Solving the Graph 4-Coloring Problem using a Quantum Circuit Generator

Tuesday, 4/30/2018

Graph Coloring

Assign a color to each vertex such that adjacent vertices (those sharing an edge) have different colors.

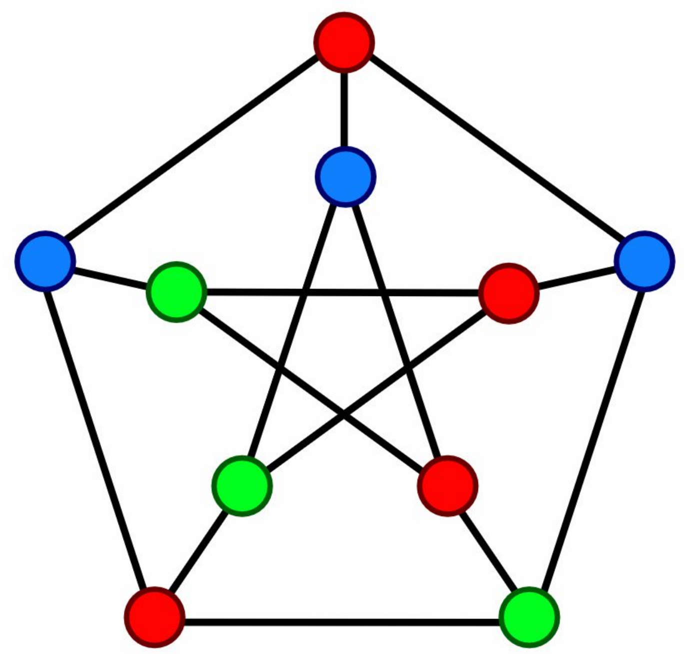


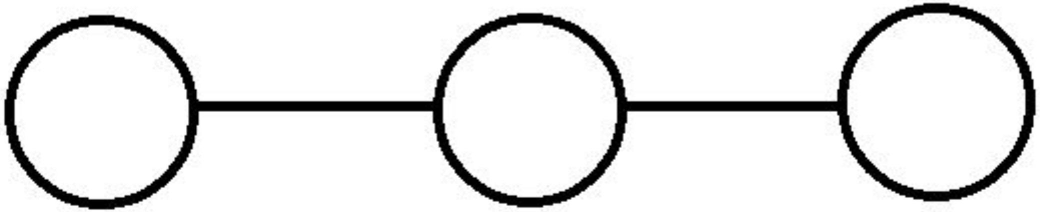
Image Source: https://upload.wikimedia.org/wikipedia/commons/9/90/Petersen_graph_3-coloring.svg

Quantum Search: Grover’s Algorithm

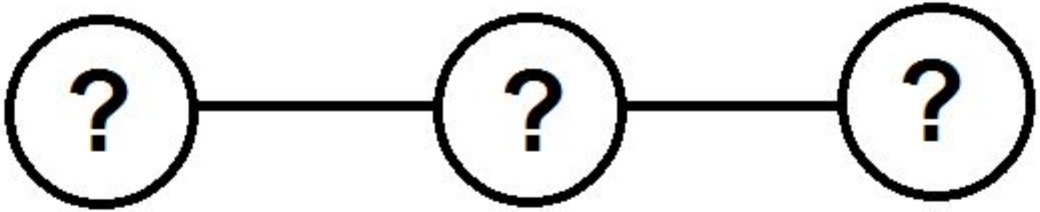
Starting with entangled qubits:

1. Create a superposition to make each state equally likely
2. Repeat K times:
    a. Apply the quantum oracle operation to negate the amplitude of the desired result
    b. Invert about the mean, to increase the amplitude of the negated result

2-Coloring

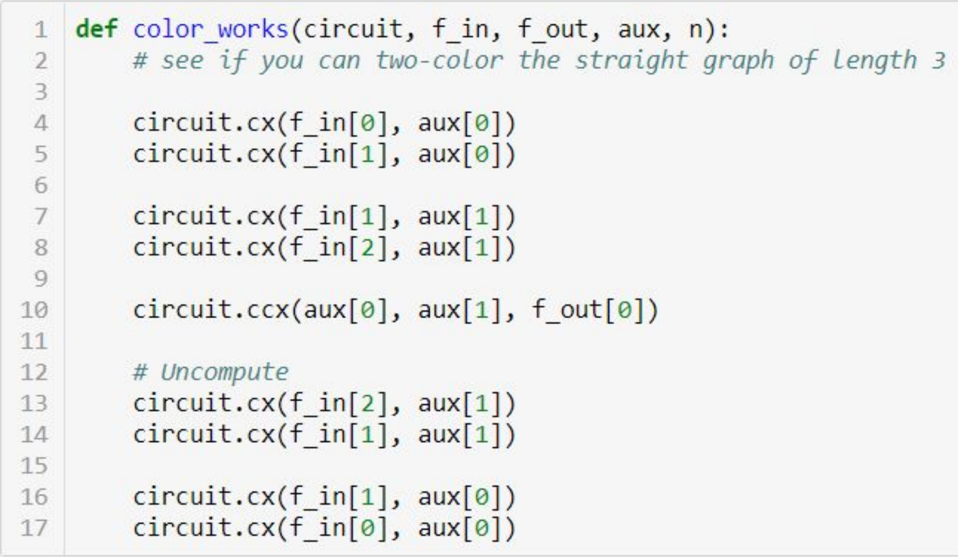


2-Coloring



2-Coloring

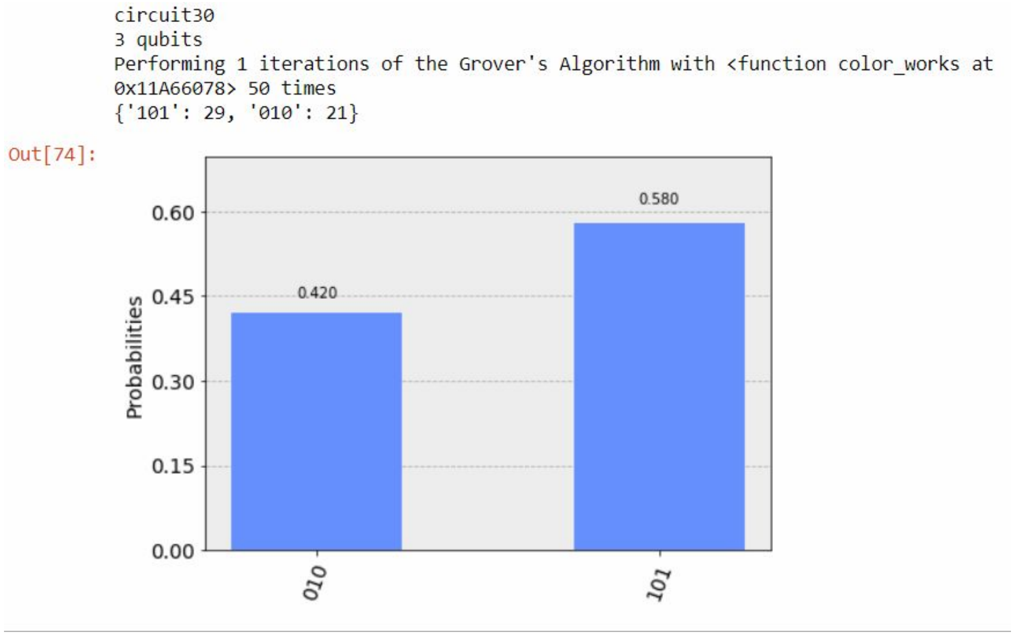
func: color2_works



```python
def color_works(circuit, f_in, f_out, aux, n):
    # see if you can two-color the straight graph of length 3

    circuit.cx(f_in[0], aux[0])
    circuit.cx(f_in[1], aux[0])

    circuit.cx(f_in[1], aux[1])
    circuit.cx(f_in[2], aux[1])

    circuit.ccx(aux[0], aux[1], f_out[0])

    # Uncompute
    circuit.cx(f_in[2], aux[1])
    circuit.cx(f_in[1], aux[1])

    circuit.cx(f_in[1], aux[0])
    circuit.cx(f_in[0], aux[0])
```

2-Coloring

`func: color2_works`



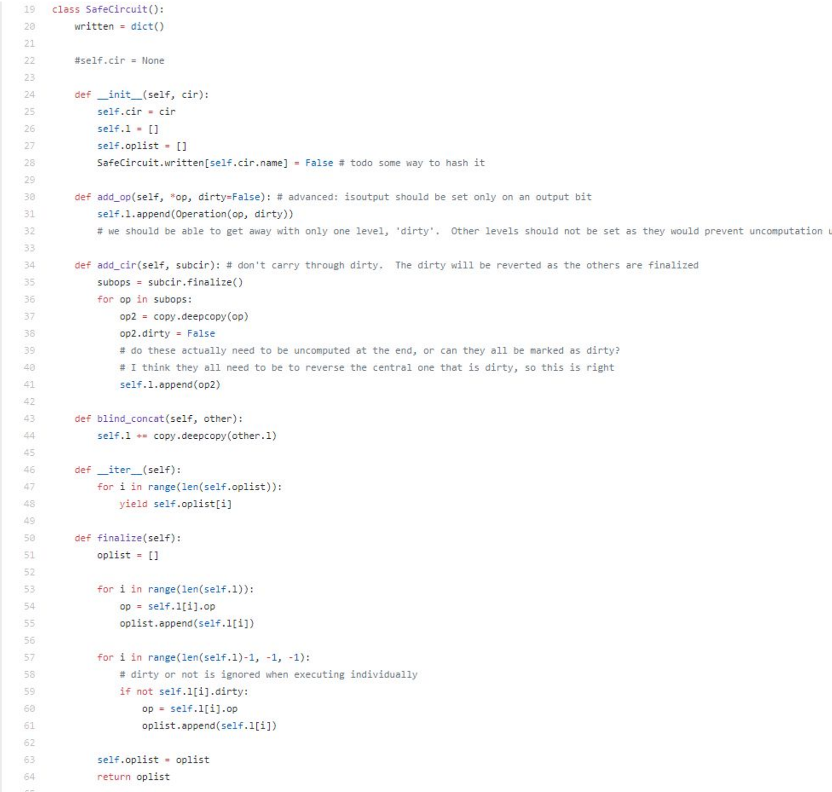
```
circuit30
3 qubits
Performing 1 iterations of the Grover's Algorithm with <function color_works at
0x11A66078> 50 times
{'101': 29, '010': 21}
```

Out[74]:

The Need for SafeCircuit

Takes care of the uncomputation required.

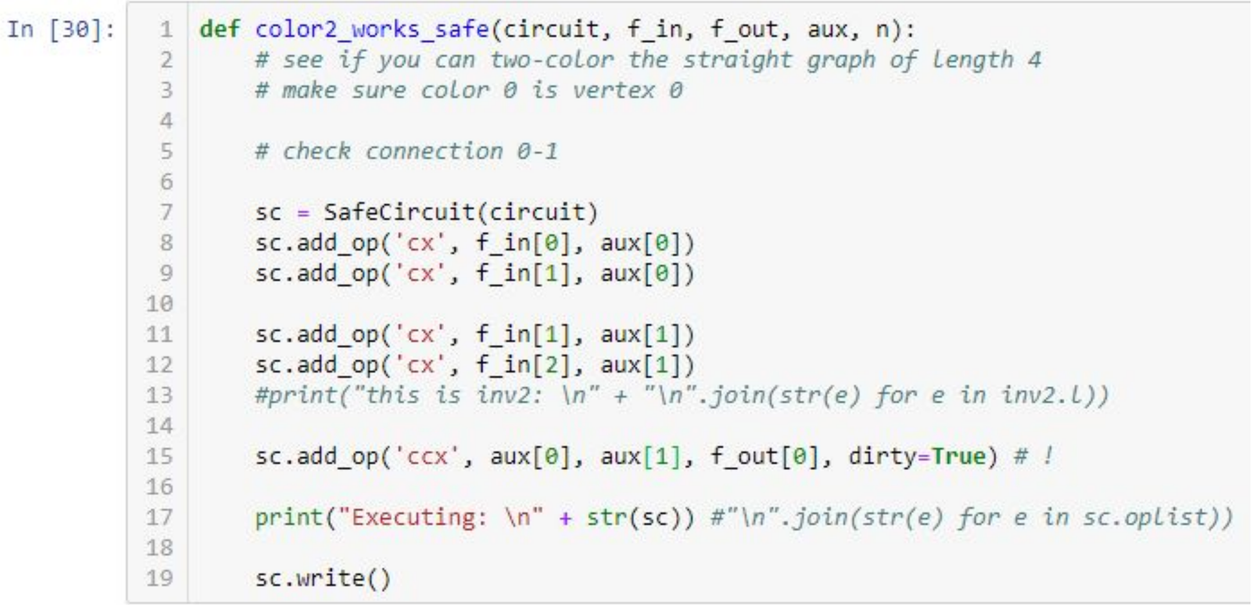
Allows you to combine multiple such circuits.



```python
class SafeCircuit():
    written = dict()

    #self.cir = None

    def __init__(self, cir):
        self.cir = cir
        self.l = []
        self.oplist = []
        SafeCircuit.written[self.cir.name] = False # todo some way to hash it

    def add_op(self, *op, dirty=False): # advanced: isoutput should be set only on an output bit
        self.l.append(Operation(op, dirty))
        # we should be able to get away with only one level, 'dirty'.  Other levels should not be set as they would prevent uncomputation u

    def add_cir(self, subcir): # don't carry through dirty.  The dirty will be reverted as the others are finalized
        subops = subcir.finalize()
        for op in subops:
            op2 = copy.deepcopy(op)
            op2.dirty = False
            # do these actually need to be uncomputed at the end, or can they all be marked as dirty?
            # I think they all need to be to reverse the central one that is dirty, so this is right
            self.l.append(op2)

    def blind_concat(self, other):
        self.l += copy.deepcopy(other.l)

    def __iter__(self):
        for i in range(len(self.oplist)):
            yield self.oplist[i]

    def finalize(self):
        oplist = []

        for i in range(len(self.l)):
            op = self.l[i].op
            oplist.append(self.l[i])

        for i in range(len(self.l)-1, -1, -1):
            # dirty or not is ignored when executing individually
            if not self.l[i].dirty:
                op = self.l[i].op
                oplist.append(self.l[i])

        self.oplist = oplist
        return oplist
```

2-Coloring

func: color2_works_safe

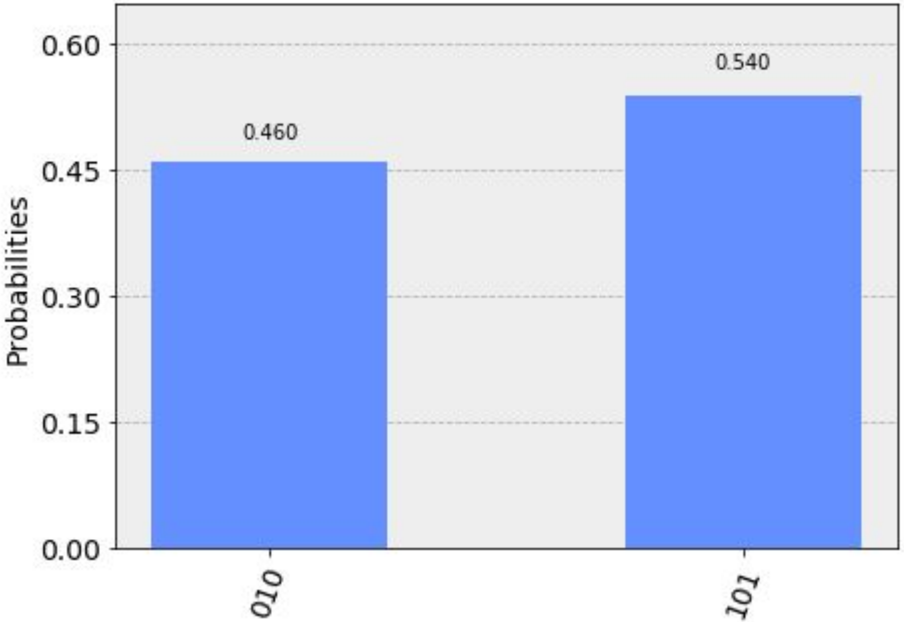


```python
def color2_works_safe(circuit, f_in, f_out, aux, n):
    # see if you can two-color the straight graph of length 4
    # make sure color 0 is vertex 0

    # check connection 0-1

    sc = SafeCircuit(circuit)
    sc.add_op('cx', f_in[0], aux[0])
    sc.add_op('cx', f_in[1], aux[0])

    sc.add_op('cx', f_in[1], aux[1])
    sc.add_op('cx', f_in[2], aux[1])
    #print("this is inv2: \n" + "\n".join(str(e) for e in inv2.l))

    sc.add_op('ccx', aux[0], aux[1], f_out[0], dirty=True) # !

    print("Executing: \n" + str(sc)) #"\n".join(str(e) for e in sc.oplist))

    sc.write()
```

2-Coloring

```
func: color2_works_safe
```

```
Out[36]:
```



2-Coloring

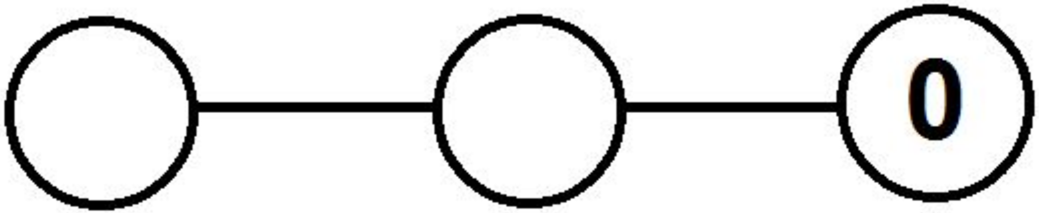
We get either 010 or 101. For a larger amount of colors, when noise is introduced this might diminish our results as we end up with exponentially more colorings for the same size of graph.

We can introduce additional conditions. In the next one, we force q[2] to be 0.

2-Coloring

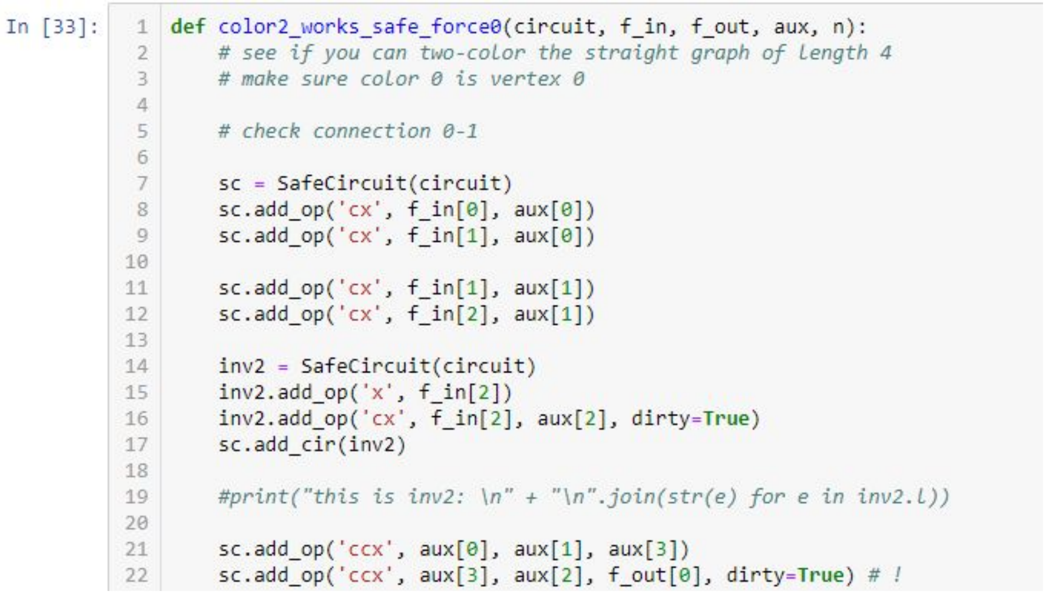
We get either 010 or 101. For a larger amount of colors, when noise is introduced this might diminish our results as we end up with exponentially more colorings for the same size of graph.

We can introduce additional conditions. In the next one, we force q[2] to be 0.



2-Coloring: Forcing a Unique Coloring

func: color2_works_safe_force0

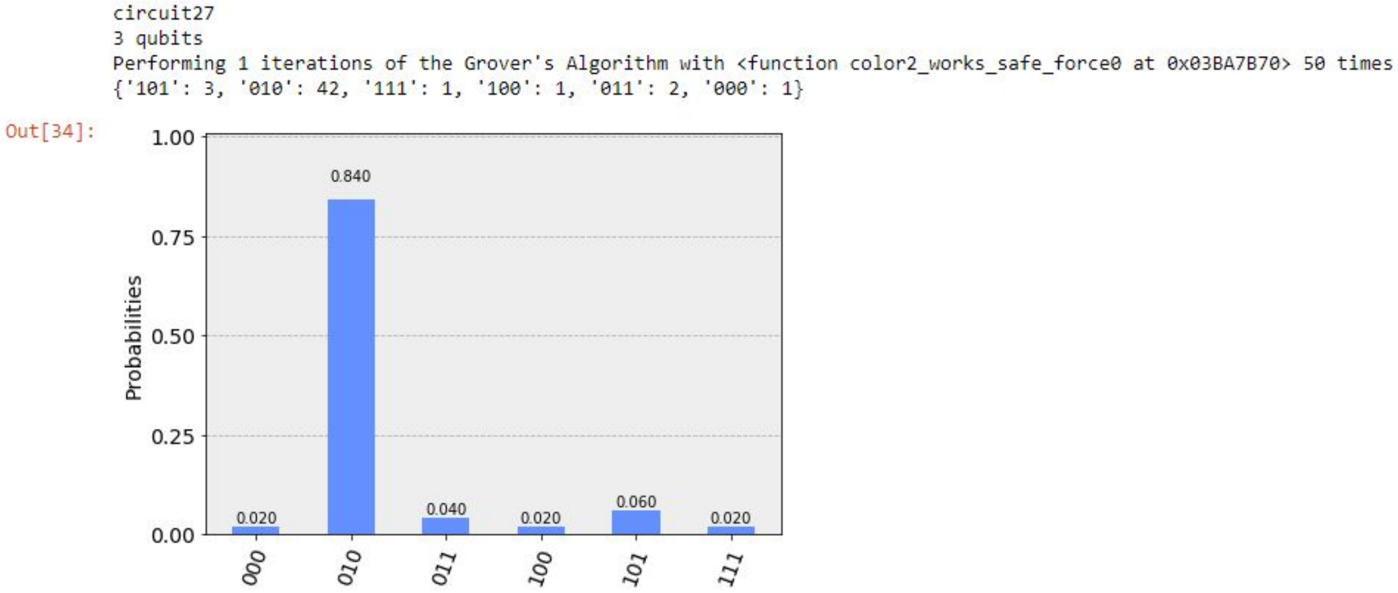


In [33]:

```python
def color2_works_safe_force0(circuit, f_in, f_out, aux, n):
    # see if you can two-color the straight graph of length 4
    # make sure color 0 is vertex 0

    # check connection 0-1

    sc = SafeCircuit(circuit)
    sc.add_op('cx', f_in[0], aux[0])
    sc.add_op('cx', f_in[1], aux[0])

    sc.add_op('cx', f_in[1], aux[1])
    sc.add_op('cx', f_in[2], aux[1])

    inv2 = SafeCircuit(circuit)
    inv2.add_op('x', f_in[2])
    inv2.add_op('cx', f_in[2], aux[2], dirty=True)
    sc.add_cir(inv2)

    #print("this is inv2: \n" + "\n".join(str(e) for e in inv2.l))

    sc.add_op('ccx', aux[0], aux[1], aux[3])
    sc.add_op('ccx', aux[3], aux[2], f_out[0], dirty=True) # !
```

2-Coloring: Forcing a Unique Coloring

`func: color2_works_safe_force0`



```
circuit27
3 qubits
Performing 1 iterations of the Grover's Algorithm with <function color2_works_safe_force0 at 0x03BA7B70> 50 times
{'101': 3, '010': 42, '111': 1, '100': 1, '011': 2, '000': 1}
```

Out[34]:

From 2-Coloring to 4-Coloring

Construct a truth table.

We want to see whether the colors are different in any of the bits. We can check each bit and then compare them.

x[0]	x[2]	t1
0	0	0
0	1	1
1	0	1
1	1	0

x[1]	x[3]	t2
0	0	0
0	1	1
1	0	1
1	1	0

t1	t2	Answer
0	0	0
0	1	1
1	0	1
1	1	1

From 2-Coloring to 4-Coloring

Construct a truth table.

We want to see whether the colors are different in any of the bits. We can check each bit and then compare them.

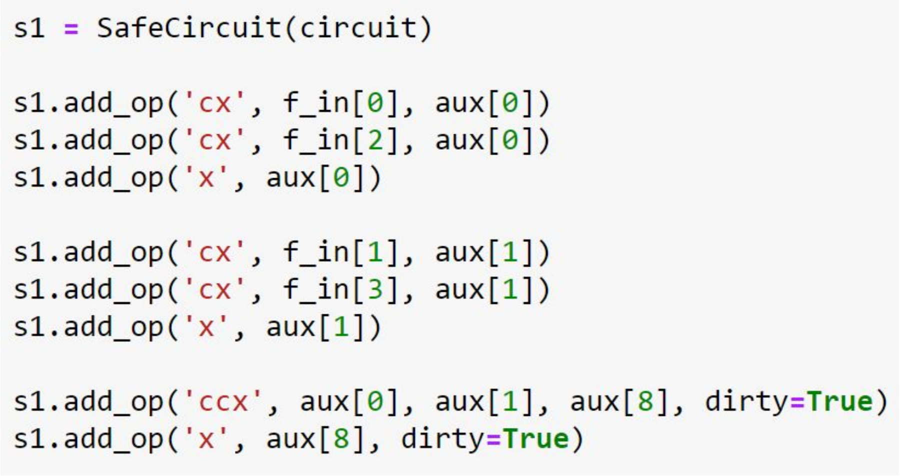
x[0]	x[2]	t1
0	0	0
0	1	1
1	0	1
1	1	0

x[1]	x[3]	t2
0	0	0
0	1	1
1	0	1
1	1	0

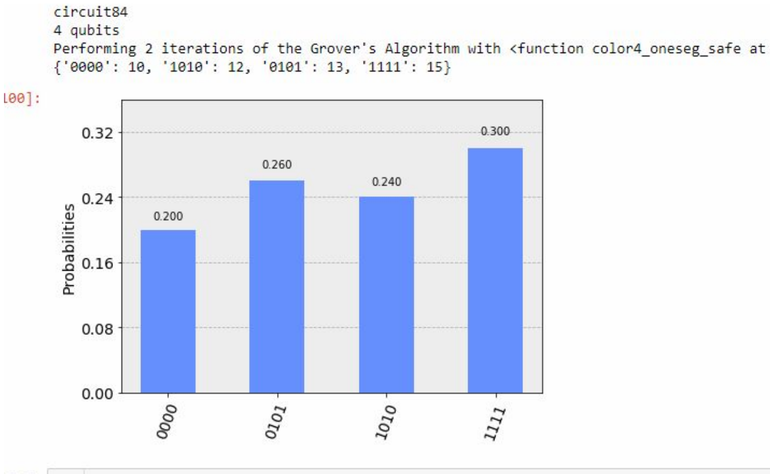
t1	t2	Answer
0	0	0
0	1	1
1	0	1
1	1	1

These first 2 look like an XOR, and the final one is a NOR. Let’s implement these with reversible quantum gates.

From 2-Coloring to 4-Coloring



```
s1 = SafeCircuit(circuit)

s1.add_op('cx', f_in[0], aux[0])
s1.add_op('cx', f_in[2], aux[0])
s1.add_op('x', aux[0])

s1.add_op('cx', f_in[1], aux[1])
s1.add_op('cx', f_in[3], aux[1])
s1.add_op('x', aux[1])

s1.add_op('ccx', aux[0], aux[1], aux[8], dirty=True)
s1.add_op('x', aux[8], dirty=True)
```

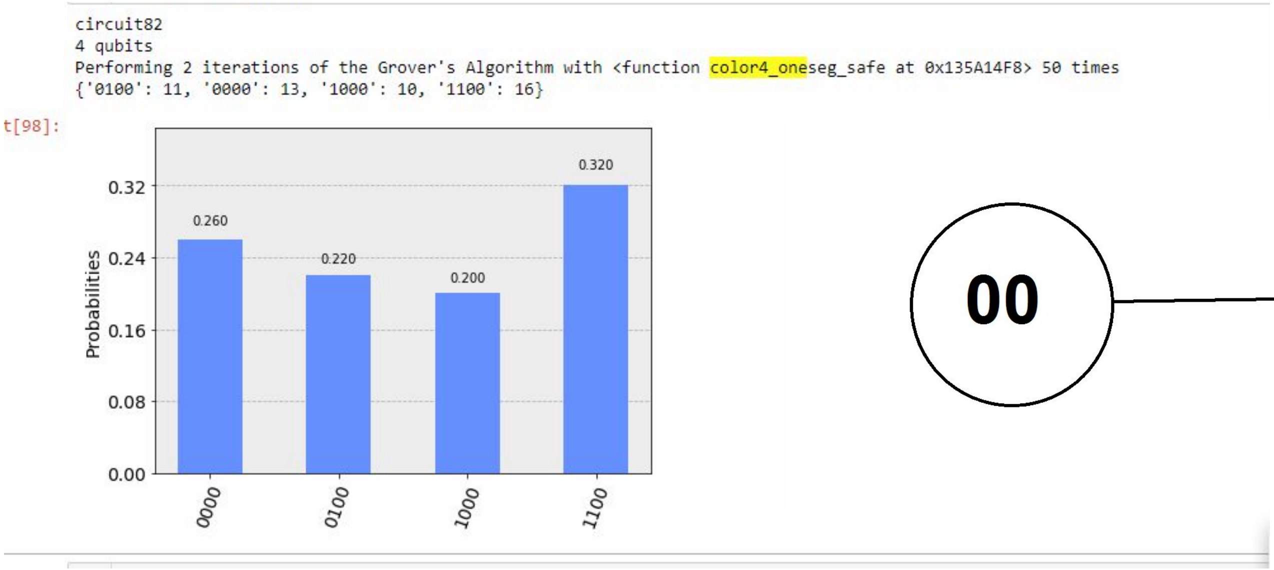


```
circuit84
4 qubits
Performing 2 iterations of the Grover's Algorithm with <function color4_oneseg_safe at
{'0000': 10, '1010': 12, '0101': 13, '1111': 15}
```

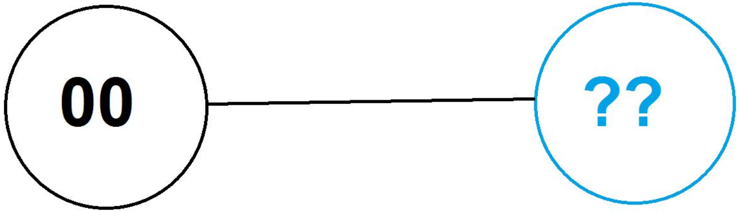
This works. How might we integrate this to test actual graphs, and test the implementation of our algorithm?

Restricting Permutations of Colorings

Let's start our 4-coloring by extending our work with forcing a color to set the first color to 00 (remember the reversed bit ordering):



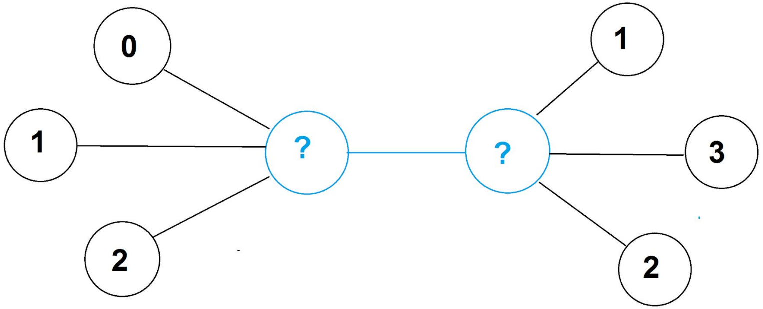
```
circuit82
4 qubits
Performing 2 iterations of the Grover's Algorithm with <function color4_oneseg_safe at 0x135A14F8> 50 times
{'0100': 11, '0000': 13, '1000': 10, '1100': 16}
```



Testing the Implementation

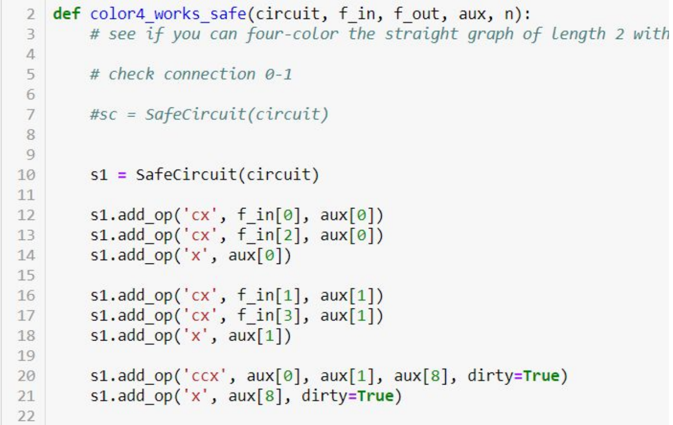
Testing the algorithm on 6 or more input qubits (only 3 nodes) would take about 20 hours per test run on the simulator on my machine for 200 shots.

So, to demonstrate correctness, I constructed connections to dummy nodes on the 2-node (4-qubit) graph shown below.

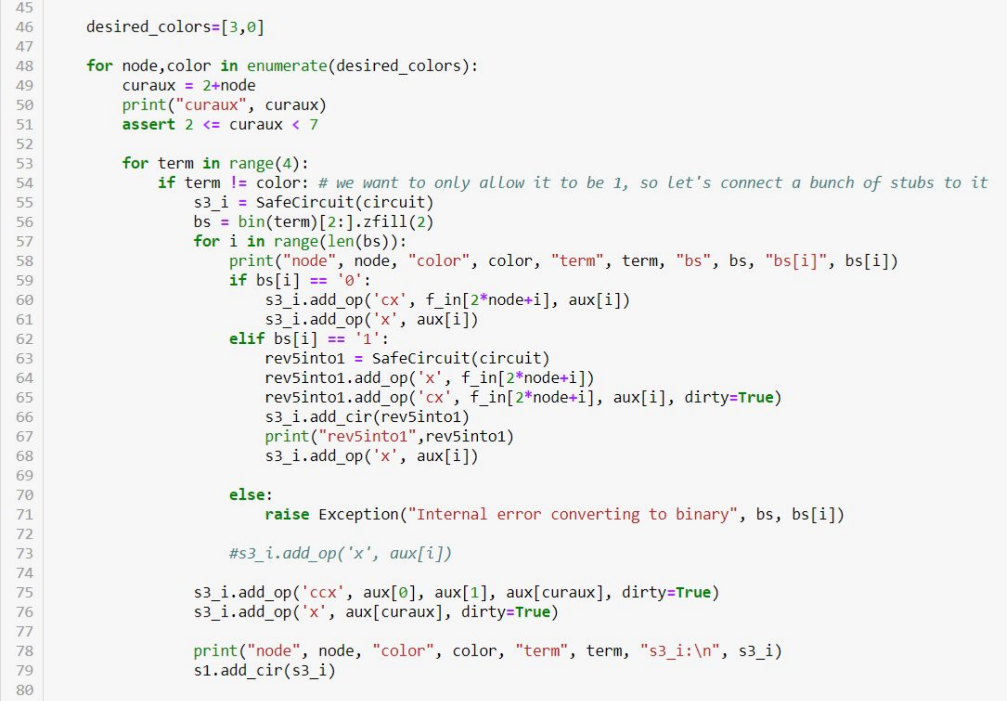


Testing the Implementation

On the left is the code for the graph topology; on the right is the dummy edges.



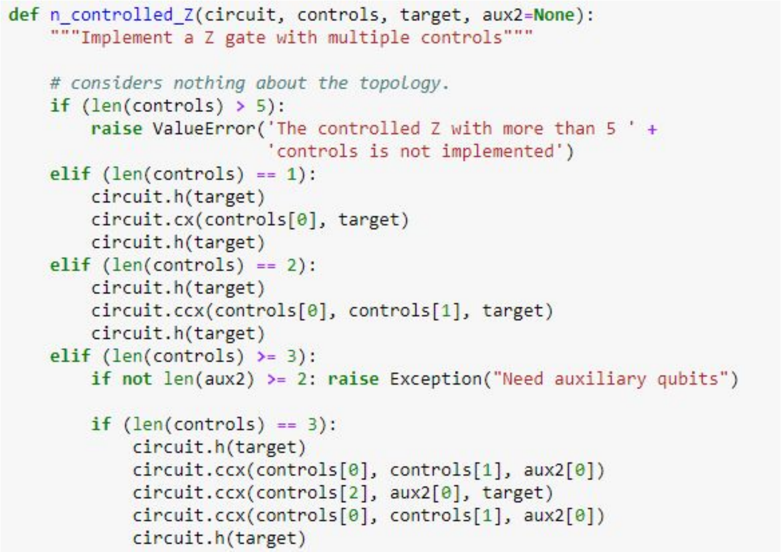
```
def color4_works_safe(circuit, f_in, f_out, aux, n):
    # see if you can four-color the straight graph of length 2 with

    # check connection 0-1

    #sc = SafeCircuit(circuit)


    s1 = SafeCircuit(circuit)

    s1.add_op('cx', f_in[0], aux[0])
    s1.add_op('cx', f_in[2], aux[0])
    s1.add_op('x', aux[0])

    s1.add_op('cx', f_in[1], aux[1])
    s1.add_op('cx', f_in[3], aux[1])
    s1.add_op('x', aux[1])

    s1.add_op('ccx', aux[0], aux[1], aux[8], dirty=True)
    s1.add_op('x', aux[8], dirty=True)

```



```

    desired_colors=[3,0]

    for node,color in enumerate(desired_colors):
        curaux = 2+node
        print("curaux", curaux)
        assert 2 <= curaux < 7

        for term in range(4):
            if term != color: # we want to only allow it to be 1, so let's connect a bunch of stubs to it
                s3_i = SafeCircuit(circuit)
                bs = bin(term)[2:].zfill(2)
                for i in range(len(bs)):
                    print("node", node, "color", color, "term", term, "bs", bs, "bs[i]", bs[i])
                    if bs[i] == '0':
                        s3_i.add_op('cx', f_in[2*node+i], aux[i])
                        s3_i.add_op('x', aux[i])
                    elif bs[i] == '1':
                        rev5into1 = SafeCircuit(circuit)
                        rev5into1.add_op('x', f_in[2*node+i])
                        rev5into1.add_op('cx', f_in[2*node+i], aux[i], dirty=True)
                        s3_i.add_cir(rev5into1)
                        print("rev5into1",rev5into1)
                        s3_i.add_op('x', aux[i])

                    else:
                        raise Exception("Internal error converting to binary", bs, bs[i])

                    #s3_i.add_op('x', aux[i])

                s3_i.add_op('ccx', aux[0], aux[1], aux[curaux], dirty=True)
                s3_i.add_op('x', aux[curaux], dirty=True)

                print("node", node, "color", color, "term", term, "s3_i:\n", s3_i)
                s1.add_cir(s3_i)

```

Building a CNZ (N-controlled Z) Gate

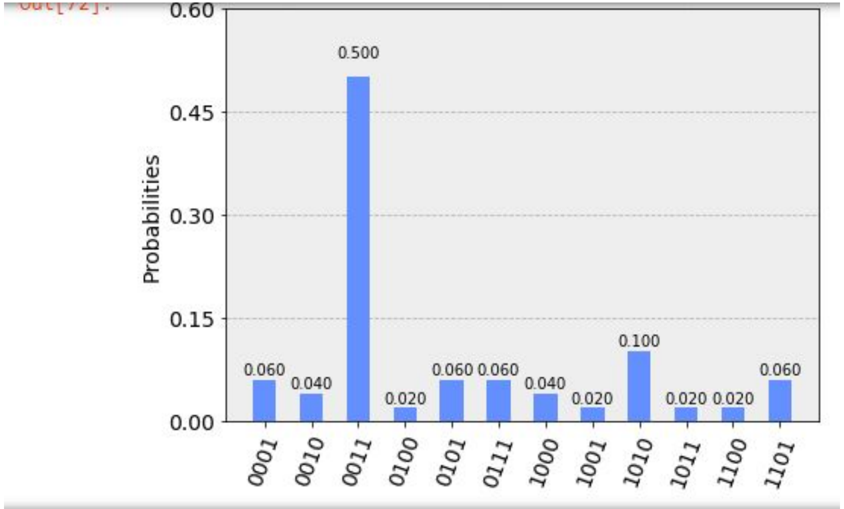
Ancilla (auxiliary) bits need to be used for 3 controls, and reset appropriately before the next use.

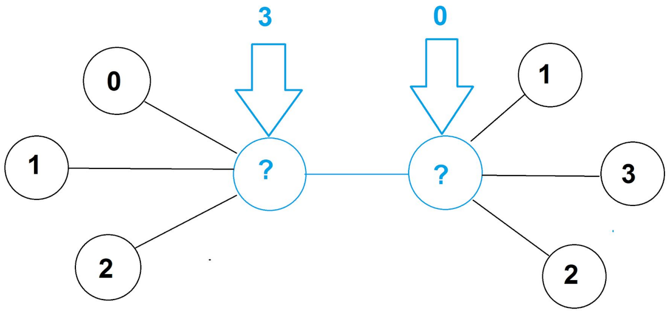


```python
def n_controlled_Z(circuit, controls, target, aux2=None):
    """Implement a Z gate with multiple controls"""

    # considers nothing about the topology.
    if (len(controls) > 5):
        raise ValueError('The controlled Z with more than 5 ' +
                         'controls is not implemented')
    elif (len(controls) == 1):
        circuit.h(target)
        circuit.cx(controls[0], target)
        circuit.h(target)
    elif (len(controls) == 2):
        circuit.h(target)
        circuit.ccx(controls[0], controls[1], target)
        circuit.h(target)
    elif (len(controls) >= 3):
        if not len(aux2) >= 2: raise Exception("Need auxiliary qubits")

        if (len(controls) == 3):
            circuit.h(target)
            circuit.ccx(controls[0], controls[1], aux2[0])
            circuit.ccx(controls[2], aux2[0], target)
            circuit.ccx(controls[0], controls[1], aux2[0])
            circuit.h(target)
```

From 2-Coloring to 4-Coloring

Solution is clear.

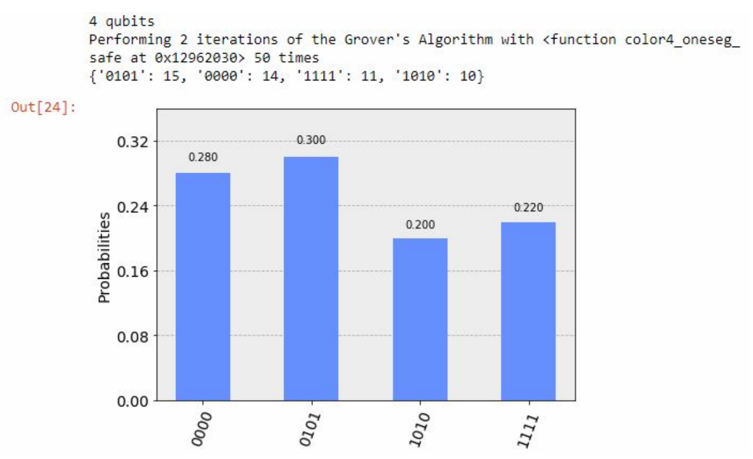




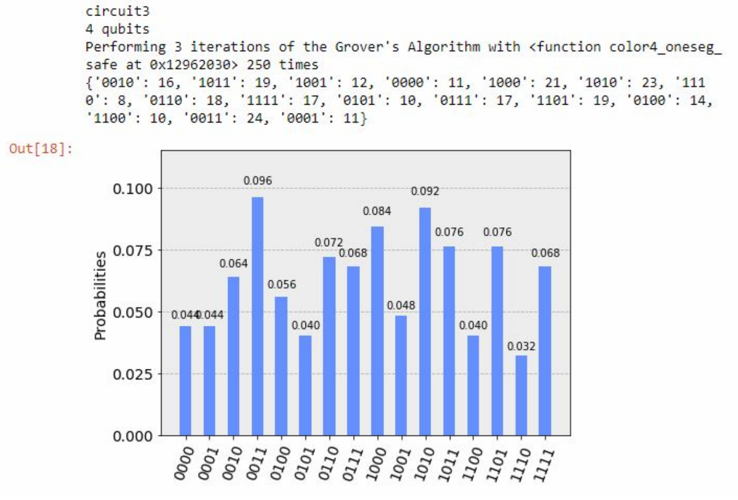
Other Lessons Learned

The optimal number of iterations in Grover’s Algorithm seems to vary.

Sometimes, all of the results returned were valid, but running additional iterations to reach the suggested amount led to more incorrect results returned!



```
4 qubits
Performing 2 iterations of the Grover's Algorithm with <function color4_oneseg_
safe at 0x12962030> 50 times
{'0101': 15, '0000': 14, '1111': 11, '1010': 10}
```



```
circuit3
4 qubits
Performing 3 iterations of the Grover's Algorithm with <function color4_oneseg_
safe at 0x12962030> 250 times
{'0010': 16, '1011': 19, '1001': 12, '0000': 11, '1000': 21, '1010': 23, '111
0': 8, '0110': 18, '1111': 17, '0101': 10, '0111': 17, '1101': 19, '0100': 14,
'1100': 10, '0011': 24, '0001': 11}
```

Performance

- 50% accuracy, yielding a clear answer with 50 shots
- Simulation executes in ~90 seconds on Intel i7-5600U processor @ 2.6 GHz

Summary

- Built a framework that performs reversible computation automatically, ridding programmers from dealing with annoying bugs and “copy and paste”.

- Applied it to solve an instance of the 4-coloring problem, using Grover’s Algorithm.

References

Qiskit. “Qiskit Tutorials”, 2017. Retrieved from https://nbviewer.jupyter.org/github/Qiskit/qiskit-tutorial/blob/master/index.ipynb