PYL749 - Quantum Information & Computing
Colouring graphs with Quantum Computing

Yash Solanki, 2021PH10813

November 2023

1 Introduction: K-Colourable graphs

1.1 Graphs:

A graph is an ordered pair G = (V, E) such that:

- V is a set of vertices(called Nodes), and
- E $\subseteq$ {{x, y}, x $\in$ V and y $\in$ V and x $\neq$ y}, is the set of edges, an unordered pair {x, y}

The set E shows the association between the vertex pair {x, y}.

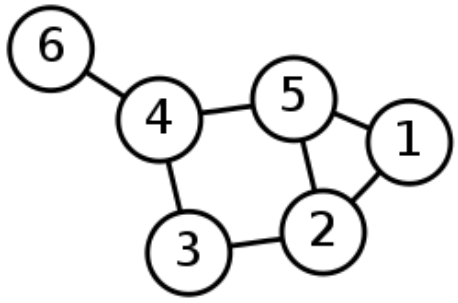

Figure 1: A graph

1.2 Colouring a graph:

In graph theory, colouring a graph assigns labels $\forall$ v ϵ V, such that none of its neighbours has the same label(read: colour). Colouring, as a computational problem, has a large number of practical applications, such as studying networks in electrical engineering, matching in computer science, exam scheduling for institutions and solving Sudoku puzzles for the early birds. K-colouring and

bipartite matching are some of the essential concepts used for these applications.

K-colouring is a colouring of graph G such that different labels/colours are used at exactly' K'. The chromatic number χ(G) denotes the minimum number of colours required to colour a graph. For our application, we will consider the case of K $\geq \chi$(G) or report that it doesn't exist.

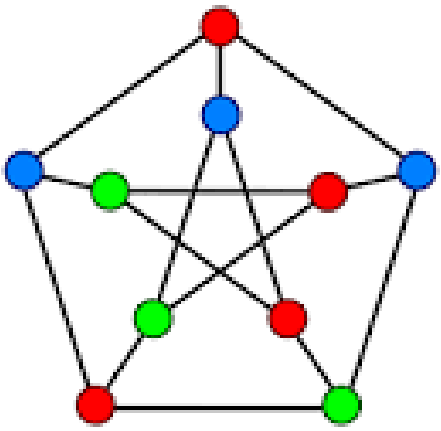

Figure 2: A 4-coloured graph

The above graph is 4-coloured. Interestingly, any planar graph can be 4-coloured(which is really cool)! In general, K-colouring is an NP-complete problem. Hence, we have to 'search' through all the possible solutions to check where a graph is K-colourable and find one such colouring.

2 Grover's Algorithm- The Quantum search algorithm:

2.1 Grover's algorithm for one solution:

Grover's algorithm is a quantum search algorithm on unstructured(without any known order of constraint) that finds the output with a high probability that a boolean function is satisfied. The time complexity of Grover's algorithm is $O(N^{1/2})$, where N is the number of elements present in the domain of possible solutions. Since NP-complete problems require an exhaustive search subroutine, we can often speed up the search with Grover's algorithm.

We have a boolean f: {0, 1, 2 ... N - 1} $\mapsto$ {0, 1} such that f is true for only one value of x in the domain. We then have a subroutine/oracle U_ω such that:

- if f(x) = 1, then $U_\omega|x\rangle$ = -$|x\rangle$

- if f(x) = 0, then $U_\omega|x\rangle = |x\rangle$

The algorithm gives x such that f(x) = 1 with a probability of $\geq 1/2$ for each iteration. We can get arbitrarily large accuracy by running the routine several times. We then invert the amplitudes about the mean frequency, increasing the required x's amplitude and suppressing other values. Measuring the amplitude of states then gives us the required answer.$(\pi N^{1/2})/4$ is the approximate number of steps which give us the exact result

More formally, the algorithm follows the following steps:

- Initialize the system to the uniform superposition over all states
- Perform the subroutine r(N) times. First apply U_ω followed by $U_s = 2|s\rangle\langle s| - I$, called Grover's diffusion operator
- Measure the resulting quantum state in the computational basis.

The circuit for this algorithm is as follows:

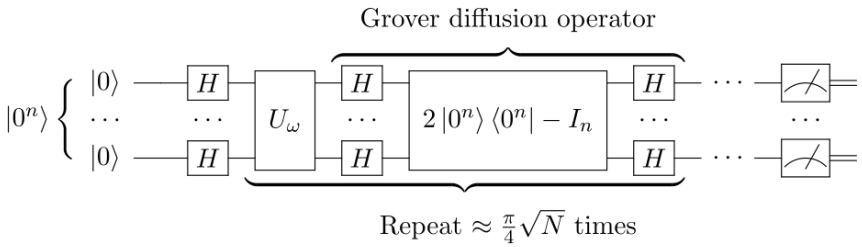

Figure 3: Quantum Circuit for Grover's algorithm, adapted from [4]

The main idea behind Grover's algorithm is amplitude amplification of the state for which we are searching. We start with an initial state that is superposition of the domain in which we have to find the solution. This if followed by inverting the amplitude of state which satisfies our constraints.

These steps are followed by Grover's diffusion, which is equivalent inverting the values about the mean. Since, only the value for which f(x) = 1, the state has a negative phase whereas all other states have a positive phase, inverting about the mean amplifies the negative phase state. Finally, on measuring the state in computational basis, the basis vector with f(x) has the highest probability of being measured.

To make sure that we get the right state upon measurement, we repeat the above subroutine/oracle r(N) times so that the probability of measuring the right state tends to 1.

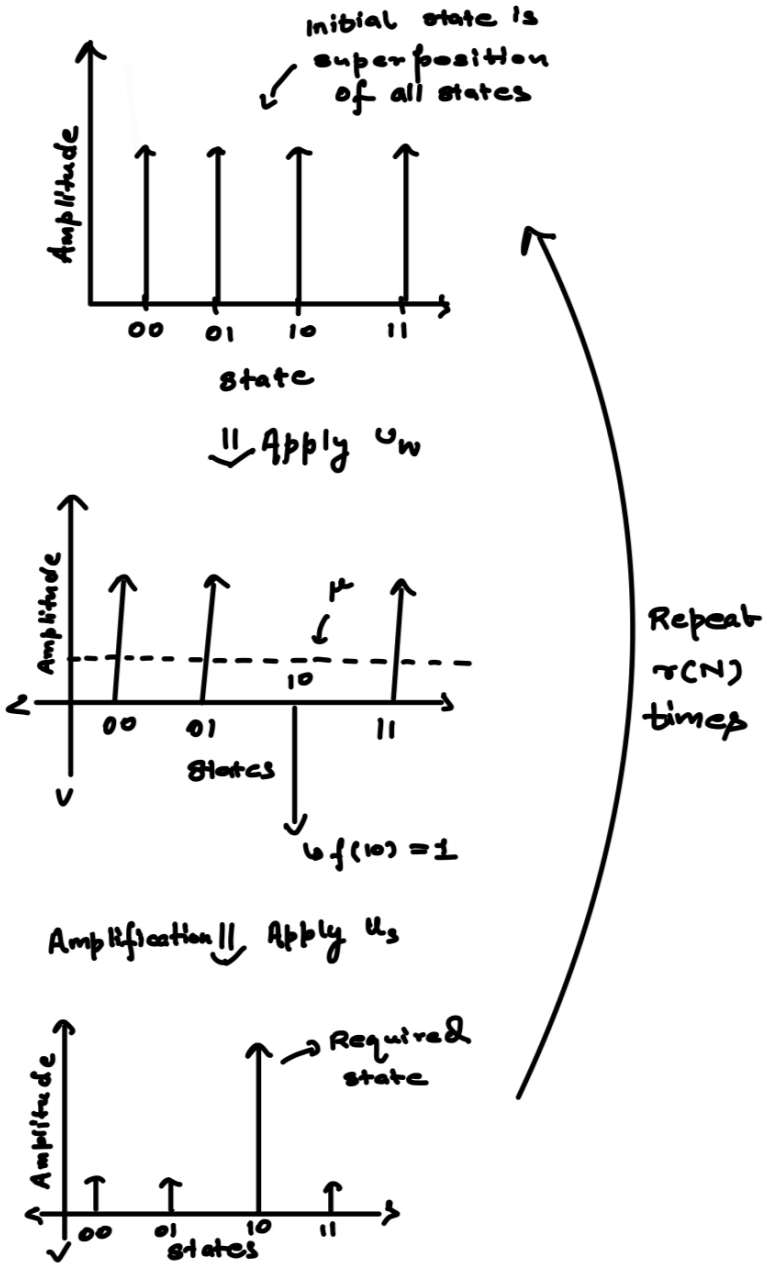

Figure 4: Schematic of Grover's algorithm

2.2 Modification for many solutions:

It is known that for oracles with at most one solution, the optimal number of iterations to apply is $\frac{\pi}{4}\sqrt{N}$, where N = 2^n is the size of the search space. However, it is also possible to handle oracles with multiple solutions. For oracles with k solutions, the optimal number of iterations is instead $\frac{\pi}{4}\sqrt{\frac{N}{k}}$. To handle oracles with a possibly unknown number of solutions, the algorithm can be repeated using powers of two for k, which converges to an amortized complexity $O(\sqrt{N})$.

Changing the number of iterations suffices to handle oracles with unknown numbers of solutions. Hence, in our implementation, we restrict ourselves to the case of oracles with at most a single solution, since the circuit reduction is independent of number of satisfying assignments in the Boolean formula. This modification allows us to find all possible solutions for the colouring.

3 Coding in Qiskit:

3.1 The code:

For the sake of simplicity and limited availability of hardward on actual quantum computer, I've considered 2-colouring of graphs, a.k.a. bipartite graphs.

Firstly, we import the required libraries of Qiskit, numpt and matplotlib:

```
import numpy as np
from qiskit import QuantumCircuit, execute, transpile, Aer, IBMQ
from qiskit.visualization import *
import matplotlib.pyplot as plt
from qiskit.visualization import plot_histogram
from qiskit import QuantumRegister, ClassicalRegister,
    QuantumCircuit, IBMQ, execute
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit.providers.ibmq import least_busy
from qiskit.visualization import plot_histogram
```

We then define a XOR gate function that takes a quantum circuit(qc), two inputs(a and b) and a output as arguments. Also, we define our graph in the edge-list format, where the pair [x, y] implies that there exists an edge between the vertices x and y. The variable num_vertex denotes the total number of nodes in our graph.

```
edge_list = [[0,1], [0,2], [1,3], [2,3]]    # This is the edge list
num_vertex = 4                               # This is the number of
     nodes

def XOR(qc, a, b, output):
# This function takes the quantum circuit 'qc' and make controlled-
    not gate with a and b as input
    qc.cx(a, output)
    qc.cx(b, output)
```

We define the variables for our circuit. Var_qubits, edge_qubits, output_qubits, cbits and qc are the the qubits for the variable, edges, final output state, classical bit-length and the Quantum circuit, respectively.

```
var_qubits = QuantumRegister(num_vertex, name='v')
edge_qubits = QuantumRegister(len(edge_list), name='c')
output_qubit = QuantumRegister(1, name='out')
cbits = ClassicalRegister(num_vertex, name='cbits')
qc = QuantumCircuit(var_qubits, edge_qubits, output_qubit, cbits)
```

We now define the edge colour oracle and apply the controlled-not gate between each ends of an edge using the XOR function defined above. We flip the output if all edges are satisfied which corresponds to the inversion in U_ω define about, i.e., $(-1)^{f(x)} |x\rangle$

```
def edgeColor_oracle(qc, edge_list, edge_qubits):
    i = 0
    for edge in edge_list:
        XOR(qc, edge[0], edge[1], edge_qubits[i])
        i += 1
    # Flip 'output' bit if all edges are satisfied, mcx is the muli
    -control x operator
    qc.mcx(edge_qubits, output_qubit)
    # Reset the edges
    i = 0
    for edge in edge_list:
        XOR(qc, edge[0], edge[1], edge_qubits[i])
        i += 1

edgeColor_oracle(qc, edge_list, edge_qubits)
```

Now, we define the operation of the Grover's diffuser circuit, i.e. U_s from above. We define the diffuser as a gate in revresible fashion. First apply the Hadamard gate on each qubit, which is followed by controlled-not operation on all the qubits. We then use the mcx gate similar to the one from oracle.

```
def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # |s> -> |00..0>, prepare states
    for qubit in range(nqubits):
        qc.h(qubit)
    # Apply cnot |00..0> -> |11..1>
    for qubit in range(nqubits):
        qc.x(qubit)
    qc.h(nqubits-1)
    qc.mcx(list(range(nqubits-1)), nqubits-1)  # multi-controlled
    cnot
    qc.h(nqubits-1)
    # |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)
    # Go back to |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)
    # We create the greate U_s in this process
    U_s = qc.to_gate()
```

```
    U_s.name = "U$_s$"
    return U_s
```

Now, we finally prepare the qc output. Firstly, we initialise the output qubits. Then, we apply hadamard gate on var_qubits to prepare the input state. Itrs represents the number of iterations for convergence to our solution. We then apply the oracle subroutine followed by the Grover's diffusion gate until our solution converges. Finally, we make the measurement and get our final result.

```
# We first initialize the output as superposition of 0 states.
qc.initialize([1, -1]/np.sqrt(2), output_qubit)

# Initialize qubits in state |s>
qc.h(var_qubits)
qc.barrier()                    # For grouping the result

itrs = ((math.pi)*math.isqrt(num_vertex))/4 # Calculate the total
    number of iterations
itrs = int(math.ceil(itrs))                 # Take its int value

for itr in range(0, itrs):
    edgeColor_oracle(qc, edge_list, edge_qubits)
    qc.barrier()
    # Apply our diffuser
    qc.append(diffuser(num_vertex), list(range(num_vertex)))

# Measure the variable qubits
qc.measure(var_qubits, cbits)
```

Finally, we process the result with the following code.

```
backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots = 1024) # Run qc circuit
result = job.result()    # Result object holds the final state
print(result)
counts = result.get_counts()  # Get the result in an array
plot_histogram(counts)        # Final plot of the result
```

Here, 1024 shots have been used for sampling, i.e. we use 1024 instances for sampling our final output.

In the above code, I used qasm_simulator locally to run the simulation locally. You can get the 'API code' from your account and use the commented part from the code on my GitHub to run it on the actual IBM quantum computer. Since, sharing of API keys is not a safe practice, I've run the code on IBM's Manhattan quantum computer backend to get the results in the section 3.2.

3.2 Results:

The following results were obtained. The graphs have been visualized using [7].

1. Example of 2-colouring of graph with 4 nodes:

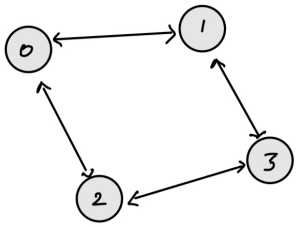

Figure 5: Input graph with 4 nodes

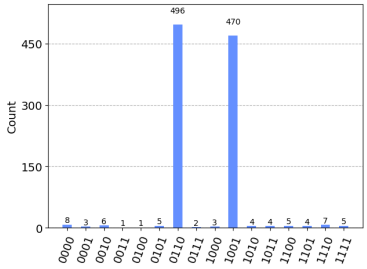

Figure 6: Output after running Grover search

In the above output, 1 corresponds to one colour choice and 0 corresponds to another. Each bit is mapped to corresponding vertex number, respectively. Hence, the following two are the possibilities:

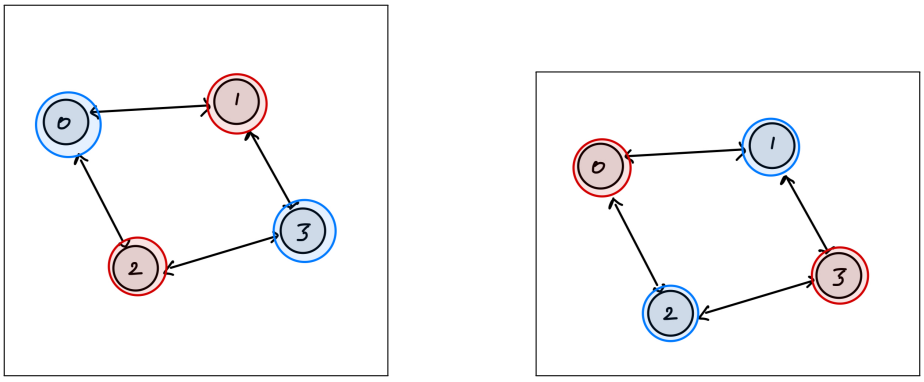

Figure 7: Possible 2-colourings of input graph

2. Example of 2-colouring of graph with 6 nodes:

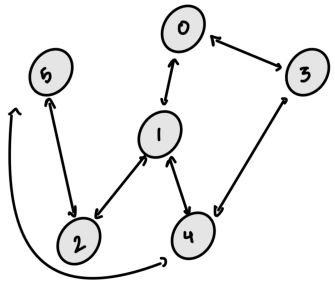

Figure 8: Input graph with 4 nodes

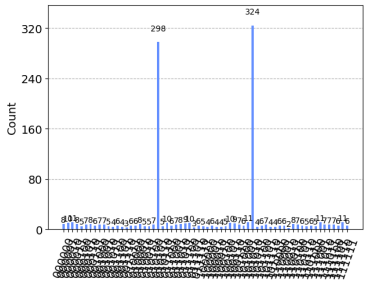

Figure 9: Output after running Grover search

In the above output, the peaks correspond to the bit-strings 101010 and 010101, respectively. The corresponding colourings are:

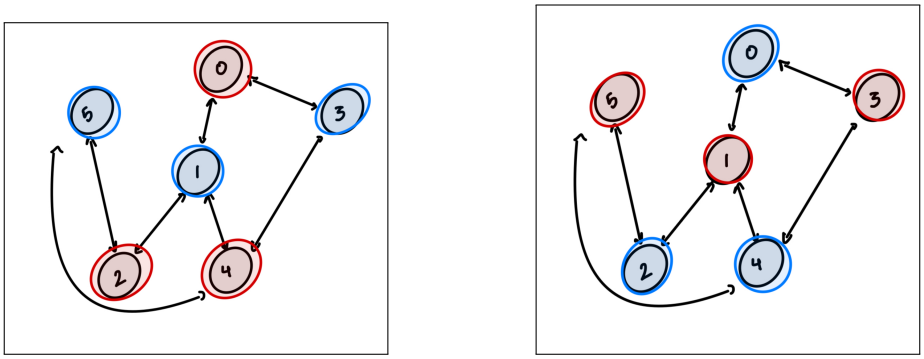

Figure 10: Possible 2-colourings of input graph

4 Results, discussion and future improvements:

It can be seen in the above examples, the quantum computer gives the correct bipartite coloring for the graph, the one where no two vertex connected by edges have the same colour! Hence, the Grover's algorithm was successful in finding the correct colouring of our graph in just $O(N^{1/2})$ iterations, which is a massive speedup to classical methods of exhaustively searching through the solution space.

However, we encounter two major hurdle in current near-term quantum computers- the lack of quantum bits to support more complex cases, and large computational overhead. Hence, such computations as of now are not very helpful. Currently, I used n*k Qubits for the computations which grows very fast and it not practical. The future generation of fault-tolerant computers with better hardware might be able to speedup the process for practical instances of problems and veritication of solutions for NP-complete problems of practical interest.

5 References and acknowledgements

1. Wikipedia: Graph Theory
2. Wikipedia: Graph Colouring
3. K-SAT to 3-SAT
4. Wikipedia: Grover's algorithm
5. K-colourability to 3-SAT
6. Descartes, Blanche, "A three colour problem", Eureka
7. CSA's graph visualization tool

6 Github link:

The code can be found at This GitHub link. You can uncomment the back-end part and add your API Key from IBM Qiskit's website to test the code on a Quantum computer.