2, 1/4 and 1/4. In practice the real values after a given number of shots are of course a bit different.

Such a non-uniformity is not elegant but not important for our purpose. Just for fun you can see a 'perfect' (but complicated) method in the Appendix 4.3).

4.2 A complete circuit

Now, the main difference with the *NK* method is how to identify the pairs of nodes that have the same colour. As |11 > has been eliminated, we just have to check the three other possibilities. On the sub-circuit

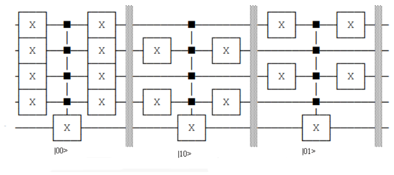


Figure 12: Q method, 3-colouring Sub-circuit to identify the pairs of nodes that have the same colour, either 00, 10, or 01.

of the figure 12, the fifth qubits is $|1>$ iif the two pairs of qubits code both one of these three colours.

The rest of the circuit is very similar to the one of the NK method, and in case of 'same colour and edge' we again set to $|0>$ all qubits that code the colouring. A complete Qiskit code is given in the Appendix 4.3.

Variants

Let us suppose we want to apply the Q method for $K > 3$. If $K = 2^k$ there is no need to "eliminate" any configuration. On the other hand, on the contrary, if $2^k < K < 2^{k+1}$ we have to eliminate $K - 2^k$ configurations. Designing the corresponding sub-circuit may be very difficult. A possible hybrid workaround, although not very satisfying, is the following:

1. Do not eliminate anything before the core of the circuit.
2. After measurement, remove (classical computing) the solutions that require more than K colours.

On the triangle graph we find after 1000 shots:

{'000110': 31, '011000': 29, '010010': 44, '001001': 37, '100100': 34, '100001': 30, '000000': 795}

For example '000110' is for (01, 10, 00), for we have to read it from right to left. It represents the three different colours of the three nodes. In this simplistic case the other solutions are just permutations of this one.

As less qubits are needed, all valid 3-colorings of the graph of the figure 7 can be found without memory error on my small laptop (it takes of course some time, though):

{'01001010': 6, '00011010': 9, '00000000': 940, '01100000': 11, '00100101': 5, '10000101': 14, '10010000': 15}

There are six valid colourings and the first one is (01, 01, 00, 10).

4.3 Complexity

Although the total number of needed qubits is smaller than the one for the NK method, we still need $\frac{N(N-1)}{2}$ qubits to define the graph. So the complexity is still $O\left(N^2\right)$.

Appendix

Qiskit for 2-colouring

```
#---------------------------- General code for 2-colouring
```

```
import numpy as np
from qiskit import *
%matplotlib inline
# Define the graph by giving the ends of each edge
end1=[0,1,2,3,1,3]
end2=[1,2,3,0,4,4]
#---
nNodes=max(end1+end2)+1
nEdges=len(end1)
nodes=np.arange(nNodes)
# Create a Quantum Circuit
q = QuantumRegister(nNodes+nEdges)
c=ClassicalRegister(nNodes)
circ = QuantumCircuit(q,c)
# Gates
circ.h(nodes)
k=nNodes
for n in range(nEdges):
    i=end1[n]
    j=end2[n]
    circ.ccx(i,j,k)
    circ.x(i);circ.x(j); circ.ccx(i,j,k); circ.x(i);circ.x(j)
    circ.cx(k,j)
    circ.barrier(i,j,k)
    k=k+1
#----------------------- Measure
circ.barrier(q)
circ.measure(nodes,c)
#----------------------- Run
from qiskit import Aer
# Use Aer's qasm_simulator
backend_sim = Aer.get_backend('qasm_simulator')
job_sim = execute(circ, backend_sim, shots=1024)
# Grab the results from the job.
result_sim = job_sim.result()
counts = result_sim.get_counts(circ)
print(counts)
#--------------- Drawing
circ.draw()
```

Qiskit for k-colouring, *NK* method

```
    # k-colouring of a graph
```

```
# N = number of nodes
# K = number of colours
'''
In this method we use a colouring matrix:
1 line for each node
1 column for each colour
one and only one 1 in each line, other values 0
It means we need at least NK qubits to describe such a matrix
'''
import numpy as np
from qiskit import *
%matplotlib inline
#from qiskit import Aer
#from qiskit.providers.aer import AerError, QasmSimulator
#backend_state = Aer.get_backend('statevector_simulator')
backend_sim = Aer.get_backend('qasm_simulator')
nNodes=3
nColors=3 # nColors <= nNodes
nc=nColors+1 # For each node, nColors qubits that will be measured,
    # + one ancillary
nn2=round((nNodes-1)*nNodes/2) # Number of pairs of different nodes
    # and of possible edges
sc=round(nc*nNodes) # Number of qubits to skip before the ancillary qubits
    # for pairs and edges
sg=round(nc*nNodes + nn2) # Beginning of the list of qubits that describe the graph
nqbits=sc + 2*nn2 # Total number of qubits
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
c=ClassicalRegister(nColors*nNodes) # To measure to "extract" the colouring matrices
qc = QuantumCircuit(q,c)
# Add the graph (binary list of nNodes*(nNodes-1)/2 elements,
# because the graph is symmetric and there is no i=>i edges.
# Set to |1> the qubits corresponding to an edge.
'''
3 nodes, needs 3 colors
0 1 1
1 0 1
1 1 0
=> 1 1 1
3 nodes, needs 2 colors
0 1 0
1 0 1
0 1 0
=> 1 0 1
2 nodes
```

```
2 n
0 1
1 0
=> 1
4 nodes, needs 3 colors
0 0 1 1
0 0 1 1
1 1 0 1
1 1 1 0
=> 0 1 1 1 1 1
'''
qc.x(sg) # Set to |1> iif there is an edge
qc.x(sg+1)
qc.x(sg+2)
#qc.x(sg+3)
#qc.x(sg+4)
#qc.x(sg+5)
# ---------------------------------------Generate a colouring matrix
# Initialisation
# Hadamard gate for qubits that represent the colouring matrix
s=0
for n in range(nNodes):
   for k in range(nColors):
      qc.h(s+k)
   s=s+nc
#-----------------------------
# Constraints
# A 1 and only one 1 in each of the coloring matrix
s=0
for n in range(nNodes):
   for k in range(nColors-1):
      for l in range(k+1,nColors):
        # Eliminate 11
        qc.ccx (s+k,s+l,s+nColors)
        qc.cx (s+nColors,s+k)
        qc.reset(s+nColors)
   # Eliminate 0* (no colour assigned to the node n)
   for k in range(nColors):
      qc.x(s+k) # if 0 => 1.

        # Not needed if you can use a negative multicontrolled gate
      cb=list(range(s,s+nColors) )
      qc.mcx (cb,s+nColors)
   for k in range(nColors):
      qc.x(s+k) # if 1 => 0
      qc.cx (s+nColors,s+nColors-1)
      qc.reset(s+nColors)
    s=s+nc
# At this point, if we measure,
# we find colouring matrices (nNodes lines, nColors columns),
# one and only one 1 in each line (a node does have a colour, and only one)
print('end of colouring matrices')
# Switch the ancillary qubits corresponding to pairs of nodes
# that have the same colour
for k in range(nColors):
    s=nc*nNodes
    for n1 in range(nNodes-1):
      for n2 in range(n1+1,nNodes):

        n11=nc*n1+k # If q[n11]=|1> it means the node n1 has the color k
        n22=nc*n2+k # If q[n22]=|1> it means the node n2 has the color k
        qc.ccx(n11,n22,s) # If same color k, set s to |1>.
          # Notice it can happens at most for one k
        s=s+1
#print([n11,n22])
# At this point, if we measure the (nNodes-1)*nNodes)/2 ancillary qubits,
# we get binary strings, in which 1 means "same color" for n1 and n2
print('end of pairs of nodes')
# Compare to the graph.
for n in range(sc,sc+nn2): # For each pair of nodes

    # If same color and there is an edge "destroy" (set to |0*>) the colouring
    for node in range(nNodes):
      qnode=nc*node
      qnc=qnode+nColors
      for k in range(nColors):
        cb=[n,n+nn2,qnode+k]
        qc.mcx (cb,qnc)
        cb=[n,n+nn2,qnc]
        qc.mcx (cb,qnode+k)
      qc.reset(qnc)
print('end of compare to graph')
# Measure (only the qubits describing the colouring matrices)
cb=0
for n in range(nNodes):
    s=n*(nColors+1)
    for k in range(nColors):
```

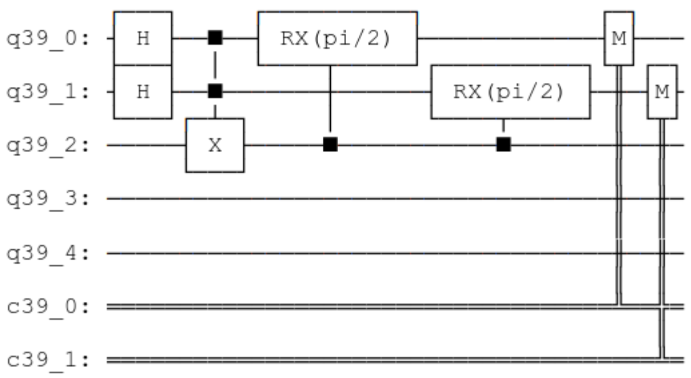


Figure 13: Rotations of the two qubits. They seriously decrease the probability of 11.

```
            qb=s+k
            qc.measure(qb,cb)
            cb=cb+1
    print('end of measures')
    # Quick on small graphs, but memory error for 4 nodes, 3 colours
    job = execute(qc, backend_sim,shots=1000)
    result=job.result()
    #print(result)
    print('end of execute')
    # Grab the results from the job.
    counts = result.get_counts(qc)
    print(counts)
    print('The solutions are given by the strings that are not 0*')
    print('Please check: some of them may need LESS than', nColors,'colours')
```

Two qubits for three values - Perfect method

Methods 1 and 2

First, we can rotate the two qubits. With the small circuit of the figure 13, we get for example the following repartition after 1000 shots: {'11': 55, '10': 302, '01': 340, '00': 303}. Not perfect, but the probability of 11 is already quite small.

Now, to set it to zero, we can complete the circuit, as on the figure 14. After the rotations, the case 11 is "transformed" into 00. After 1000 shots we get {'10' : 319, '01': 312, '00': 369}. The probability of 00 is of course a bit too high, but the repartition is nevertheless not bad. Note that we could use just one ancillary bit, by reinitialising to |0> after the rotations. However, such a reinitialisation may be difficult on a real quantum computer, that is why we use two ancillary bits.

It may be useful to visualize the qubits on Bloch spheres, step by step, and also to consider the evolution of the state vector (whose size is $2^4 = 16$). Note it can be done only on a simulation. On a real quantum computer observing the qubits would destroy them (more precisely would definitely project them on 0 or

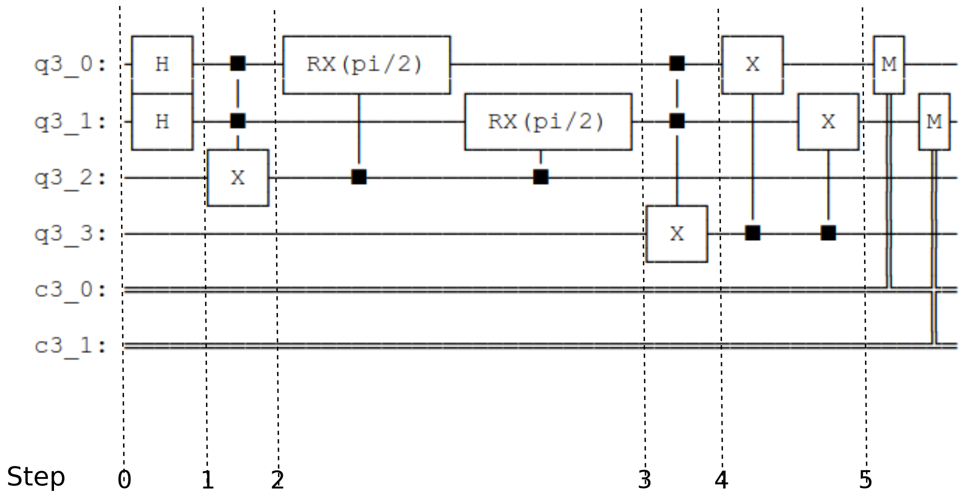


Figure 14: Elimination of the 11 case.

1).

Let us say that the coefficients of the qubit q_i on the base ($|0>$, $|1>$) are (α_i, β_i). Therefore the general form of the state vector of a 4-qubits system is

$$\Psi = \begin{array}{l} (\alpha_3\alpha_2\alpha_1\alpha_0, \\ \alpha_3\alpha_2\alpha_1\beta_0, \\ \alpha_3\alpha_2\beta_1\alpha_0, \\ \alpha_3\alpha_2\beta_1\beta_0, \\ \alpha_3\beta_2\alpha_1\alpha_0, \\ \alpha_3\beta_2\alpha_1\beta_0, \\ \alpha_3\beta_2\beta_1\alpha_0, \\ \alpha_3\beta_2\beta_1\beta_0, \\ \beta_3\alpha_2\alpha_1\alpha_0, \\ \beta_3\alpha_2\alpha_1\beta_0, \\ \beta_3\alpha_2\beta_1\alpha_0, \\ \beta_3\alpha_2\beta_1\beta_0, \\ \beta_3\beta_2\alpha_1\alpha_0, \\ \beta_3\beta_2\alpha_1\beta_0, \\ \beta_3\beta_2\beta_1\alpha_0, \\ \beta_3\beta_2\beta_1\beta_0) \end{array}$$

Each component Ψ_j can be seen as "corresponding" to a binary string we could find after measurement. By squaring a component we find the probability of this string. To remember which string corresponds to which component, just replace α by 0, β by 1, and ignore the subscripts. For example, squaring the fourth component give the probability of the string 0011.

At the very beginning (step 0), all qubits are in the state $|0>$. As $|0>=(1,0)$ it implies $\Psi_1 = 1$, and the others 0.

Step 1 (after Hadamard gates)

String	Probability
0000	$\frac{1}{4}$
0001	$\frac{1}{4}$
0010	$\frac{1}{4}$
0111	$\frac{1}{4}$

Table 2: Method 1, step 2. Non null probabilities of binary strings

The qubits 0 and 1 are in the classical “intermediate” state $\frac{|0>+|1>}{\sqrt{2}}$, the two ancillary bits remain unmodified. The Qiskit state vector is

[0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j 0. +0.j]

which is exactly what we expected

$$\begin{cases} \Psi & = \frac{|0>+|1>}{\sqrt{2}} \otimes \frac{|0>+|1>}{\sqrt{2}} \otimes |0> \otimes |0> \\ & = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right) \end{cases}$$

We indeed have

$$\begin{gathered} \alpha_0 = \frac{1}{\sqrt{2}} \\ \beta_0 = \frac{1}{\sqrt{2}} \\ \alpha_1 = \frac{1}{\sqrt{2}} \\ \beta_1 = \frac{1}{\sqrt{2}} \\ \alpha_2 = 1 \\ \beta_2 = 0 \\ \alpha_3 = 1 \\ \beta_3 = 0 \end{gathered}$$

Step 2 (after the Toffoli gate on qubit 2)

The qubit 3 is still |0>, qubit 2 is modified, but strangely qubits 0 and 1 are also slightly modified. The Qiskit state vector is

[0.5-2.77555756e-17j 0.5-2.77555756e-17j 0.5-2.77555756e-17j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0.5-5.55111512e-17j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j 0. +0.00000000e+00j]

We see very small values coming from nowhere: the Qiskit simulation is not perfect. The theoretical state vector is in fact

$$\Psi = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, \frac{1}{2}, 0, 0, 0, 0, 0, 0, 0, 0\right)$$

which is indeed we find by setting to zero the small values of the simulated state. The non null probabilities are given in the table 2.

Step 3 (after the two controlled rotations)

The rectified simulated state vector is

String	Probability
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
0111	$\frac{1}{16}$

Table 3: Method 1, step 3. Non null probabilities of binary strings

String	Probability
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
1111	$\frac{1}{16}$

Table 4: Method 1, step 4. Non null probabilities of binary strings

$$\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},0,-\frac{1}{4},-\frac{1}{4}i,-\frac{1}{4}i,\frac{1}{4},0,0,0,0,0,0,0,0\right)$$

Step 4 (after the Toffoli gate on qubit 3)
We find

$$\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},0,-\frac{1}{4},-\frac{1}{4}i,-\frac{1}{4}i,0,0,0,0,0,0,0,0,\frac{1}{4}\right)$$

Step 5 (after the two controlled NOT gates)

$$\left(\frac{1}{2},\frac{1}{2},\frac{1}{2},0,-\frac{1}{4},-\frac{1}{4}i,-\frac{1}{4}i,0,0,0,0,0,\frac{1}{4},0,0,0\right)$$

Note that, as expected, no sequence xx11 has a non null probability. We will measure the two first qubits. So the probabilities are
for 00, $\frac{1}{4}+\frac{1}{16}+\frac{1}{16}=\frac{6}{16}=0.375$
for 01 and 10, $\frac{1}{4}+\frac{1}{16}=\frac{5}{16}=0.3125$.
Let us run again the circuit, but with 100 000 shots. We find {'00': 37508, '10': 31162, '01': 31330 }. The estimated probabilities are therefore very near to the theoretical ones.

String	Probability
0000	$\frac{1}{4}$
0010	$\frac{1}{4}$
0001	$\frac{1}{4}$
0100	$\frac{1}{16}$
0101	$\frac{1}{16}$
0110	$\frac{1}{16}$
1100	$\frac{1}{16}$

Table 5: Method 1, step 5. Non null probabilities of binary strings

Method 3, theoretically perfect

Another possible way is to consider the quantum state Ψ_0 after the Hadamard gates, and the final one $\Psi_1$we want. On the base $(|00>, |00>, |10>, |01>, |11>)$ we have (remember that the norm must be always equal to 1):

$$\left\{ \begin{array}{lcl} \Psi_0 & = & \frac{1}{2}(1,1,1,1) \\ \Psi_1 & = & \frac{1}{\sqrt{3}}(1,1,1,0) \end{array} \right.$$

Now, we have to find an unitary matrix U so that (t is for "transpose")

$$U \times \Psi_0^t = \Psi_1^t$$

We know that it must be a 4×4 rotation matrix, whose general form is

$$U = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

under the constraint $det(U) = 1$. Some simple algebraic manipulations give us

$$\left\{ \begin{array}{lcl} a & = & \frac{\sqrt{3}}{2} \\ b & = & -\frac{\sqrt{3}}{6} \\ c & = & \frac{\sqrt{3}}{6} \\ d & = - & \frac{\sqrt{3}}{6} \end{array} \right.$$

$$U = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3 & 1 & -1 & 1 \\ -1 & 3 & 1 & 1 \\ 1 & -1 & 3 & 1 \\ -1 & -1 & -1 & 3 \end{pmatrix} \tag{9}$$

We can check that we indeed have $U \times U' = U' \times U = I$. But the main difficulty is how to design a quantum circuit whose equivalent unitary matrix is U. This is the well known problem of *quantum circuit synthesis*.

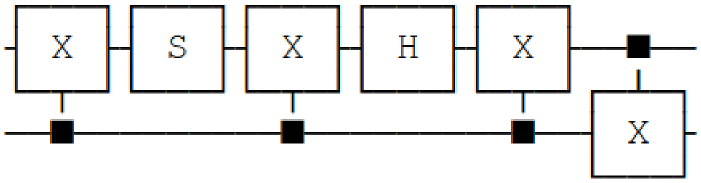


Figure 15: A circuit implementing the magic gate.

Can we do it by using only single qubits gates? If so, it would imply that we can find two 2×2 matrices $U_1$and U_2 so that $U_1 \otimes M_2 = U$, where $\otimes$ is the outer product. It is equivalent to a system of 16 equalities like $U_1(1,1)\,U_2(1,1) = a$, $U_1(1,1)\,U_2(1,2) = -b$, etc., but it is easy to see that they are incompatible [7]. So there is no solution and we have to use at least some two qubits gates.

Let us consider the entangler gate that is the two qubit gate which maps the computational basis into the magic basis (see[1][8]):

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i & 0 & 0 \\ 0 & 0 & i & 1 \\ 0 & 0 & i & -1 \\ 1 & -i & 0 & 0 \end{pmatrix}$$

Remember that magic basis of phase shifted Bell states is given by

$$\begin{cases} |m_1> & = & \frac{|00>+|11>}{\sqrt{2}} \\ |m_2> & = & \frac{i|00>-i|11>}{\sqrt{2}} \\ |m_3> & = & \frac{i|01>+i|10>}{\sqrt{2}} \\ |m_4> & = & \frac{|01>+|10>}{\sqrt{2}} \end{cases}$$

A possible implementation is given on the figure 15, where S is the phase gate $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$ and H the Hadamard gate.

For U is real orthogonal we know (this is a theorem) that there exist two 2-dimensional unitary matrices A_1 and A_2 so that

$$U = M'\left(A_1 \otimes A_2\right) M$$

It implies

$$A_1 \otimes A_2 = MUM' = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3-i & 1+i & 0 & 0 \\ -1+i & 3+i & 0 & 0 \\ 0 & 0 & 3-i & 1+i \\ 0 & 0 & -1+i & 3+i \end{pmatrix}$$

7. More technically, it is related to the Schmidt decomposition of the state Ψ_1. Not all Schmidt coefficients are strictly positive, meaning that the state is not separable but entangled.

8. *Warning* - The circuit given in the paper (figure 2) is wrong: inversion of the last two CNOT gates

Here we immediately have

$$A_1 \otimes A_2 = I \otimes A$$

where

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$A = \frac{1}{2\sqrt{3}} \begin{pmatrix} 3-i & 1+i \\ -1+i & 3+i \end{pmatrix}$$

So we now just have to find how implementing A. As any single qubit gate A can be decomposed as

$$A = e^{i\theta_0} \begin{pmatrix} e^{-i\theta_1/2} & 0 \\ 0 & e^{i\theta_1/2} \end{pmatrix} \begin{pmatrix} \cos\left(\frac{\theta_2}{2}\right) & -\sin\left(\frac{\theta_2}{2}\right) \\ \sin\left(\frac{\theta_2}{2}\right) & \cos\left(\frac{\theta_2}{2}\right) \end{pmatrix} \begin{pmatrix} e^{-i\theta_3/2} & 0 \\ 0 & e^{i\theta_3/2} \end{pmatrix}$$

$$A = \begin{pmatrix} \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1-\theta_3)/2} & -\sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1+\theta_3)/2} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1-\theta_3)/2} & \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1+\theta_3)/2} \end{pmatrix}$$

In terms of rotation gates

$$A = e^{i\left(\theta_0 - \frac{\theta_1}{2} - \frac{\theta_3}{2}\right)} R_z(\theta_1) R_y(\theta_2) R_z(\theta_3)$$

Remember that, for example

$$R_z(\theta_1) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta_1} \end{pmatrix}$$

hence the coefficients $e^{-\frac{\theta_1}{2}}$, etc.
By identification we have four complex equations to find the four θ values:

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1-\theta_3)/2} & = & \frac{3-i}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1-\theta_3)/2} & = & \frac{-1+i}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0-\theta_1+\theta_3)/2} & = & \frac{1+i}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right) e^{i(2\theta_0+\theta_1+\theta_3)/2} & = & \frac{3+i}{2\sqrt{3}} \end{cases}$$

They imply eight real ones, but only some sets of four are independent, for example

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right) \cos\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) & = & \frac{3}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right) \sin\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) & = & \frac{-1}{2\sqrt{3}} \\ \sin\left(\frac{\theta_2}{2}\right) \cos\left(\frac{2\theta_0-\theta_1+\theta_3}{2}\right) & = & \frac{1}{2\sqrt{3}} \\ \cos\left(\frac{\theta_2}{2}\right) \cos\left(\frac{2\theta_0+\theta_1+\theta_3}{2}\right) & = & \frac{3}{2\sqrt{3}} \end{cases}$$

They give

$$\begin{cases} \cos\left(\frac{\theta_2}{2}\right) & = \pm & \frac{\sqrt{5}}{\sqrt{2}\sqrt{3}} \\ \sin\left(\frac{2\theta_0-\theta_1-\theta_3}{2}\right) & = & \pm\frac{1}{\sqrt{2}\sqrt{5}} \\ \cos\left(\frac{2\theta_0-\theta_1+\theta_3}{2}\right) & = & \mp\frac{1}{\sqrt{2}} \\ \cos\left(\frac{2\theta_0+\theta_1+\theta_3}{2}\right) & = & \pm\frac{3}{\sqrt{2}\sqrt{5}} \end{cases}$$

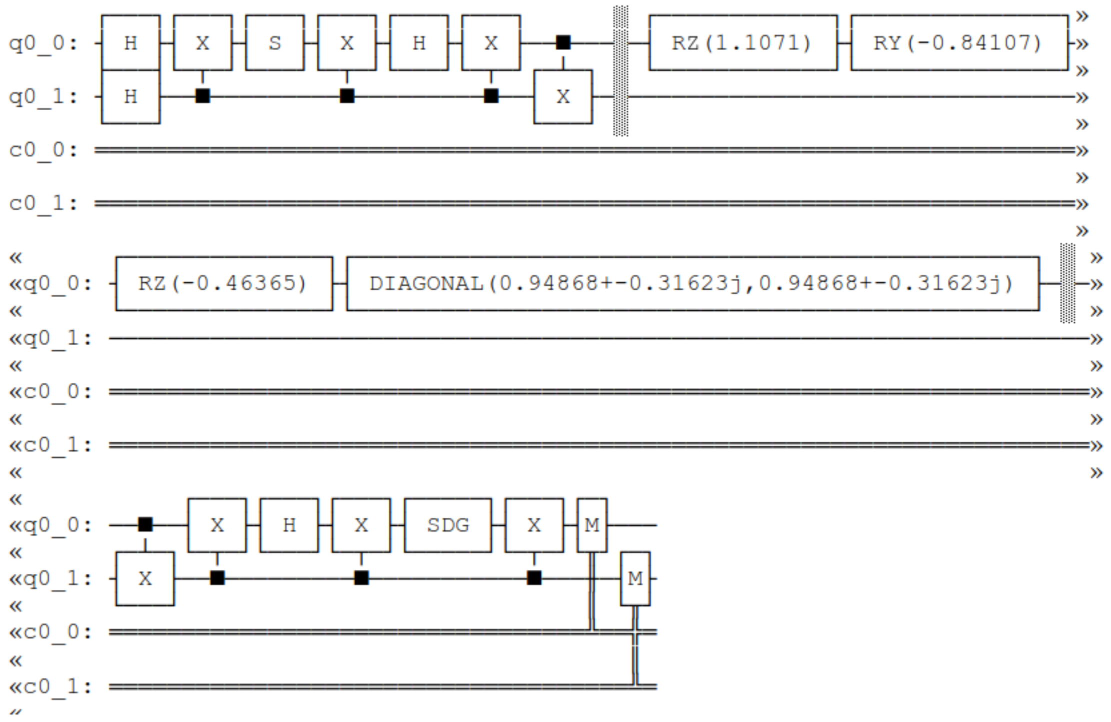


Figure 16: Exact method

Let us define

$$\begin{cases} \varphi_0 & = & -2\arcsin\left(\frac{1}{\sqrt{2}\sqrt{5}}\right) \\ \varphi_1 & = & 2\arccos\left(\frac{1}{\sqrt{2}}\right) \\ \varphi_2 & = & 2\arccos\left(\frac{\sqrt{5}}{\sqrt{2}\sqrt{3}}\right) \\ \varphi_3 & = & 2\arccos\left(\frac{3}{\sqrt{2}\sqrt{5}}\right) \end{cases}$$

A solution for the θ angles is then

$$\begin{cases} \theta_0 & = & \frac{\varphi_0+\varphi_3}{4} \\ \theta_1 & = & \frac{\varphi_3-\varphi_1}{2} \\ \theta_2 & = & -\varphi_2 \\ \theta_3 & = & \frac{\varphi_1-\varphi_0}{2} \end{cases}$$

Note that $\theta_0 = 0$, so we can ignore it. The gate corresponding to the coefficient $e^{-i\left(\frac{\theta_1}{2}-\frac{\theta_3}{2}\right)}$ is simply a diagonal gate with this value on the diagonal. The final circuit is given on the figure 16. It is seriously more complicated than the method 2 of the figure 14 because not only we have to perform the three rotations, but also to frame them by the circuits of M' and M.

With Qiskit simulation and 100 000 shots we find {'00': 33458, '01': 33200, '10': 33342}.

Qiskit for 3-coloring, *Q* method

```
#------------------------- Q method, Generate all 3-coloring
'''
N= number of nodes
K= number of colors
Each color is coded by at most Q=ceil(ln(K)/ln(2)) qubits
The main difficulty is to "eliminate", if any, Q-K states
Example, K=3
needs 2 qubits, but it implies 4 states 00, 10, 01, 11
so we use a circuit that eliminates 11
'''
import numpy as np
from qiskit import *
%matplotlib inline
from qiskit.quantum_info.operators import Operator
from qiskit import Aer
backend_sim = Aer.get_backend('qasm_simulator')
nNodes=3
nColors=3 # nColors <= nNodes. WARNING, only 3 with this code
nqcode=2 # WARNING, only 2, for nColors=3
nn2=round((nNodes-1)*nNodes/2) # Number of pairs of different nodes and of possible edges
sc=round(nqcode*nNodes) # # Beginning of the list of qubits that describe the pairs of nodes
sg=round(sc + nn2) # Beginning of the list of qubits that describe the graph
nqbits=sg + nn2+2 # Total number of qubits
# Create a Quantum Circuit
q = QuantumRegister(nqbits)
c=ClassicalRegister(nqcode*nNodes) # To measure to "extract" the coloring
qc = QuantumCircuit(q,c)
# Add the graph (binary list of nNodes*(nNodes-1)/2 elements, because the graph is symmetric
# and there is no i=>i edges. Set to |1> the qubits corresponding to an edge.
'''
Graphs
3 nodes, needs 3 colors
0 1 1
1 0 1
1 1 0
=> 1 1 1
4 nodes, needs 3 colors
0 0 1 1
0 0 1 1
1 1 0 1
1 1 1 0
=> 0 1 1 1 1 1
'''
```

```
qc.x(sg) # Set to |1> iif there is an edge
qc.x(sg+1)
qc.x(sg+2)
#qc.x(sg+3)
#qc.x(sg+4)
#qc.x(sg+5)
# Initialisation
# Hadamard gate for qubits that represent the coloring
s=0
for n in range(nNodes):
    for k in range(nqcode):
        qc.h(s)
        s=s+1
# Eliminate |11>
remov_op = Operator([[3, 1, -1, 1],
[-1, 3, 1, 1],
[1, -1, 3, 1],
[-1, -1, -1, 3]]/(2*np.sqrt(3)))
# We can also use a sub-circuit, as described in the Appendix 4.3
for n in range(nNodes):
    q1=n*nqcode
    q2=q1+1
    qc.unitary(remov_op, [q1, q2], label='remov 11')
# At this point, if we measure, we find 3 possible colors for each node: 00, 10, 01
print('end of coloring')
# Switch the ancillary qubits corresponding to pairs of nodes that have the same color
q4=sc # Ancillary qubit to use
for n1 in range(nNodes-1):
    q0=2*n1
    q1=q0+1
    for n2 in range(n1+1,nNodes):
        q2=2*n2
        q3=q2+1
        # Case |00>
        qc.x(q0);qc.x(q1);qc.x(q2);qc.x(q3)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q0);qc.x(q1);qc.x(q2);qc.x(q3)
        qc.barrier()
        # Case |10>
        qc.x(q1);qc.x(q3)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q1);qc.x(q3)
        qc.barrier()
        # Case |01>
        qc.x(q0);qc.x(q2)
        qc.mcx([q0,q1,q2,q3],q4)
        qc.x(q0);qc.x(q2)
        qc.barrier()
        q4=q4+1 # Will use the next ancillary qubit
# At this point, if we measure the (nNodes-1)*nNodes)/2 ancillary qubits,
# we get binary strings,
# in which 1 means "same color" for n1 and n2
print('end of pairs of nodes')
# Compare to the graph.
qnc=nqbits-2
for n in range(sc,sc+nn2): # For each pair of nodes


    # If same color and there is an edge "destroy" (set to |0*>) the coloring
    qc.ccx (n,n+nn2,qnc) # Set to |1> if same color and edge
    for code in range(sc):
        qc.ccx(code,qnc,qnc+1) # Set code to |0> if qnc is |1> ...
        qc.cx(qnc+1,code) # ...
        qc.reset(qnc+1)
        qc.reset(qnc)
print('end of compare to graph')
# Measure (only the qubits describing the coloring)
cb=0
for code in range(sc):
    qc.measure(code,cb)
    cb=cb+1
print('end of measures')
# Execute the circuit
result=job.result()
#print(result)
print('end of execute')
# Grab the results from the job.
counts = result.get_counts(qc)
print(counts)
print('The solutions are given by the strings that are not 0*')
print('Please check: some of them may need LESS than', nColors,'colours')
#qc.draw()
#qc.draw(filename='circuit_triangle_3_colors, Q method')
```

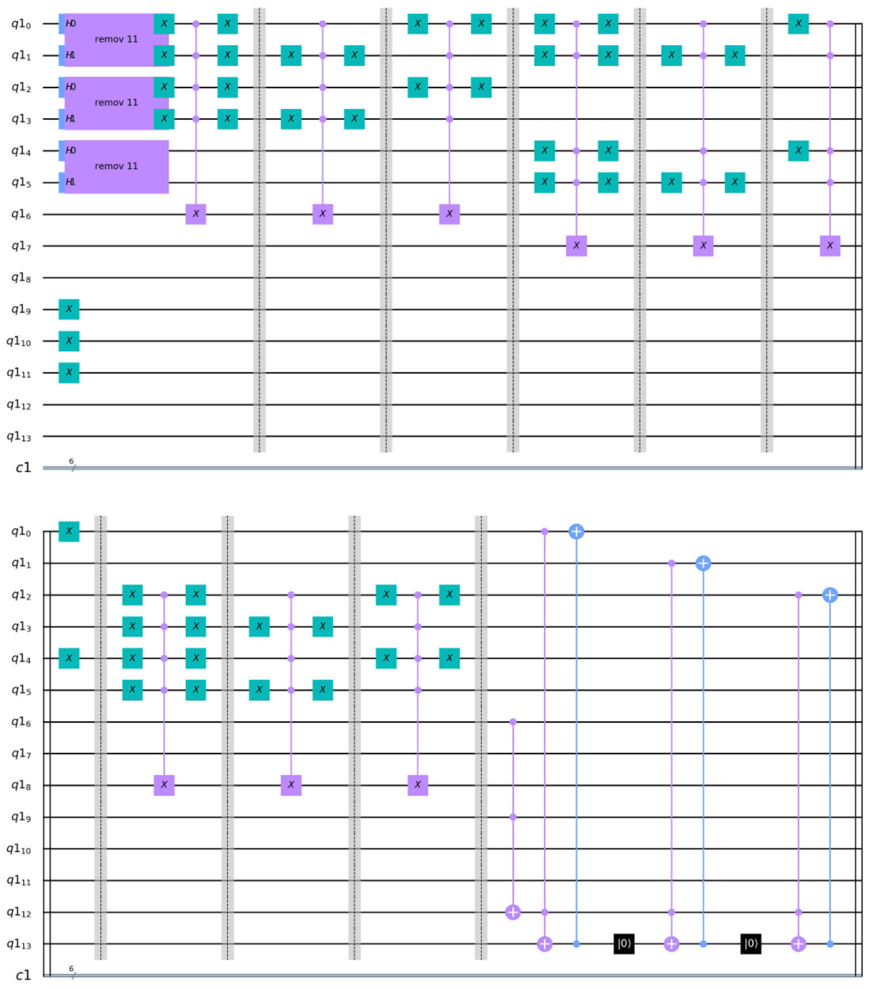


FIG. 17: Qiskit circuit, Q method, triangle graph, three colors (1/3).

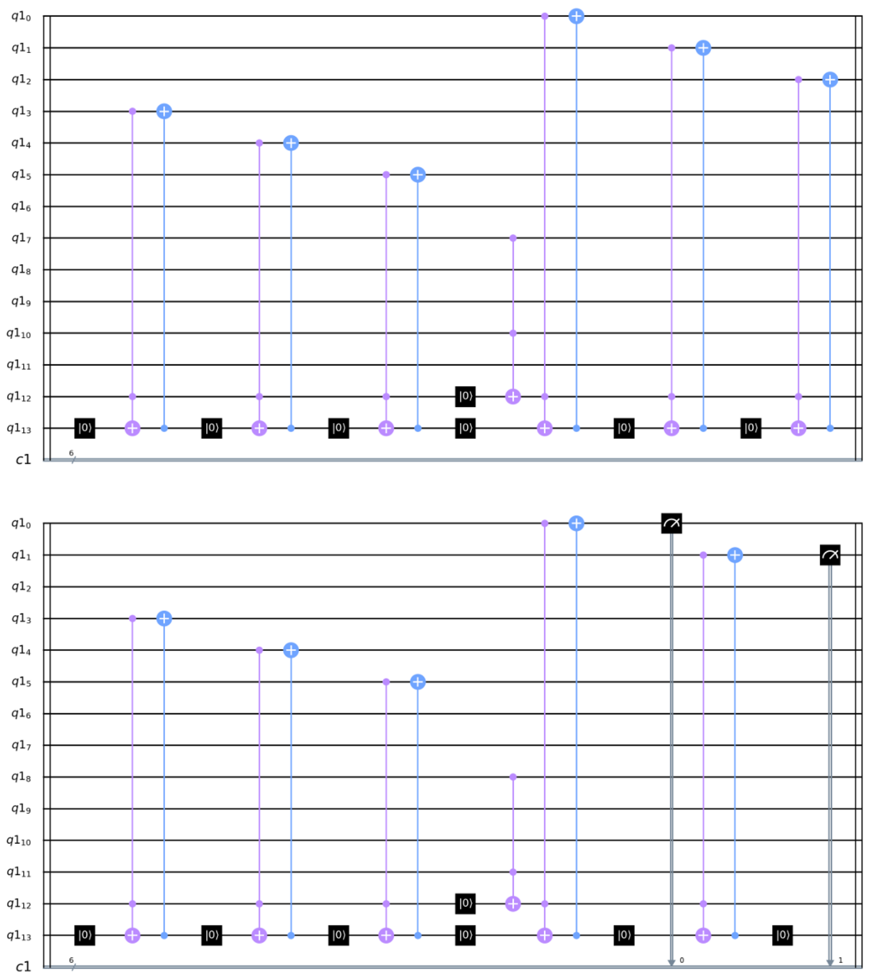


FIG. 18: Qiskit circuit, Q method, triangle graph, three colors (2/3).

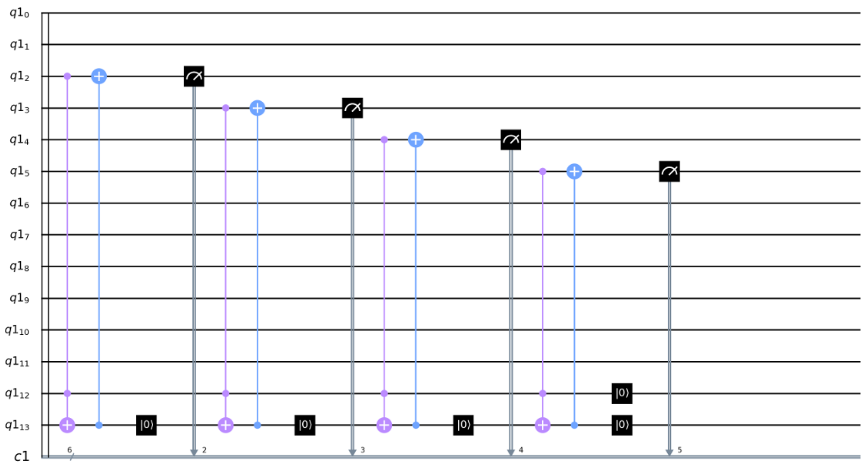


FIG. 19: Qiskit circuit, Q method, triangle graph, three colors (3/3).

References

[1] Stephen S. Bullock and Igor L. Markov. Arbitrary two-qubit computation in 23 elementary gates. *Physical Review A*, 68(1):012318, July 2003. Publisher: American Physical Society.

[2] E. O. Kiktenko, A. S. Nikolaeva, Peng Xu, G. V. Shlyapnikov, and A. K. Fedorov. Scalable quantum computing with qudits on a graph. *Physical Review A*, 101(2):022304, February 2020. arXiv: 1909.08973.

[3] Muskan Saha, Bikash Behera, and Prasanta Panigrahi. *Quantum Algorithms for Colouring of Graphs*. January 2019.