Quantum Information and Quantum Estimation 2023

ECE TUC

Athanasios Karakos

Circuit Cutting and Wire Recycling

Table of Contents

Representing Circuits Using Graphs

Method

A quantum circuit causal graph is a **directed acyclic graph** where each circuit operation (initialization, gate, measurement) is abstracted by a node, and edge directions indicate the relative temporal ordering of the circuit operations.

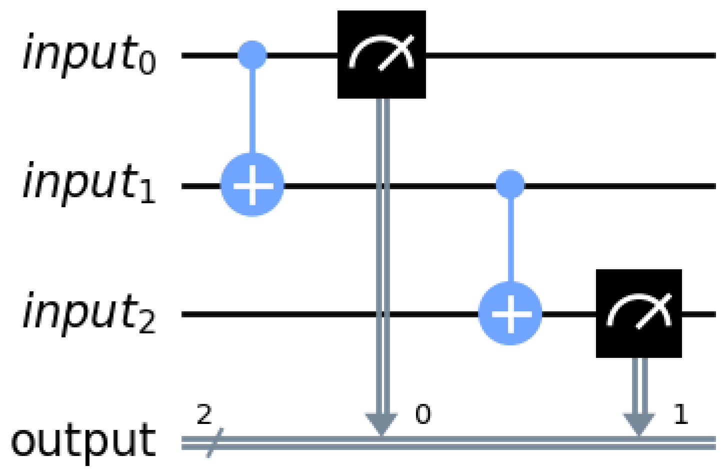


FIG: Quantum Circuit with qubits and gates

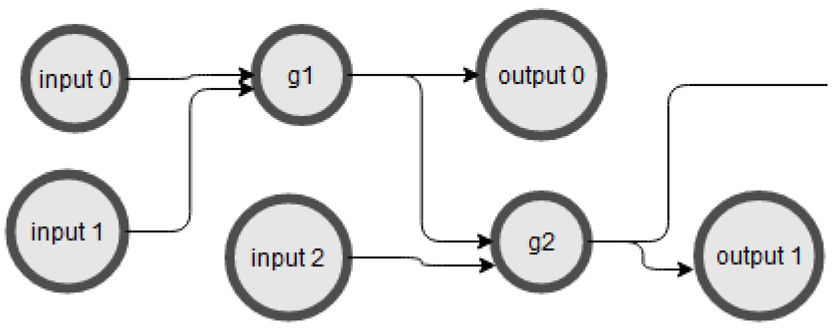


FIG: Casual Graph that corresponds to the above quantum circuit

A graph node represents an operation. A graph edge is abstracting a quantum circuit wire segment existing between the circuit operations represented by the adjacent graph nodes.

A temporal ordering between two circuit operations is established if one of the operations is based on the output of the other. Thus, **if two operations are applied to the same qubit, an edge or a chain of edges will exist** between the corresponding operation nodes.

- The temporal ordering of quantum circuit operations **can be modeled using the → operator** by writing g1 → g2 if gate g1 needs to be executed before gate g2. The situation exists if the gate g2 takes as input one of the outputs of g1.
- **The → operator is transitive**, because if a → b and b → c, then a → c.

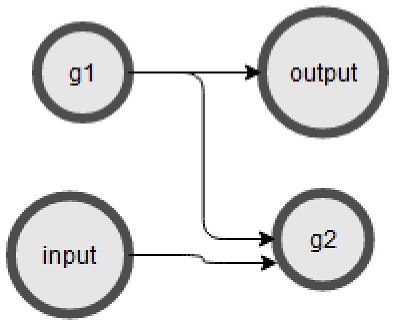
- The qubit initialization process is described with a gate action, for example, we can write input1 → g3, if the gate g3 is applied on the freshly initialized qubit existing at the circuit's input named input1.
- Qubit measurements will not precede any operation, but will always have their own predecessors. For example, g4 → output2 indicates that the output qubit of gate g4 existing on the circuit's output named output2 is measured.

Wire Recycling Algorithm

Transformation Rule:

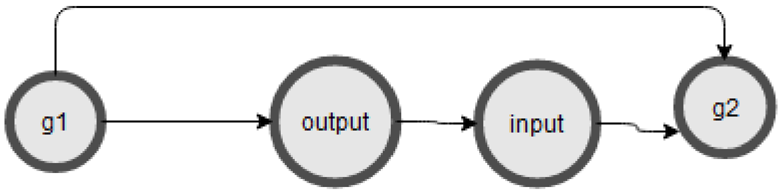
The process begins with a quantum circuit described with an **input graph having each qubit on a distinct wire** and then it is gradually transformed to an **output graph having multiple qubits on the same wire**.

Suppose the input sub-graph:



the input and the output nodes referenced distinct wires

The output sub-graph is:



gate g1 is executed; its output is measured on wire w (node output), a new qubit is initialized on wire w (node input), gate g2 is executed.

The result is that w1 and w2 are treated as segments of w. The passive stages of w1 and w2 were shortened without influencing the computation, the wires do not overlap in time and **are arranged so that w1 → w2.**

Challenges:

In the output subgraph a direct edge was added between an output preceding an input. When adding such an edge two problems arise:

1) The graph contains a cycle which is not allowed because causal graphs are acyclic

2) Although the initialization takes place before the measurement, the edge indicates that the measurement precedes the initialization, which is impossible.

Implementation with Graph Data Structure:

```
In [2]: from qiskit import QuantumRegister, ClassicalRegister
        from qiskit import QuantumCircuit, Aer, transpile, assemble
        from qiskit.visualization import plot_bloch_multivector, plot_histogram, array_to_latex
        from qiskit.tools.visualization import circuit_drawer
        from qiskit.quantum_info import Statevector
        from numpy import pi
        from qiskit.transpiler import PassManager
        from qiskit.transpiler.passes import Unroller
        import random
        import networkx as nx
        import matplotlib.pyplot as plt

        sim = Aer.get_backend('aer_simulator')
```

Change the Parameters (number of ancilla inputs and outputs, control gates etc)

Don't change variable names

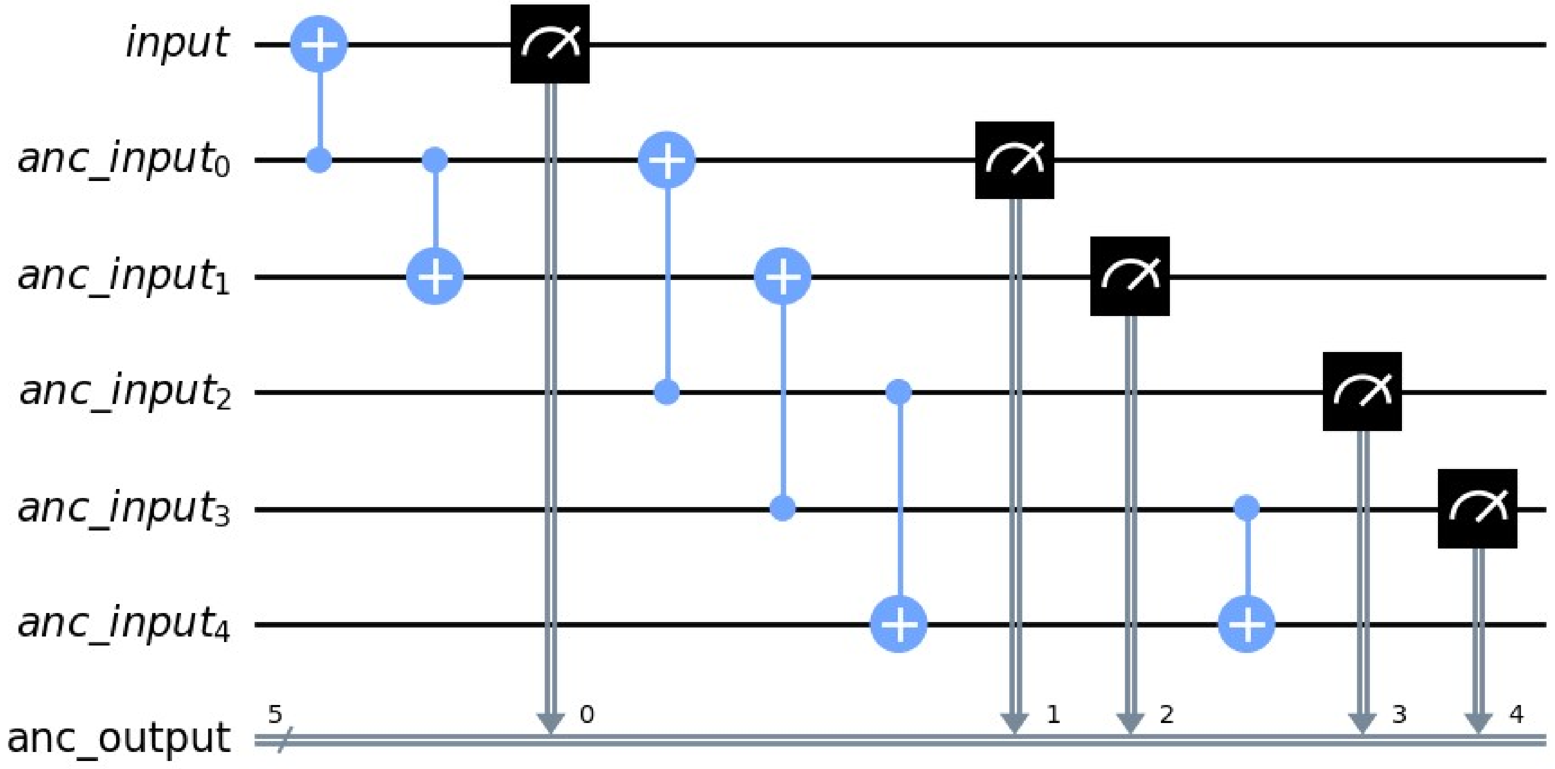
```
In [3]: %%file instruction_set.txt           ##RUN IT TO CREATE THE TXT FILE THAT CONTAINS THE CIRCUIT INFO##
        inp = QuantumRegister(1,'input')       #input register, change the number of qubits if you want
        anc = QuantumRegister(5,'anc_input')   #ancilla input register, change the qubits if you want
        cl = ClassicalRegister(5,'anc_output') #classical register for measurement purposes, change the bits if you want
        qc = QuantumCircuit(inp,anc,cl)
        qc.cx(1,0)
        qc.cx(1,2)                             ################################################
        qc.cx(3,1)                             # CHANGE THE GATES BUT TRY >1-QUBIT GATES ONLY #
        qc.cx(4,2)                             ################################################
        qc.cx(3,5)
        qc.cx(4,5)
        qc.measure([0,1,2,3,4],cl)             #change the qubits that are measured
        qc.draw(output='mpl')
```

```
UsageError: unrecognized arguments: ##RUN IT TO CREATE THE TXT FILE THAT CONTAINS THE CIRCUIT INFO##
```

Draw the Circuit, (any changes from the above cell must be made in this cell also)

In [240]:
```
inp = QuantumRegister(1,'input')
anc = QuantumRegister(5,'anc_input')
cl = ClassicalRegister(5,'anc_output')
qc = QuantumCircuit(inp,anc,cl)
qc.cx(1,0)
qc.cx(1,2)
qc.cx(3,1)
qc.cx(4,2)
qc.cx(3,5)
qc.cx(4,5)
qc.measure([0,1,2,3,4],cl)
qc.draw(output='mpl')
```

Out[240]:



Creating abstarct types (Node, Graph)

In [4]:
```
class Node:
    def __init__(self, type_str):
        self.label = type_str
        self.point = []
        self.head = None
        self.value = None

    def add_point(self, new_pointer):
        self.point.append(new_pointer)

    def get_type(self):
        return self.label

    def get_head(self):
        if self.head:
            return self.head
        else:
            return False



###########################################

class Graph:
    def __init__(self):
        self.count = 0
        self.q = []
        pass

    def connect(self, node_left, node_right):
        if node_left.get_type()[:-1] == 'input' and node_left.get_head():
            node = node_left.head
            node.add_point(node_right)
            node_left.head = node_right
            print(f"{node.get_type()}->{node_right.get_type()}")
        else:
            node_left.add_point(node_right)
            print(f"{node_left.get_type()}->{node_right.get_type()}")
            node_left.head = node_right

    def Q(self):
        return self.q

    def calculate_cost(self, node):
        self.count = 0
        cost = self.calculate_helper(node)
        return cost

    def calculate_helper(self, node):
        for p in node.point:
            if p.get_type()[:-1] == 'output':
                self.count += 1
                if len(p.point)>0:
                    self.calculate_helper(p)
                break

            else:
                self.calculate_helper(p)

        return self.count
```

Reading instruction_set.txt and translates txt file to graph

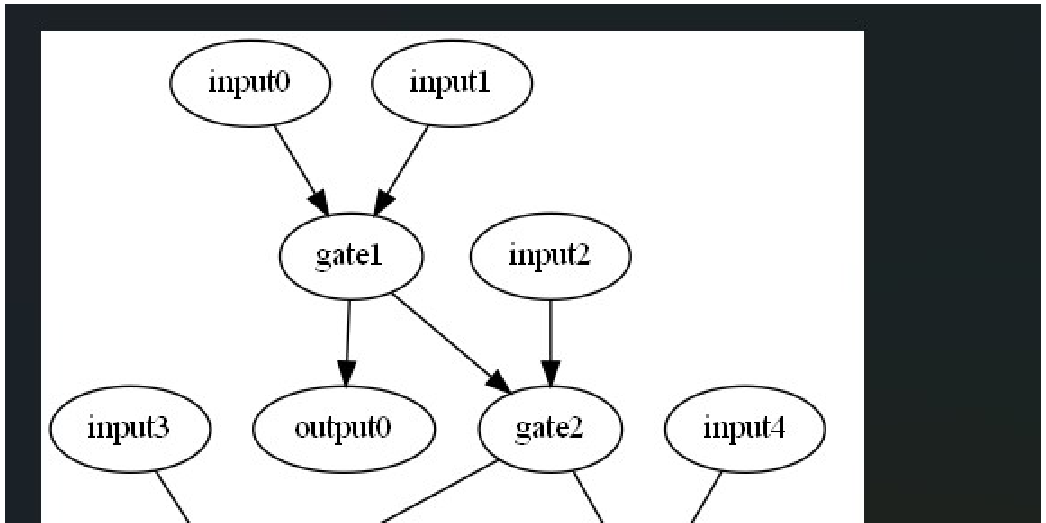
```
In [13]: circ = open('instruction_set.txt', "r")
         g = Graph()
         i = 0
         gate_num = 0
         for line in circ:
             if i == 0:
                 input_num = int(line[22])
                 input_nodes = [Node('input'+str(k)) for k in range(input_num)]
             elif i == 1:
                 ancilla_in = int(line[22])
                 ancin_nodes = [Node('input'+str(k+input_num)) for k in range(ancilla_in)]
             elif i == 2:
                 ancilla_out = int(line[23])
                 ancout_nodes = []
             elif i >= 4:
                 if line[3] == 'c':
                     gate_num += 1
                     target = int(line[-3])
                     control = int(line[-5])
                     target_node = (input_nodes[target] if target < input_num else ancin_nodes[target-input_num])
                     control_node = (input_nodes[control] if control < input_num else ancin_nodes[control-input_num])
                     gate = Node('gate'+str(i-4 + 1))

                     g.connect(target_node, gate)
                     g.connect(control_node, gate)
                     print('=============================')

                 elif line[3] == 'm':
                     indx = -2
                     for j in range(ancilla_out):
                         out_ind = int(line[-5 + indx])
                         out_node = (input_nodes[out_ind] if out_ind < input_num else ancin_nodes[out_ind-input_num])
                         output = Node('output'+str(out_ind))
                         #out_node.step.append(2)

                         g.connect(out_node, output)
                         indx += -2

             i += 1
```

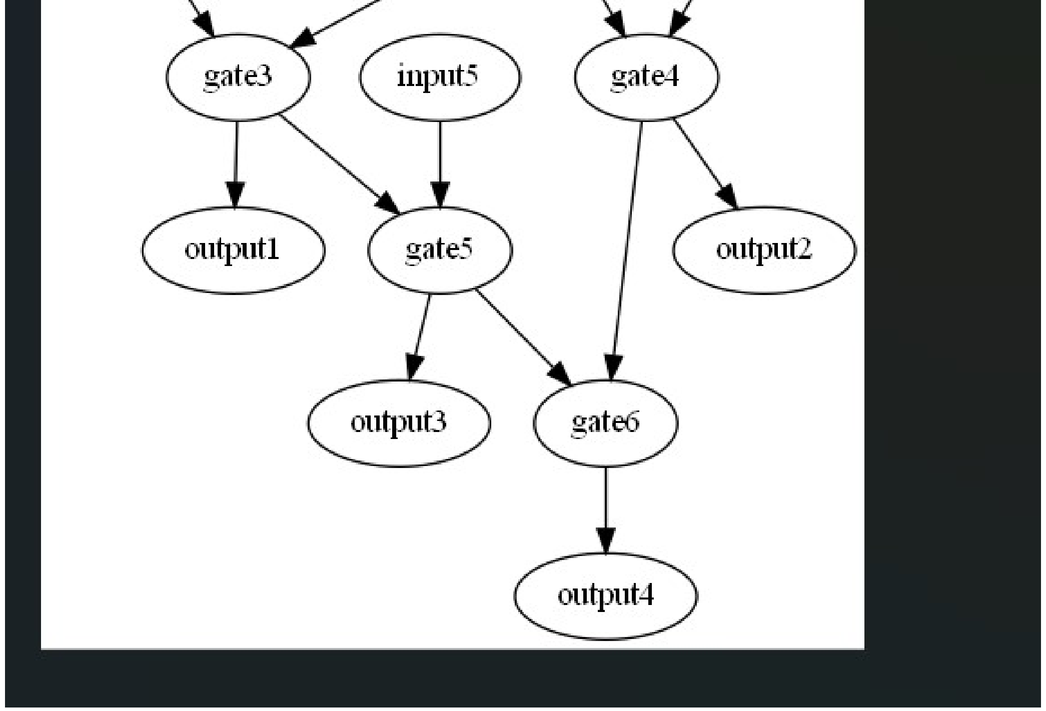
```
input0->gate1
input1->gate1
=============================
input2->gate2
gate1->gate2
=============================
gate2->gate3
input3->gate3
=============================
gate2->gate4
input4->gate4
=============================
input5->gate5
gate3->gate5
=============================
gate5->gate6
gate4->gate6
=============================
gate6->output4
gate5->output3
gate4->output2
gate3->output1
gate1->output0
```

Visualizing Graph that corresponds to the quantum circuit (install graphviz library first)

In [371]:
```
import graphviz

def add_nodes_edges(node, edges_dict):
    if len(node.point) > 0:
        dot.node(node.point[0].get_type())
        edge = (node.get_type(), node.point[0].get_type())
        if edge not in edges_dict:
            dot.edge(*edge)
            edges_dict.add(edge)
            add_nodes_edges(node.point[0], edges_dict)

    if len(node.point) > 1:
        dot.node(node.point[1].get_type())
        edge = (node.get_type(), node.point[1].get_type())
        if edge not in edges_dict:
            dot.edge(*edge)
            edges_dict.add(edge)
            add_nodes_edges(node.point[1], edges_dict)


dot = graphviz.Digraph()
dot.node(input_nodes[0].get_type())
add_nodes_edges(input_nodes[0], set())
if input_num > 1:
    for i in range (input_num):
        dot.node(input_nodes[i+1].get_type())
        dot.edge(input_nodes[i+1].get_type(), input_nodes[i+1].point[0].get_type())
for i in range (ancilla_in):
    dot.node(ancin_nodes[i].get_type())
    dot.edge(ancin_nodes[i].get_type(), ancin_nodes[i].point[0].get_type())
dot.render('graph', view=True, format='png')
```

Out[371]:
```
'graph.png'
```





Algorithm

Finding which input node has fewer ancilla outputs as successor nodes

In [14]:
```
node_list = []
for i in range(ancilla_in):
    ancin_nodes[i].value = ancilla_out - g.calculate_cost(ancin_nodes[i])
    node_list.append(ancin_nodes[i])

sorted_nodes = list(reversed(sorted(node_list, key=lambda x: x.value)))
g.q = sorted_nodes
print(' QUEUE')
for k in sorted_nodes:
    print(k.get_type())
```

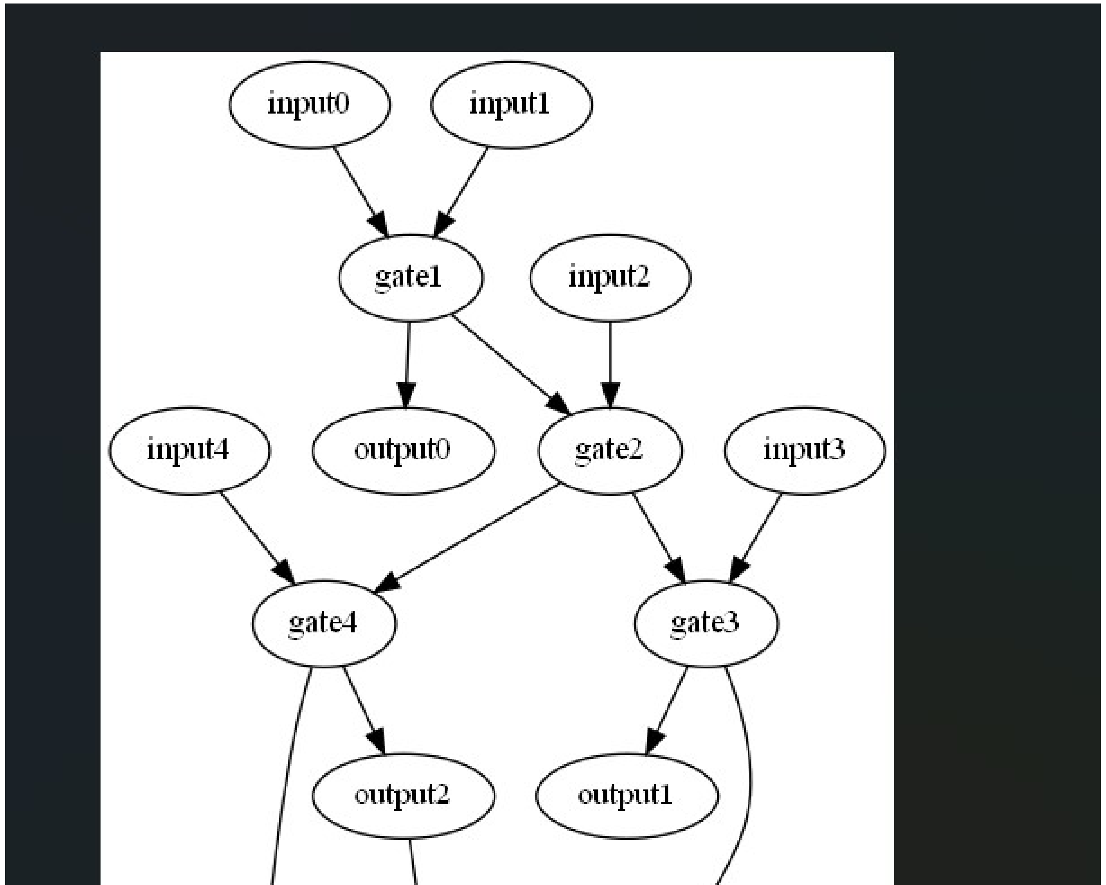
```
 QUEUE
input5
input4
input3
input2
input1
```

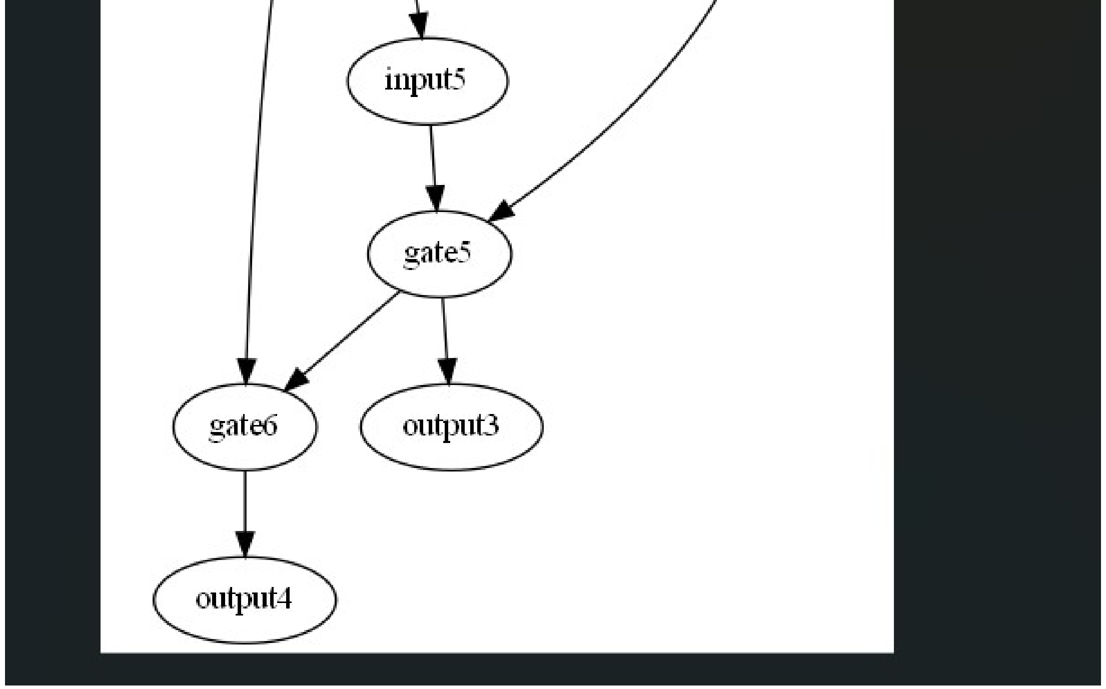
Functions to run the Algorithm

In [15]:
```
def output_helper(node, pList):
    for p in node.point:
        if p.get_type()[:-1] == 'output':
            pList.append(p)
            if len(p.point)>0:
                output_helper(p, pList)
            break

        else:
            output_helper(p, pList)

def precending_outputs(node):
    n_list = []
    output_helper(node, n_list)
    pr_list = []
    for x in node_list:
        if node == x:
            continue
        else:
            temp_list = []
            output_helper(x, temp_list)
            difference_result = [item for item in temp_list if item not in n_list]
            break

    pr_list = difference_result

    #print(pr_list[0].get_type())
    if len(pr_list)<1:
        return False

    return pr_list[0]

def new_cost():
    for i in range(len(g.Q())):
        g.Q()[i].value = ancilla_out - g.calculate_cost(g.Q()[i])

    sorted_nodes = list(reversed(sorted(g.Q(), key=lambda x: x.value)))
    g.q = sorted_nodes


def algorithm():
    for a in g.Q():
        #a = g.Q()[0]
        #print(a.get_type())
        current_node = a
        g.Q().remove(a)
        p_o = precending_outputs(current_node)
        if not p_o:
            break
        node_list.remove(current_node)
        ancin_nodes.remove(current_node)
        g.connect(p_o, current_node)

        new_cost()

###############################################################################
node_list = []
for i in range(ancilla_in):
    ancin_nodes[i].value = ancilla_out - g.calculate_cost(ancin_nodes[i])
    node_list.append(ancin_nodes[i])

sorted_nodes = list(reversed(sorted(node_list, key=lambda x: x.value)))
g.q = sorted_nodes

algorithm()
```

```
output1->input5
output2->input3
output0->input2
```

First iteration



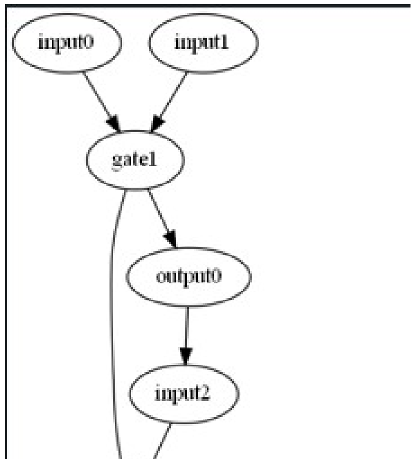


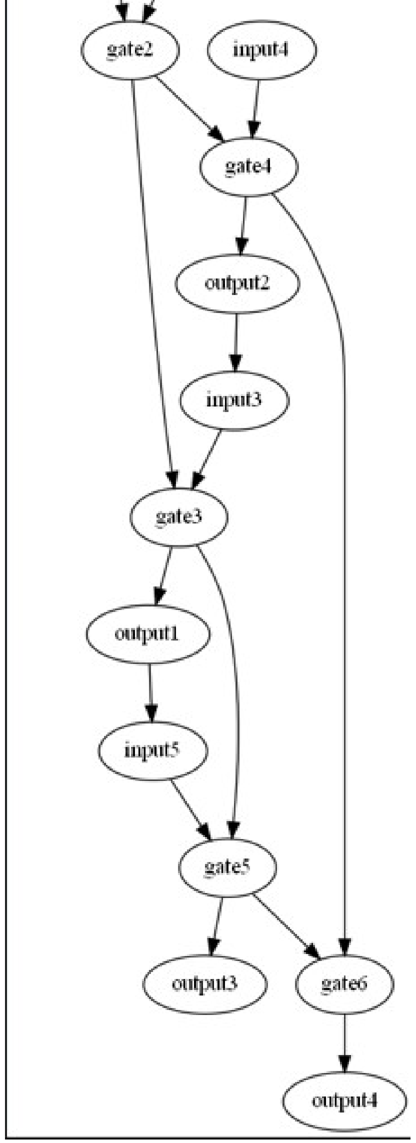
Visualizing the Result of the Algorithm

In [16]:
```
import graphviz

def add_nodes_edges(node, edges_dict):
    if len(node.point) > 0:
        dot.node(node.point[0].get_type())
        edge = (node.get_type(), node.point[0].get_type())
        if edge not in edges_dict:
            dot.edge(*edge)
            edges_dict.add(edge)
            add_nodes_edges(node.point[0], edges_dict)

    if len(node.point) > 1:
        dot.node(node.point[1].get_type())
        edge = (node.get_type(), node.point[1].get_type())
        if edge not in edges_dict:
            dot.edge(*edge)
            edges_dict.add(edge)
            add_nodes_edges(node.point[1], edges_dict)


dot = graphviz.Digraph()
dot.node(input_nodes[0].get_type())
add_nodes_edges(input_nodes[0], set())
if input_num > 1:
    for i in range (input_num):
        dot.node(input_nodes[i+1].get_type())
        dot.edge(input_nodes[i+1].get_type(), input_nodes[i+1].point[0].get_type())
for an in ancin_nodes:
    dot.node(an.get_type())
    dot.edge(an.get_type(), an.point[0].get_type())
dot.render('graph2', view=True, format='png')
```

Out[16]:
```
'graph2.png'
```

Final Optimal Result



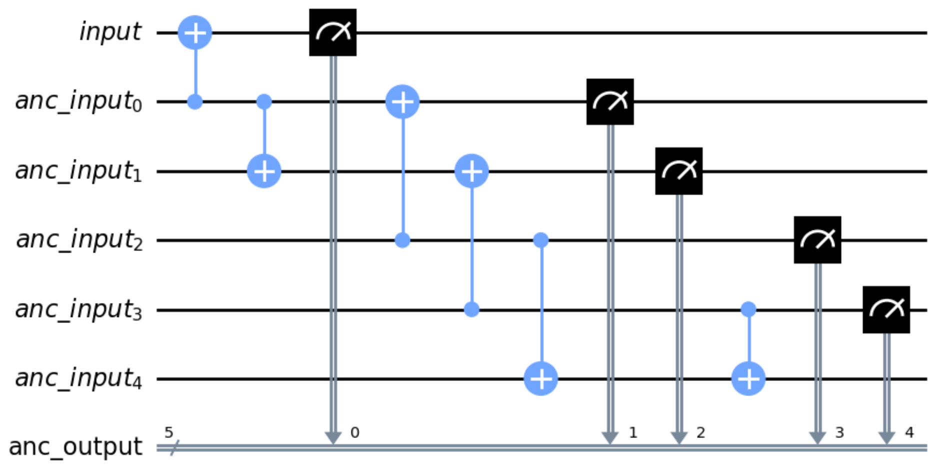

gate2
input4
gate4
output2
input3
gate3
output1
input5
gate5
output3
gate6
output4

```
In [ ]:
```

Explaining the “Notebook”

1) Suppose the initial quantum circuit

```
inp = QuantumRegister(1,'input')
anc = QuantumRegister(5,'anc_input')
cl = ClassicalRegister(5,'anc_output')
qc = QuantumCircuit(inp,anc,cl)
qc.cx(1,0)
qc.cx(1,2)
qc.cx(3,1)
qc.cx(4,2)
qc.cx(3,5)
qc.cx(4,5)
qc.measure([0,1,2,3,4],cl)
qc.draw(output='mpl')
```

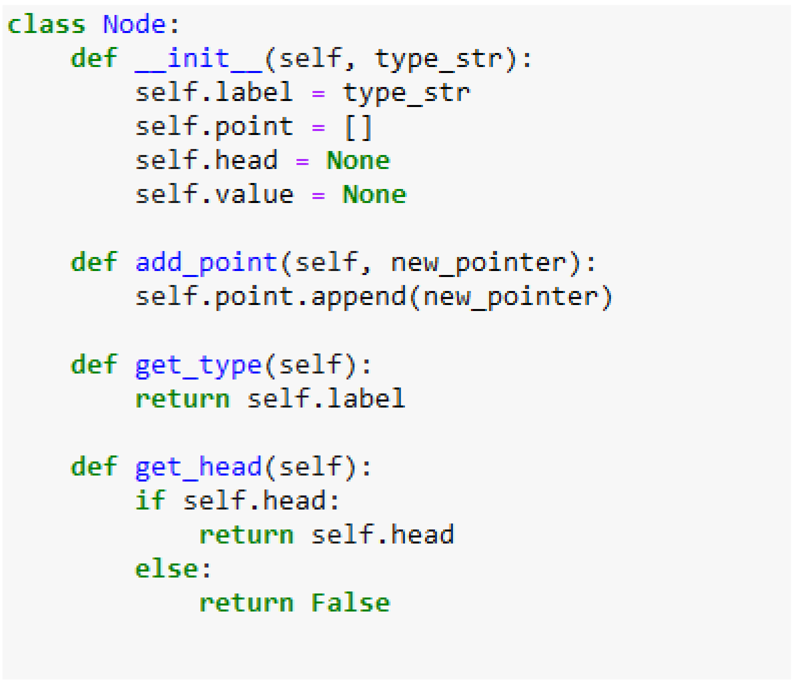


2)

```
%%file instruction_set.txt
inp = QuantumRegister(1,'input')
anc = QuantumRegister(5,'anc_input')
cl = ClassicalRegister(5,'anc_output'
qc = QuantumCircuit(inp,anc,cl)
qc.cx(1,0)
qc.cx(1,2)
qc.cx(3,1)
qc.cx(4,2)
qc.cx(3,5)
qc.cx(4,5)
qc.measure([0,1,2,3,4],cl)
qc.draw(output='mpl')
```

We use the “magic” command %%file in order the write everything this cell contains into a txt file

3) Create Node class



```python
class Node:
    def __init__(self, type_str):
        self.label = type_str
        self.point = []
        self.head = None
        self.value = None

    def add_point(self, new_pointer):
        self.point.append(new_pointer)

    def get_type(self):
        return self.label

    def get_head(self):
        if self.head:
            return self.head
        else:
            return False
```

- Node can be input, gate or output
- Get type returns the name of the node (input1, gate3 etc) if we want to find the actual we use list slices: node.get_type()[:-1]
- Add.point function creates the arrow in the graph

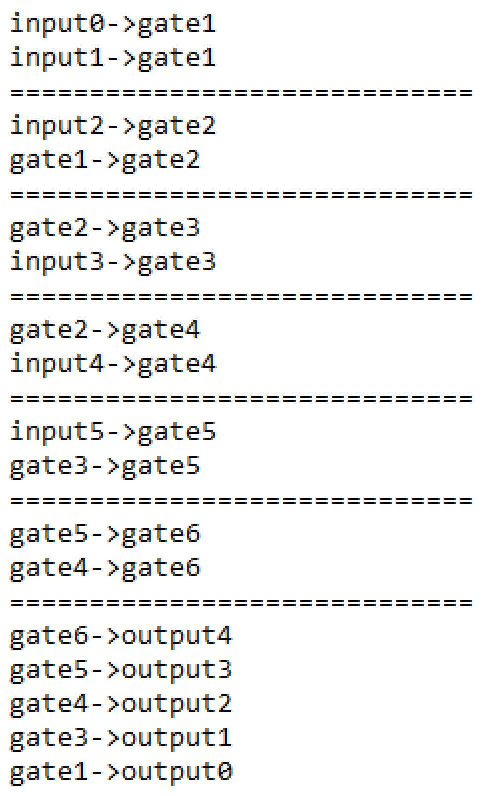
4) Create Graph data structure

```
class Graph:
    def __init__(self):
        self.count = 0
        self.q = []
        pass

    def connect(self, node_left, node_right):
        if node_left.get_type()[:-1] == 'input' and node_left.get_head():
            node = node_left.head
            node.add_point(node_right)
            node_left.head = node_right
            print(f"{node.get_type()}->{node_right.get_type()}")
        else:
            node_left.add_point(node_right)
            print(f"{node_left.get_type()}->{node_right.get_type()}")
            node_left.head = node_right

    def Q(self):
        return self.q

    def calculate_cost(self, node):
        self.count = 0
        cost = self.calculate_helper(node)
        return cost

    def calculate_helper(self, node):
        for p in node.point:
            if p.get_type()[:-1] == 'output':
                self.count += 1
                if len(p.point)>0:
                    self.calculate_helper(p)
                break

            else:
                self.calculate_helper(p)

        return self.count
```

- Q() is the priority Queue that contains the ancilla inputs according to their node.value
- The node.value is the sum of the ancilla-outputs – the calculate cost
- Calculate cost is the number of the outputs after the node (the outputs that the input leads to in the graph)

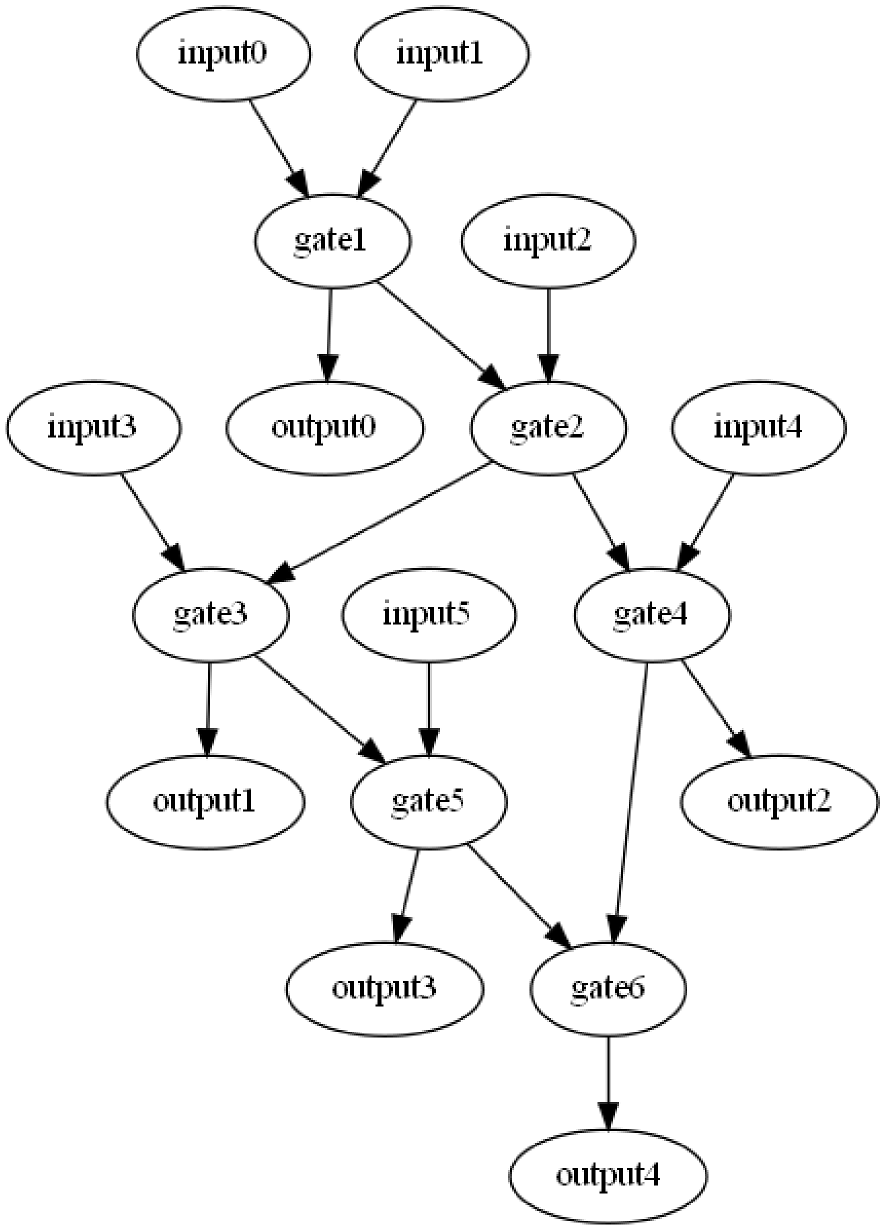
- !!! When a connection occurs:
  A gate g1 is applied on a qubit **(input node)** then the input-node.head becomes the gate g1. **Then the graph draws the arrow input-node -> g1.** When a gate g2 is applied on the input-node again it is applied on the input-gate.head , **thus g2 is applied on the outcome of g1 and the graph draws the arrow g1->g2**

5) The program reads the txt file and transforms the circuit into graph. The output shows the connections

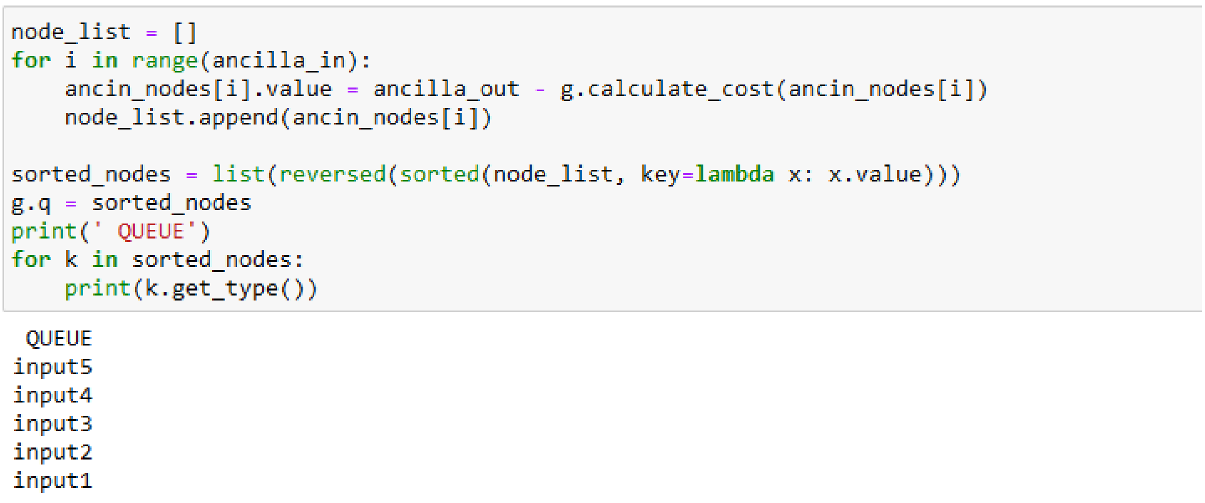


```
input0->gate1
input1->gate1
=============================
input2->gate2
gate1->gate2
=============================
gate2->gate3
input3->gate3
=============================
gate2->gate4
input4->gate4
=============================
input5->gate5
gate3->gate5
=============================
gate5->gate6
gate4->gate6
=============================
gate6->output4
gate5->output3
gate4->output2
gate3->output1
gate1->output0
```

6) The graph is now:



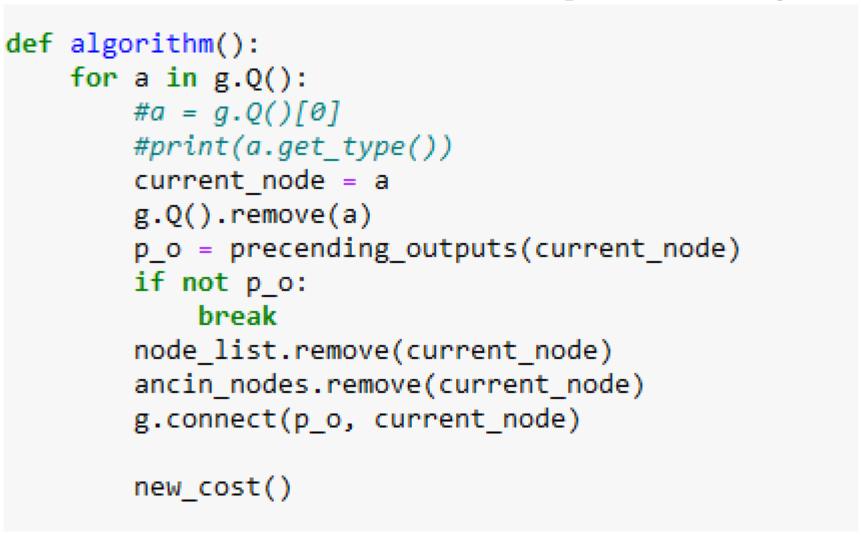
7) Now we execute the algorithm.
   First of all we should sort the ancilla inputs in order to create the priority queue of the graph



```
node_list = []
for i in range(ancilla_in):
    ancin_nodes[i].value = ancilla_out - g.calculate_cost(ancin_nodes[i])
    node_list.append(ancin_nodes[i])

sorted_nodes = list(reversed(sorted(node_list, key=lambda x: x.value)))
g.q = sorted_nodes
print(' QUEUE')
for k in sorted_nodes:
    print(k.get_type())
```

```
 QUEUE
input5
input4
input3
input2
input1
```

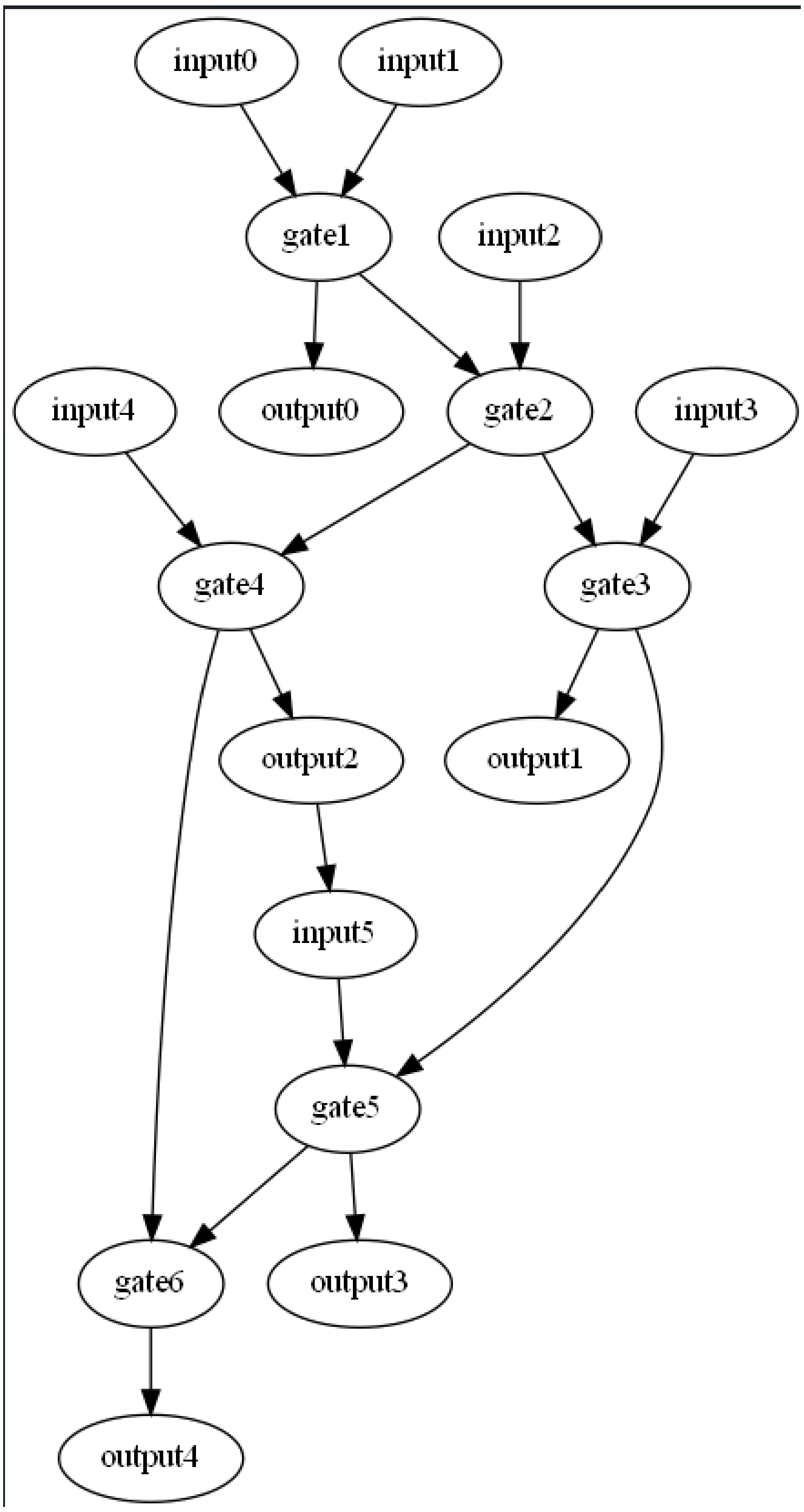
8) We create the function that runs the optimization algorithm



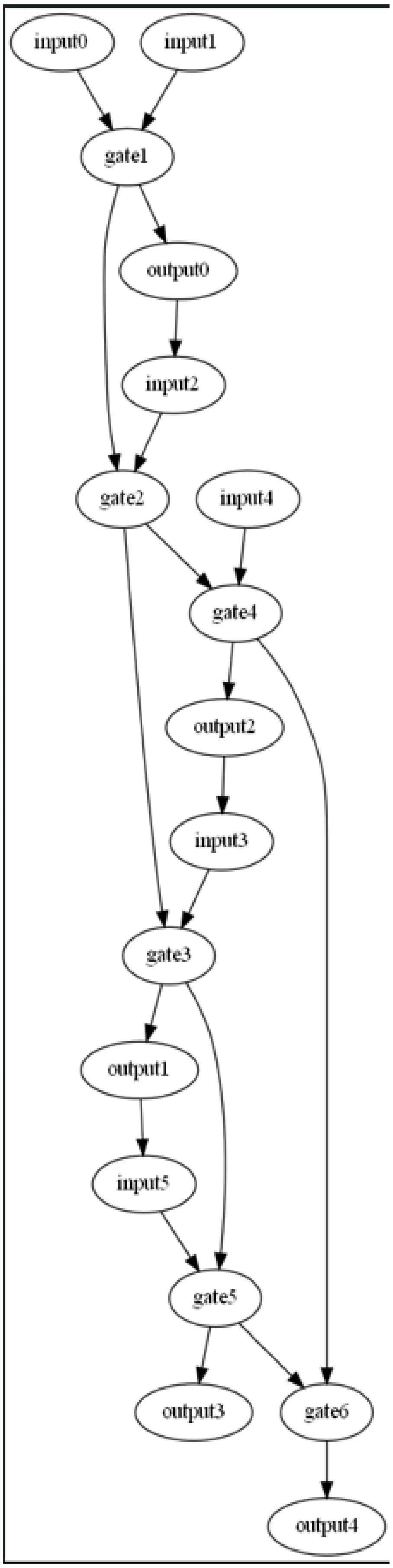
```
def algorithm():
    for a in g.Q():
        #a = g.Q()[0]
        #print(a.get_type())
        current_node = a
        g.Q().remove(a)
        p_o = precending_outputs(current_node)
        if not p_o:
            break
        node_list.remove(current_node)
        ancin_nodes.remove(current_node)
        g.connect(p_o, current_node)

        new_cost()
```

We iterate through the queue. Keep in mind that the queue gets updated after the end of each iteration. **First the first element pops, then we search for the preceding output. Namely we search for the measurement that happened just before the current input initialization. After that we connect the output to the current input and we update the Q and continue until there aren't preceding outputs to the inputs.**

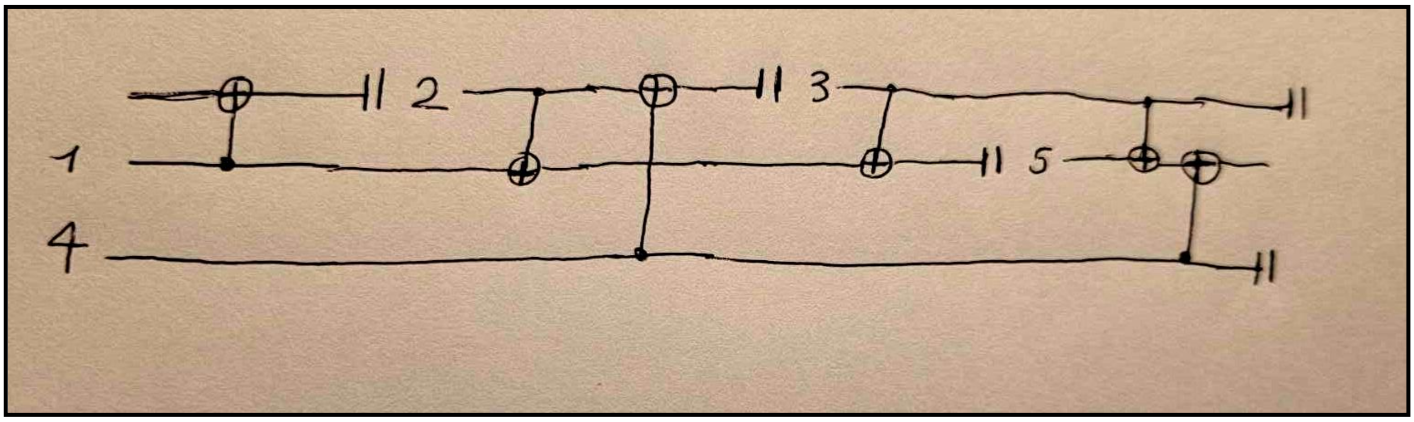
After the first loop



The final graph is


input0
input1
gate1
output0
input2
gate2
input4
gate4
output2
input3
gate3
output1
input5
gate5
output3
gate6
output4

This graph represents this circuit:



Circuit Cutting

Theory

Configuring the Basis

Cutting a qubit wire originates from the fact that the unitary matrix of an arbitrary quantum operation in a quantum circuit can be decomposed into any set of orthonormal matrix bases.

For example, the set of Pauli matrices I, X, Y, Z is a convenient basis to use.

Specifically, an arbitrary 2×2 matrix A can be decomposed as:

$$A = \frac{Tr(AI)I + Tr(AX)X + Tr(AY)Y + Tr(AZ)Z}{2}$$

This approach, however, requires access to complex amplitudes, which are not available on quantum computers.

To overcome this obstacle we further decompose the Pauli matrices into their eigenbasis and organize the terms.

$$A = \frac{A1 + A2 + A3 + A4}{2}$$

$$\begin{gathered} where \\ A1 = [Tr(AI) + Tr(AZ)]\, |0\rangle \langle 0| \\ A2 = [Tr(AI) - Tr(AZ)]\, |1\rangle \langle 1| \\ A3 = Tr(AX)[2\, |+\rangle \langle +| - |0\rangle \langle 0| - |1\rangle \langle 1|] \\ A4 = Tr(AY)[2\, |\uparrow\rangle \langle \uparrow| - |0\rangle \langle 0| - |1\rangle \langle 1|] \end{gathered}$$

- Each trace operator corresponds physically to measure the qubit in one of the Pauli bases.
- Each of the density matrices corresponds physically to initialize the qubit in one of the eigenstates.

Challenges

- The first challenge is to find cut locations. While quantum circuits can always be split into smaller ones, **large quantum circuits may require more than one cut in order to be separated into subcircuits**. For general quantum circuits with n quantum edges, this task faces an O $(n!)$ combinatorial search

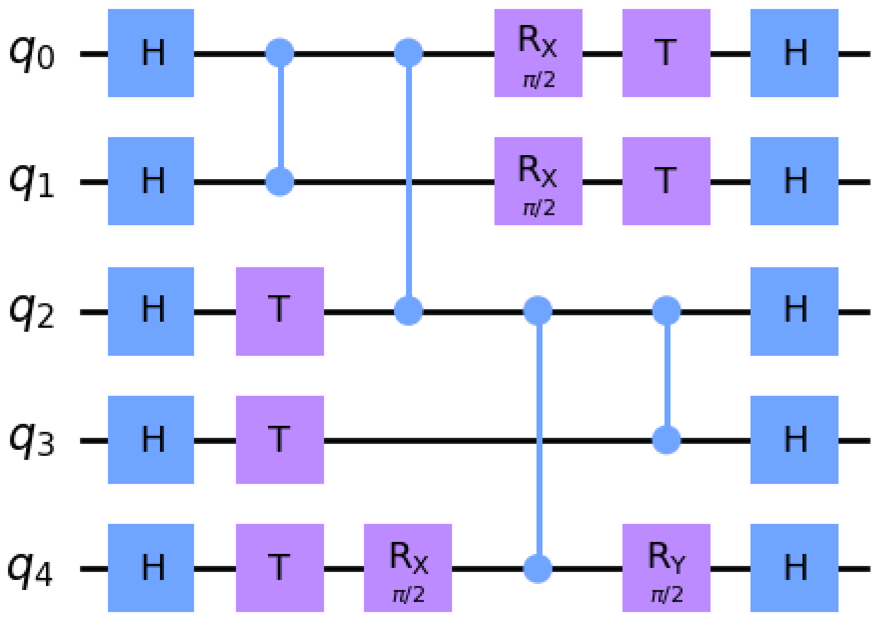
- The second challenge is to scale the results. Large quantum circuits have exponentially increasing state space that quickly **becomes intractable to even store the full-state probabilities.**

Cutting process

Suppose the initial 5-qubit circuit. The qubits are initialized in the state:

|q0 q1 q2 q3 q4 >, where $qi \in \{|0\rangle, |1\rangle, |+\rangle, |+i\rangle\}$

Let the output be measured in the $M0, \ldots Mn{-}1$ basis, where $Mi \in \{I, X, Y, Z\}$.

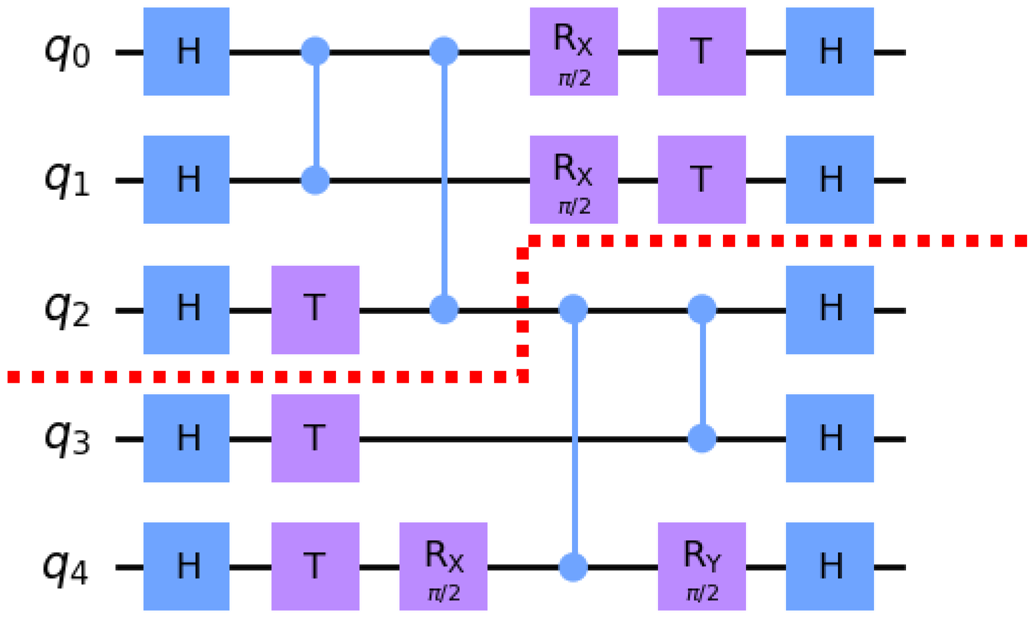


Initial state = |00000⟩

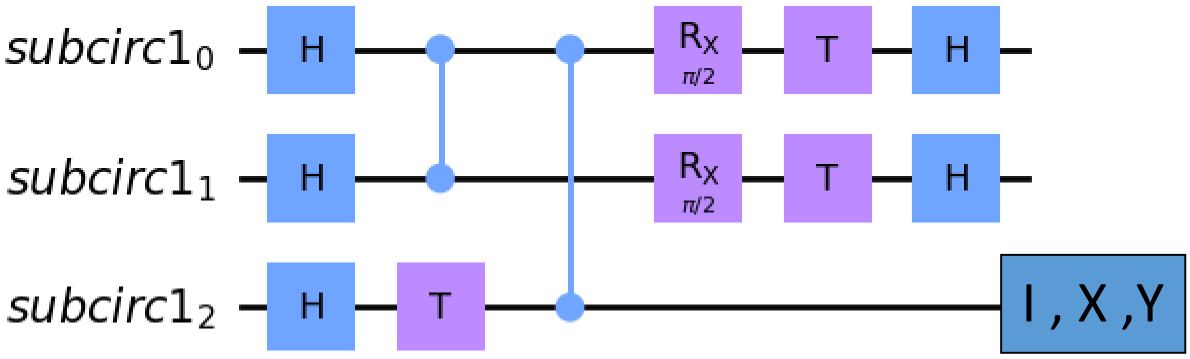
Selecting cut locations

The cut location is chosen manually.

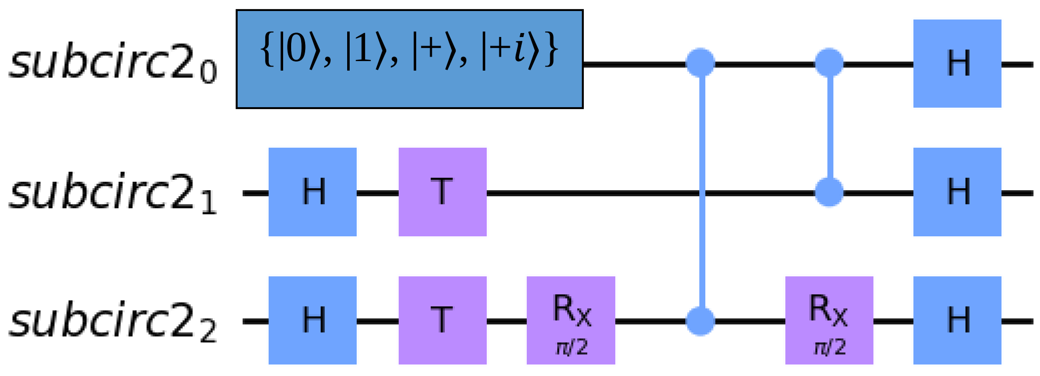
The five-qubit circuit is cut into two smaller subcircuits of three qubits each. The subcircuits are produced by cutting the $q2$ wire between the first two cZ gates.



Subcircuit 1 has 3 variations:

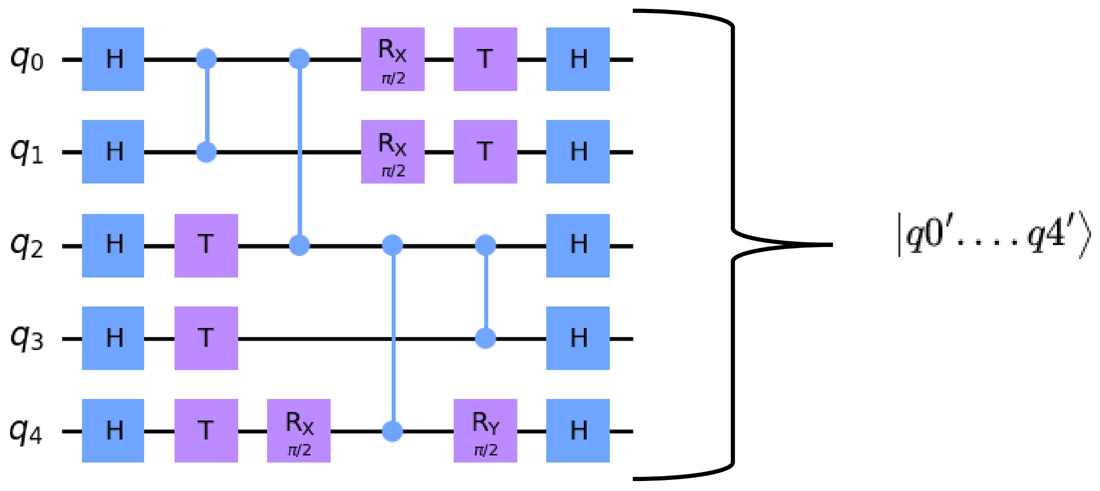


Subcircuit 2 also 3 variations:



Building the full probabilities

The uncut circuit results to the state:



The final state contains 2^5 results.

Let's calculate the probability the final state to be |01010⟩

For the subcirc1 we know that *subcirc*12 does not appear in the final output of the uncut circuit.

The relevant state of the subcirc1 is |01>

The probabilities are:

$p1{,}1 = p(|010\rangle\ |I) + p(|011\rangle\ |I) + p(|010\rangle\ |Z) - p(|011\rangle\ |Z)$

$p1{,}2 = p(|010\rangle\ |I) + p(|011\rangle\ |I) - p(|010\rangle\ |Z) + p(|011\rangle\ |Z)$

$p1{,}3 = p(|010\rangle\ |X) - p(|011\rangle\ |X)$

$p1{,}4 = p(|010\rangle\ |Y) - p(|011\rangle\ |Y)$.

The relevant state of subcircuit 2 is $|010\rangle$. Hence, its four terms are

$p2{,}1 = p(|010\rangle \mid |0\rangle)$

$p2{,}2 = p(|010\rangle \mid |1\rangle)$

$p2{,}3 = 2p(|010\rangle \mid |+\rangle) - p(|010\rangle \mid |0\rangle) - p(|010\rangle \mid |1\rangle)$

$p2{,}4 = 2p(|010\rangle \mid |i\rangle) - p(|010\rangle \mid |0\rangle) - p(|010\rangle \mid |1\rangle)$.

The final reconstructed probability of the uncut state |01010⟩ is:

$p(|01010\rangle) = \dfrac{1}{2}\displaystyle\sum_{i=1}^{4} p_{1,i} \otimes p_{2,i}$

Simulating the 2 subcircuits

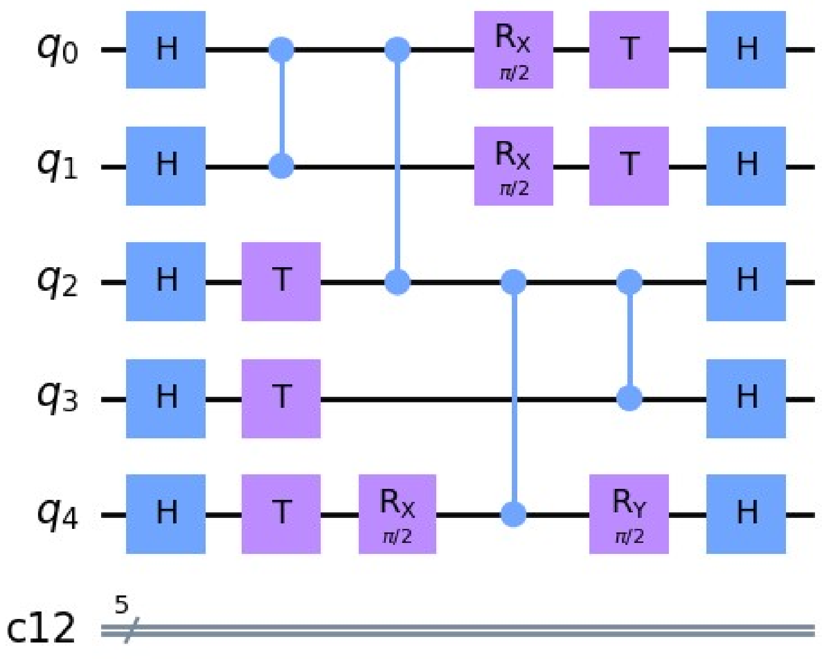
In [93]:
```
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_bloch_multivector, plot_histogram, array_to_latex
from qiskit.tools.visualization import circuit_drawer
from qiskit.quantum_info import Statevector
from numpy import pi
from qiskit.transpiler import PassManager
from qiskit.transpiler.passes import Unroller
import random
import networkx as nx
import matplotlib.pyplot as plt

sim = Aer.get_backend('aer_simulator')
```

Initial circuit

copy the cell

In [99]:
```
q = QuantumRegister(5,'q')
cl = ClassicalRegister(5)
qc = QuantumCircuit(q,cl)
qc.h(q)
qc.cz(0,1)
qc.t([2,3,4])
qc.cz(0,2)
qc.rx(pi/2,4)
qc.rx(pi/2,0)
qc.cz(2,4)
qc.rx(pi/2,1)
qc.t([0,1])
qc.cz(2,3)
qc.ry(pi/2,4)
qc.h(q)
qc.draw(output='mpl')
```

Out[99]:



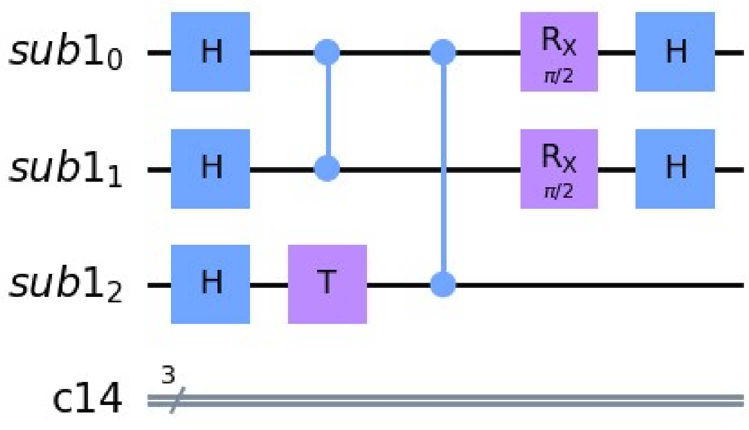
Configure Cuts

paste the circuit's instructions and create the subcircuits

First subcircuit

In [101]:
```
q1 = QuantumRegister(3,'sub1')
cl1 = ClassicalRegister(3)
qc1 = QuantumCircuit(q1,cl1)
qc1.h(q1)
qc1.cz(0,1)
qc1.t([2])
qc1.cz(0,2)
qc1.rx(pi/2,0)
qc1.rx(pi/2,1)
qc1.h([0,1])
qc1.draw(output='mpl')
```

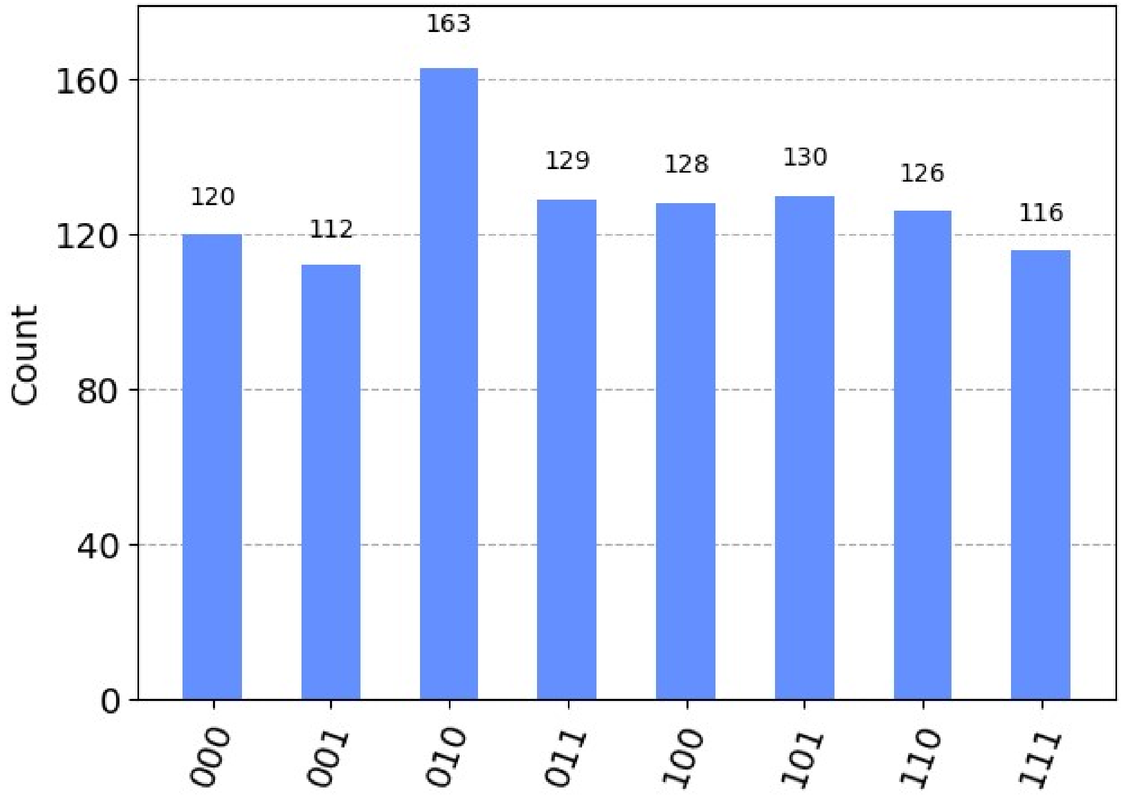
Out[101]:



Measure subcircuit 1

In [102]:
```
qc1.measure(q1,cl1)
qobj = assemble(qc1)  # Assemble circuit into a Qobj that can be run
counts1 = sim.run(qobj).result().get_counts()  # Do the simulation, returning the state vector
plot_histogram(counts1)  # Display the output on measurement of state vector
```

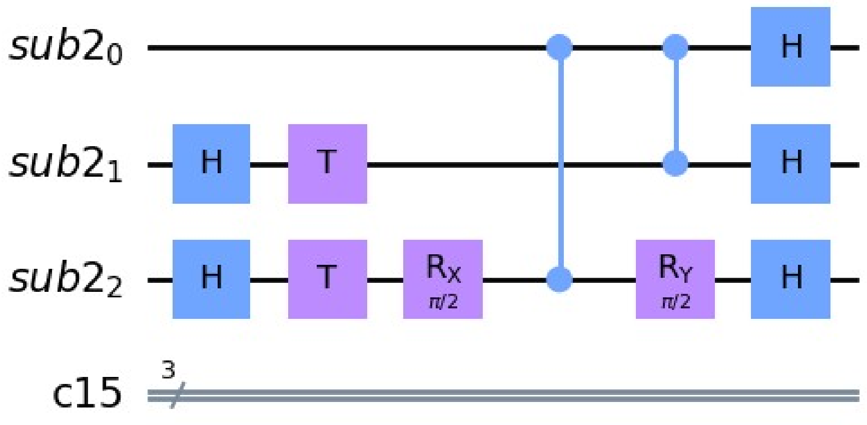
Out[102]:



Second subcircuit

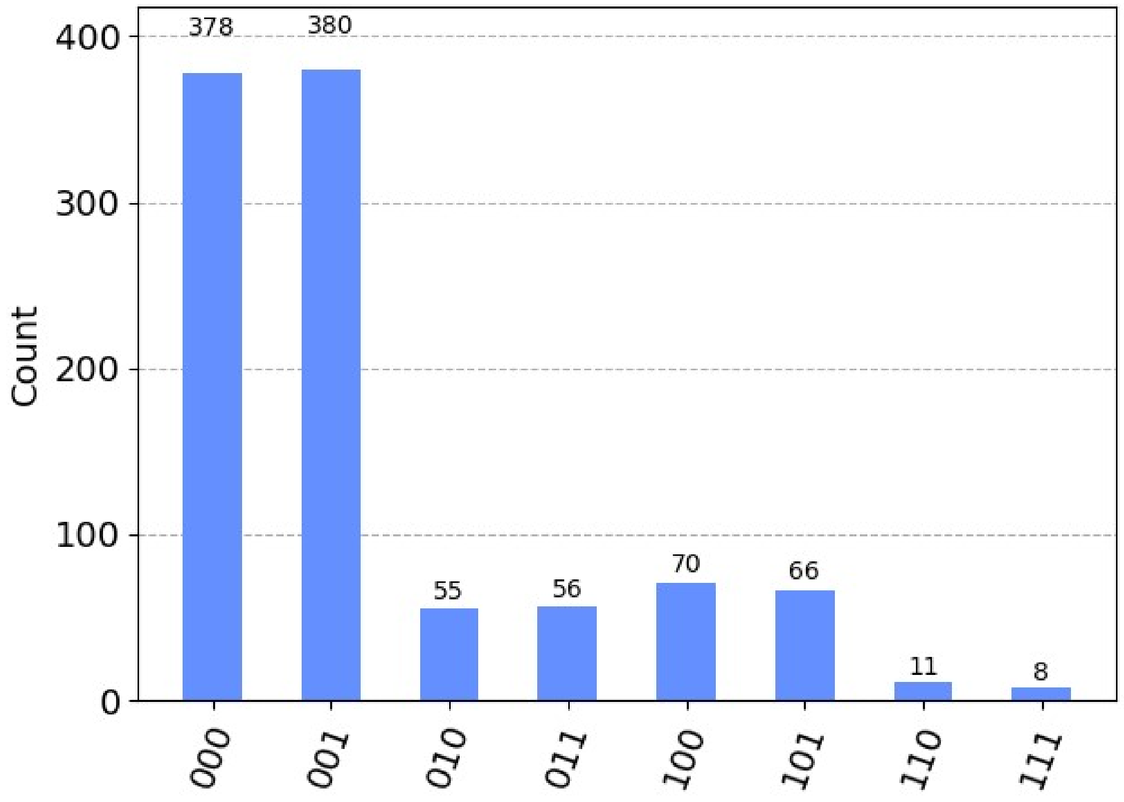
In [103]:
```
q2 = QuantumRegister(3,'sub2')
cl2 = ClassicalRegister(3)
qc2 = QuantumCircuit(q2,cl2)
qc2.h([1,2])
qc2.t([1,2])
qc2.rx(pi/2,2)
qc2.cz(0,2)
qc2.cz(0,1)
qc2.ry(pi/2,2)
qc2.h(q2)
qc2.draw(output='mpl')
```

Out[103]:



In [104]:
```
qc2.measure(q2,cl2)
qobj = assemble(qc2)  # Assemble circuit into a Qobj that can be run
counts2 = sim.run(qobj).result().get_counts()  # Do the simulation, returning the state vector
plot_histogram(counts2)  # Display the output on measurement of state vector
```

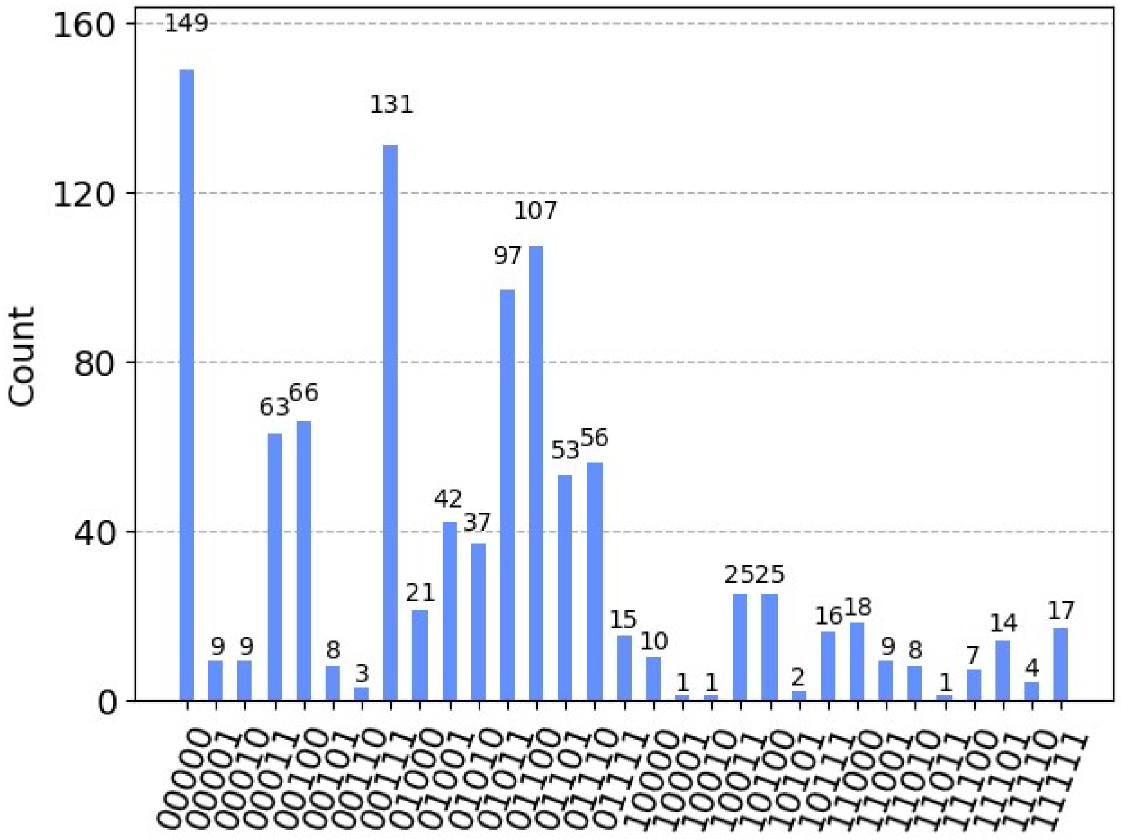
Out[104]:



Display measurements of the uncut initial circuit

In [105]:
```
qc.measure(q,cl)
qobj = assemble(qc)  # Assemble circuit into a Qobj that can be run
counts = sim.run(qobj).result().get_counts()  # Do the simulation, returning the state vector
plot_histogram(counts)  # Display the output on measurement of state vector
```

Out[105]:



Compare the resluts

In [127]:
```
# calculate the measurement result of the circuits
sub_prob = []
for p in counts1.keys():
    temp1 = counts1[p] / 1000  #normalize probabilities
    for p2 in counts2.keys():
        temp2 = counts2[p2] / 1000  #normalize probabilities
        sub_prob.append(temp1*temp2/2)
####################################################################
#calculate the measurement result of the uncut circuit
prob = []
for pr in counts.keys():
    prob.append(counts[pr]/1000)
###################################################################
#calculate error rate
rate = []
for i in range(len(prob)):
    temp = abs(sub_prob[i] - prob[i])
    rate.append(temp)
###################################################################
final = sum(rate)/len(rate)
print(f"Error rate is: {final*100} %")
```

```
Error rate is: 2.9245096774193544 %
```

In [ ]:

Simulation Results

When we measure the qubits of all the circuits (2 subcircuits and the uncut circuit)

We calculate the joint probability for the subcircuits:

```
# calculate the measurement result of the circuits
sub_prob = []
for p in counts1.keys():
    temp1 = counts1[p] / 1000  #normalize probabilities
    for p2 in counts2.keys():
        temp2 = counts2[p2] / 1000  #normalize probabilities
        sub_prob.append(temp1*temp2/2)
```

Using the formula $\frac{1}{2}\sum_{i=1}^{4} p_{1,i} \otimes p_{2,i}$.

After that we calculate the absolute difference for each possible measurement result between the subcircuits and the initial circuit

```
#calculate error rate
rate = []
for i in range(len(prob)):
    temp = abs(sub_prob[i] - prob[i])
    rate.append(temp)
#####################################################################
```

And finally it is printed the error rate percentage:

```
Error rate is: 2.9245096774193544 %
```

Sources

https://arxiv.org/pdf/1609.00803.pdf

https://arxiv.org/ftp/arxiv/papers/2012/2012.02333.pdf