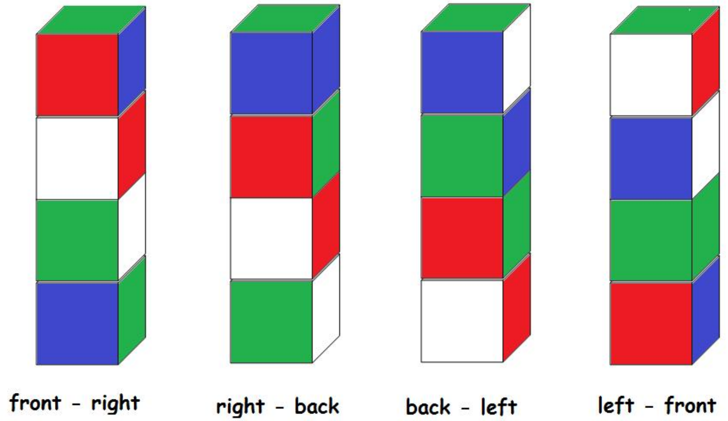
Instant Insanity Puzzle: QAOA Approach

By: Sisir Pynda & Martin Carignan

What is the Instant Insanity Puzzle?

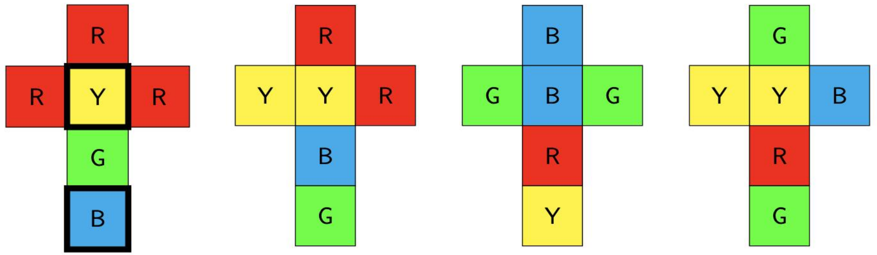
- Instant Insanity is a graphing problem puzzle.
- You are given four cubes that have different colors on each side, however, there are up to four unique colors in this entire puzzle.
- The objective of this game is to stack the cubes in a way that each cube on the face of a stack does not have a repeating color.

3D Visualization: https://instantinsanity.z20.web.core.windows.net/

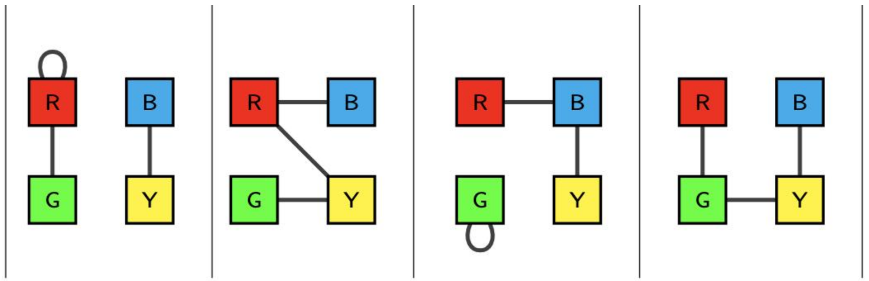


Graphing Approach

- Draw a graph for each cube, connecting opposite faces



Cube 1 Cube 2 Cube 3 Cube 4

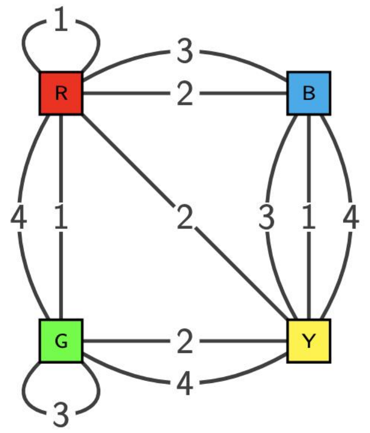


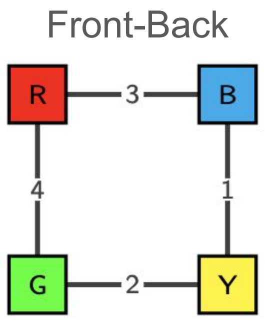
Cube 1 Graph Cube 2 Graph Cube 3 Graph Cube 4 Graph

Graphing Approach (cont.)

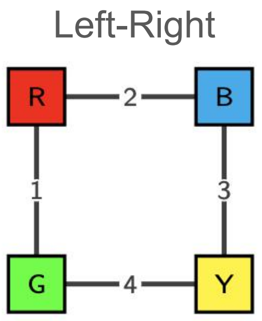
Steps:

- Put all 4 graphs from Step 1 into a single graph
- Find 2 subgraphs using specific instructions:
  - The 2 subgraphs have no edges in common
  - Each subgraph contains exactly one edge from each cube
  - Each node only has 2 edges connected to it



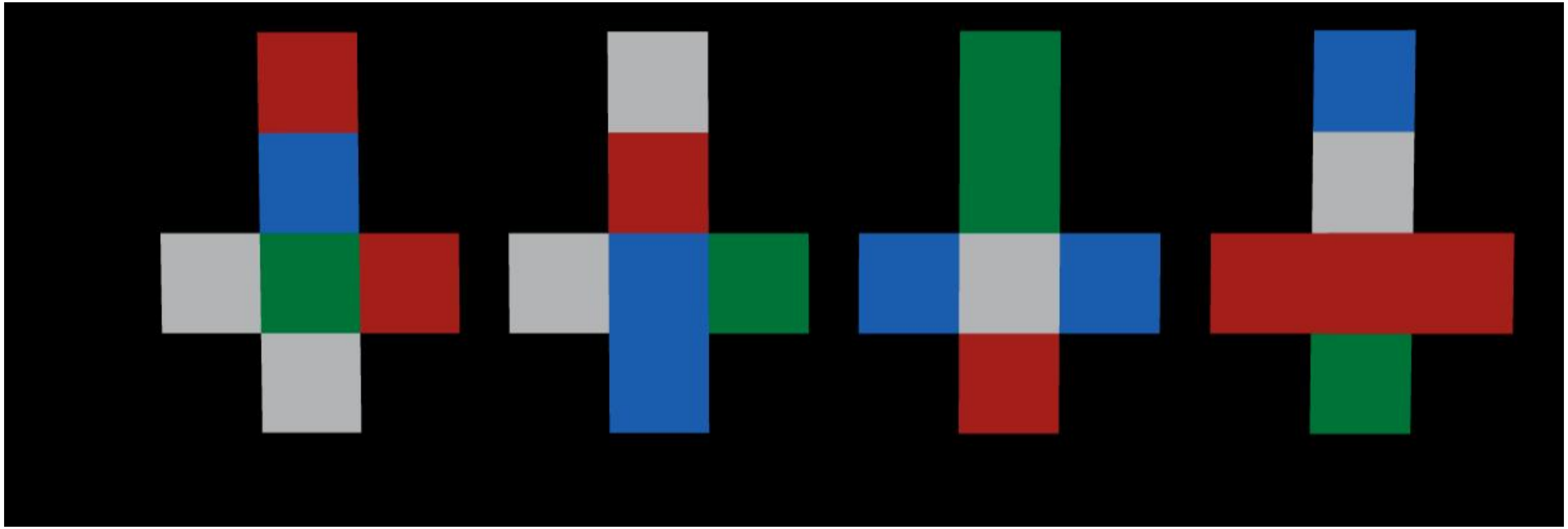


Subgraph 1



Subgraph 2

The Problem Cube



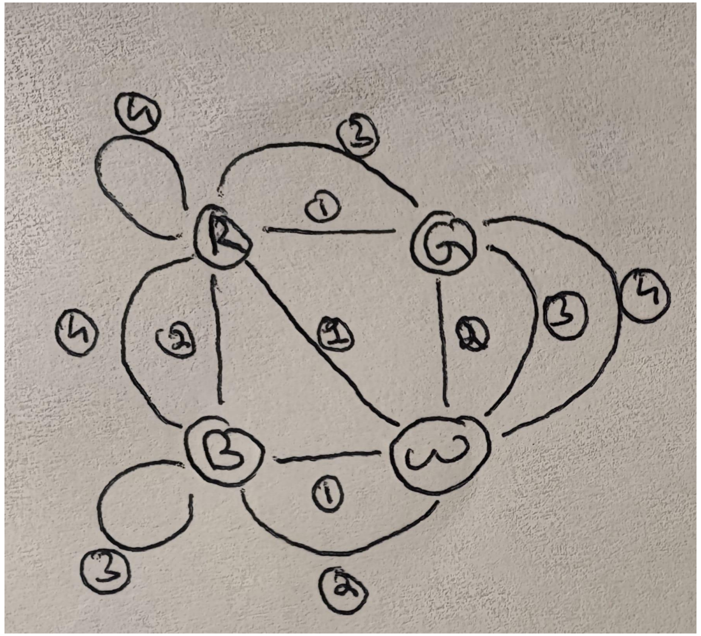
Interactive Visualization

Modeling the Problem

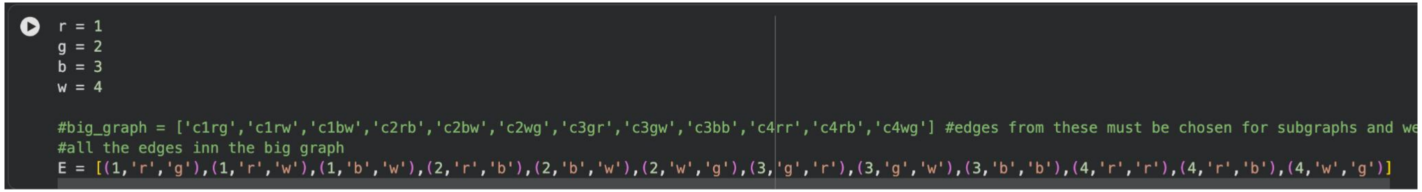
This Problem was solved using graph theory.

Nodes: Colours white, green, reg,blue

Edges: Cube 1,2,3,4



4 cube system



```
r = 1
g = 2
b = 3
w = 4

#big_graph = ['c1rg','c1rw','c1bw','c2rb','c2bw','c2wg','c3gr','c3gw','c3bb','c4rr','c4rb','c4wg'] #edges from these must be chosen for subgraphs and we
#all the edges inn the big graph
E = [(1,'r','g'),(1,'r','w'),(1,'b','w'),(2,'r','b'),(2,'b','w'),(2,'w','g'),(3,'g','r'),(3,'g','w'),(3,'b','b'),(4,'r','r'),(4,'r','b'),(4,'w','g')]
```

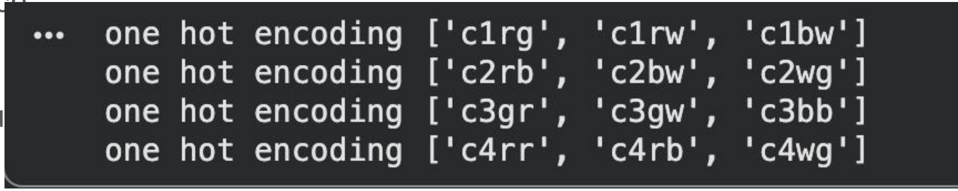
Classical Approach

- To solve the problem we needed to extract two subgraphs from the main graph (These subgraphs represent the Front-back and Left-Right side relationships of the solution)
- Embedded Rules :

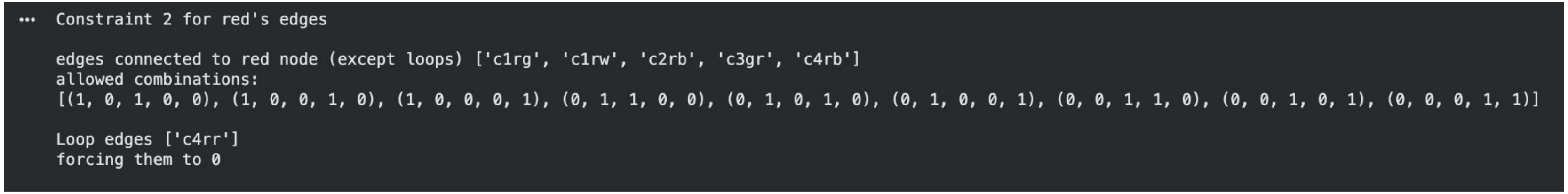
  Rule 1: Degree of each node is 2 (one for each side)

  Rule 2: All 4 cubes have an edge in each subgraph

- Our planned approach is to use QAOA to find the correct cu combinations that solve the puzzle.



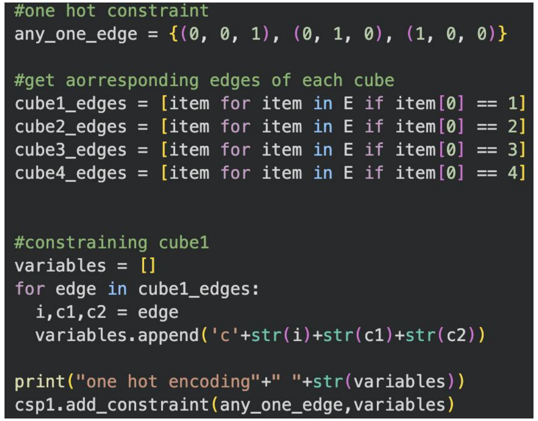
```
•••  one hot encoding ['c1rg', 'c1rw', 'c1bw']
     one hot encoding ['c2rb', 'c2bw', 'c2wg']
     one hot encoding ['c3gr', 'c3gw', 'c3bb']
     one hot encoding ['c4rr', 'c4rb', 'c4wg']
```



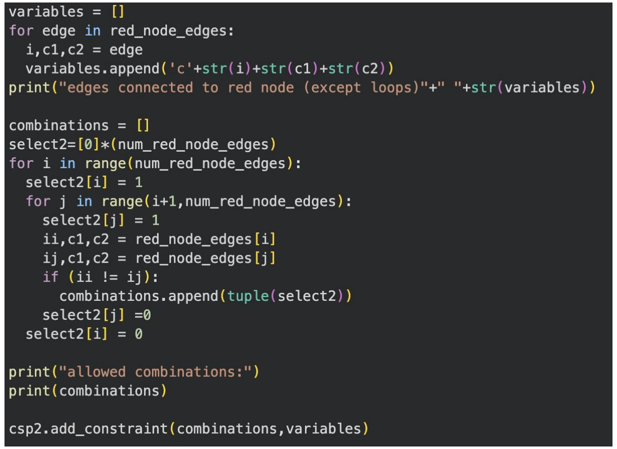
```
•••  Constraint 2 for red's edges

     edges connected to red node (except loops) ['c1rg', 'c1rw', 'c2rb', 'c3gr', 'c4rb']
     allowed combinations:
     [(1, 0, 1, 0, 0), (1, 0, 0, 1, 0), (1, 0, 0, 0, 1), (0, 1, 1, 0, 0), (0, 1, 0, 1, 0), (0, 1, 0, 0, 1), (0, 0, 1, 1, 0), (0, 0, 1, 0, 1), (0, 0, 0, 1, 1)]

     Loop edges ['c4rr']
     forcing them to 0
```

Classical Approach (cont.)

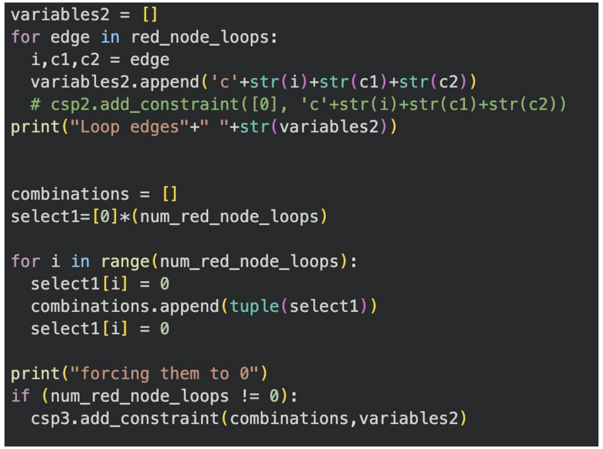
- D-Wave has the dwavebinarycsp library we can use to input our constraints problem in a classical form to translate into a QUBO.
- We turn the problem into a binary quadratic model, which was created by applying the constraints we fed the binary csp.



```
#one hot constraint
any_one_edge = {(0, 0, 1), (0, 1, 0), (1, 0, 0)}

#get aorresponding edges of each cube
cube1_edges = [item for item in E if item[0] == 1]
cube2_edges = [item for item in E if item[0] == 2]
cube3_edges = [item for item in E if item[0] == 3]
cube4_edges = [item for item in E if item[0] == 4]


#constraining cube1
variables = []
for edge in cube1_edges:
  i,c1,c2 = edge
  variables.append('c'+str(i)+str(c1)+str(c2))

print("one hot encoding"+" "+str(variables))
csp1.add_constraint(any_one_edge,variables)
```



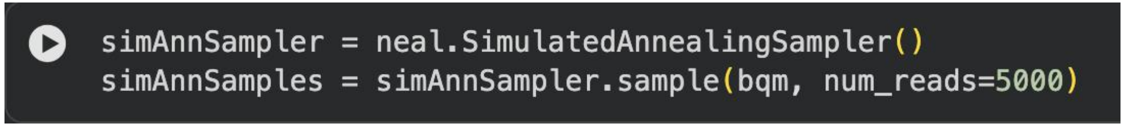
```
variables = []
for edge in red_node_edges:
  i,c1,c2 = edge
  variables.append('c'+str(i)+str(c1)+str(c2))
print("edges connected to red node (except loops)"+" "+str(variables))

combinations = []
select2=[0]*(num_red_node_edges)
for i in range(num_red_node_edges):
  select2[i] = 1
  for j in range(i+1,num_red_node_edges):
    select2[j] = 1
    ii,c1,c2 = red_node_edges[i]
    ij,c1,c2 = red_node_edges[j]
    if (ii != ij):
      combinations.append(tuple(select2))
    select2[j] =0
  select2[i] = 0

print("allowed combinations:")
print(combinations)

csp2.add_constraint(combinations,variables)
```



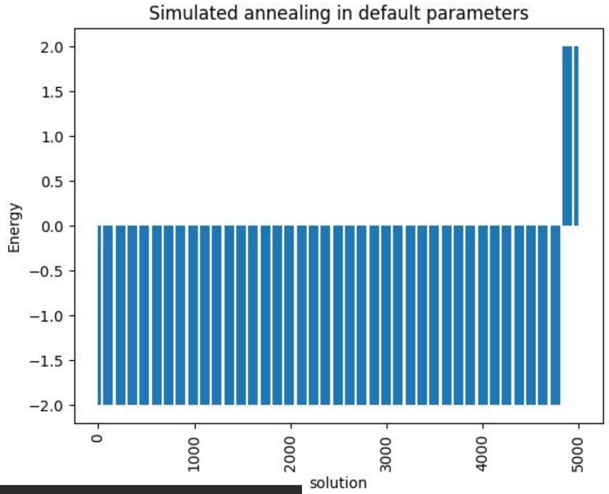
```
variables2 = []
for edge in red_node_loops:
  i,c1,c2 = edge
  variables2.append('c'+str(i)+str(c1)+str(c2))
  # csp2.add_constraint([0], 'c'+str(i)+str(c1)+str(c2))
print("Loop edges"+" "+str(variables2))


combinations = []
select1=[0]*(num_red_node_loops)

for i in range(num_red_node_loops):
  select1[i] = 0
  combinations.append(tuple(select1))
  select1[i] = 0

print("forcing them to 0")
if (num_red_node_loops != 0):
  csp3.add_constraint(combinations,variables2)
```

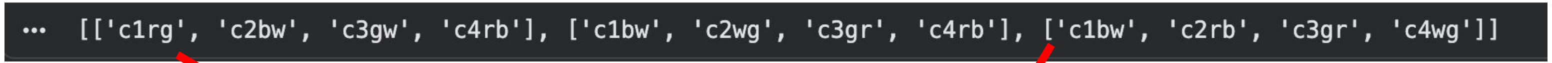
D-Wave Results

- After using simulated annealing we see the number of solutions possible.

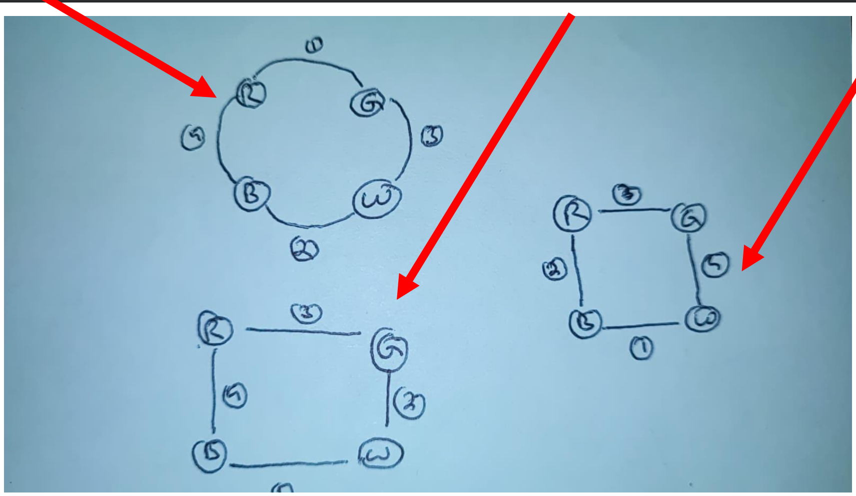


```
simAnnSampler = neal.SimulatedAnnealingSampler()
simAnnSamples = simAnnSampler.sample(bqm, num_reads=5000)
```



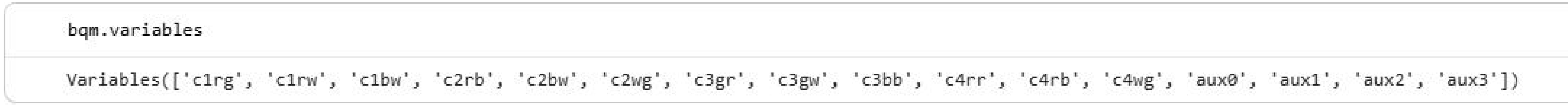


```
••• [['c1rg', 'c2bw', 'c3gw', 'c4rb'], ['c1bw', 'c2wg', 'c3gr', 'c4rb'], ['c1bw', 'c2rb', 'c3gr', 'c4wg']]
```



Formulation - Extracted QUBO equation

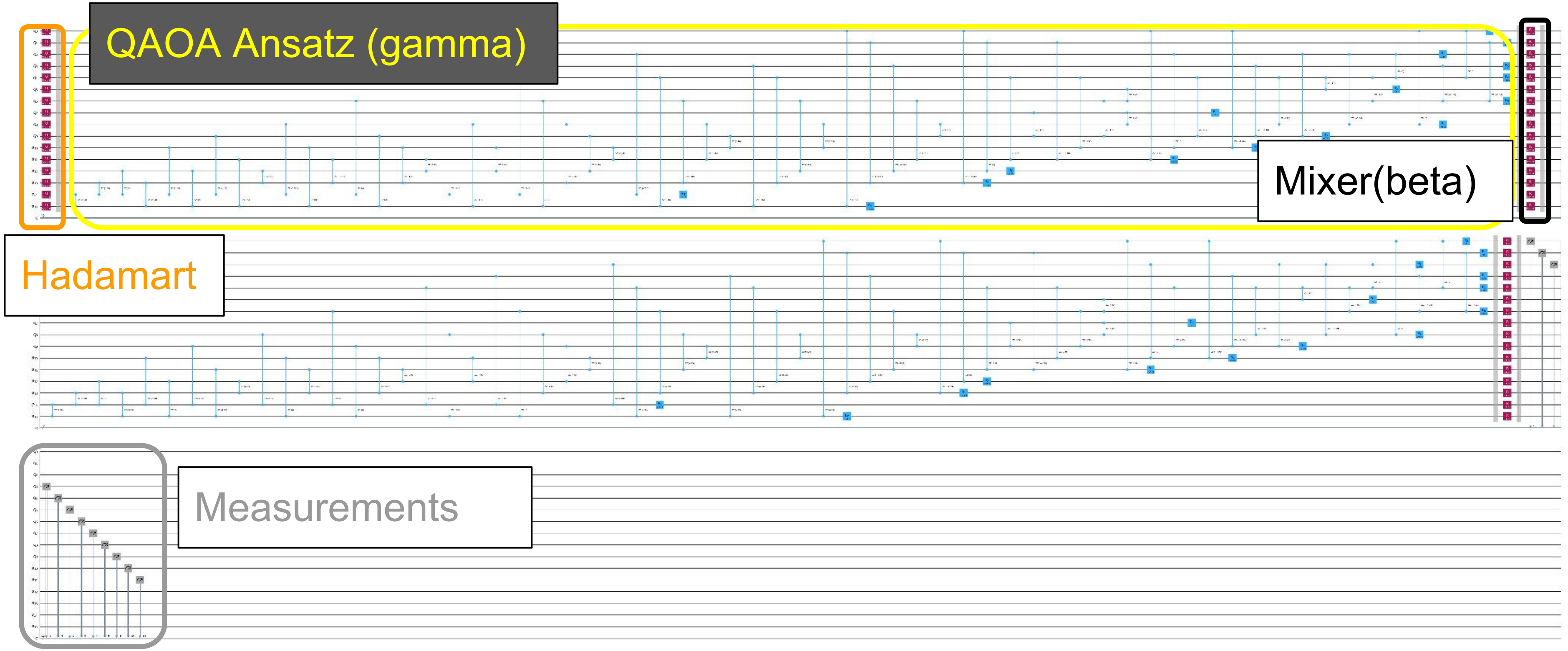
- We get the QUBO from bqm:
  - 8.0*xc1rw*xc1rg + 4.0*xc1bw*xc1rg + 8.0*xc1bw*xc1rw + 4.0*xc2rb*xc1bw + 4.0*xc2bw*xc1rw + 2.0*xc2bw*xc1bw + 8.0*xc2bw*xc2rb + 4.0*xc2wg*xc1rg + 4.0*xc2wg*xc1bw + 4.0*xc2wg*xc2rb + 8.0*xc2wg*xc2bw + 4.0*xc3gr*xc1rg + 4.0*xc3gr*xc2rb + 4.0*xc3gr*xc2wg + 2.0*xc3gw*xc1rg + 4.0*xc3gw*xc1rw + 4.0*xc3gw*xc1bw + 4.0*xc3gw*xc2bw + 8.0*xc3gw*xc2wg + 8.0*xc3gw*xc3gr + 4.0*xc3bb*xc3gr + 4.0*xc3bb*xc3gw + 4.0*xc4rb*xc1bw + 8.0*xc4rb*xc2rb + 4.0*xc4rb*xc2bw + 4.0*xc4rb*xc3gr + 4.0*xc4rb*xc4rr + 4.0*xc4wg*xc1rg + 4.0*xc4wg*xc1rw + 4.0*xc4wg*xc1bw + 4.0*xc4wg*xc2bw + 8.0*xc4wg*xc2wg + 4.0*xc4wg*xc3gr + 8.0*xc4wg*xc3gw + 4.0*xc4wg*xc4rr + 4.0*xc4wg*xc4rb + 4.0*xaux0*xc1rg + 4.0*xaux0*xc1rw + -4.0*xaux0*xc2rb + -4.0*xaux0*xc3gr + -4.0*xaux0*xc4rb + 2.0*xaux1*xc1rg + 2.0*xaux1*xc3gr + -4.0*xaux1*xc3gw + 2.0*xaux2*xc1bw + -4.0*xaux2*xc2bw + -4.0*xaux2*xc4rb + 4.0*xaux3*xc1rw + -4.0*xaux3*xc1bw + 4.0*xaux3*xc2bw + -4.0*xaux3*xc2wg + -10.0*xc1rg + -10.0*xc1rw + -10.0*xc1bw + -10.0*xc2rb + -12.0*xc2bw + -10.0*xc2wg + -10.0*xc3gr + -12.0*xc3gw + -10.0*xc4rb + -14.0*xc4wg + 4.0*xaux0 + 2.0*xaux1 + 6.0*xaux2 + 34.0
- Given our QUBO, now we can use it as our cost Hamiltonian by converting it to a SparsePauliOp with the specific weights we were given from the binary csp process.



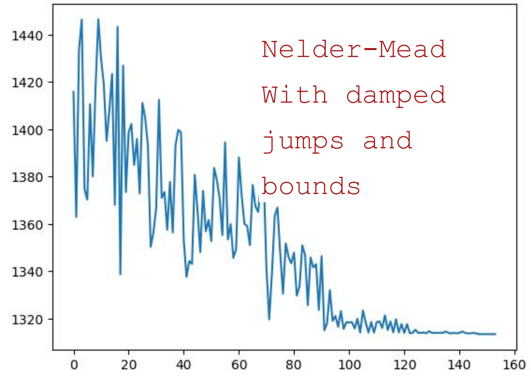
```
bqm.variables
```

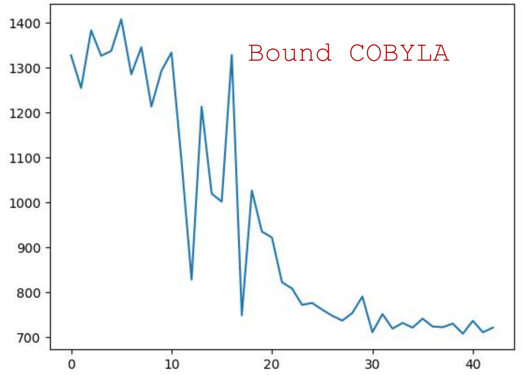
```
Variables(['c1rg', 'c1rw', 'c1bw', 'c2rb', 'c2bw', 'c2wg', 'c3gr', 'c3gw', 'c3bb', 'c4rr', 'c4rb', 'c4wg', 'aux0', 'aux1', 'aux2', 'aux3'])
```

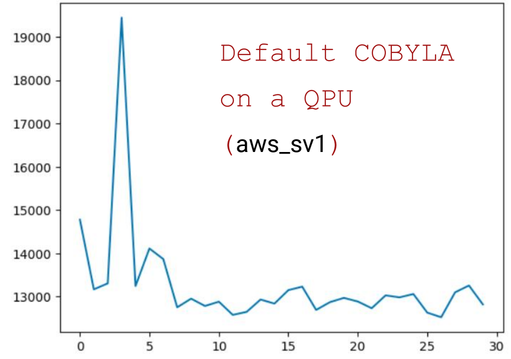
QAOA Quantum Circuit

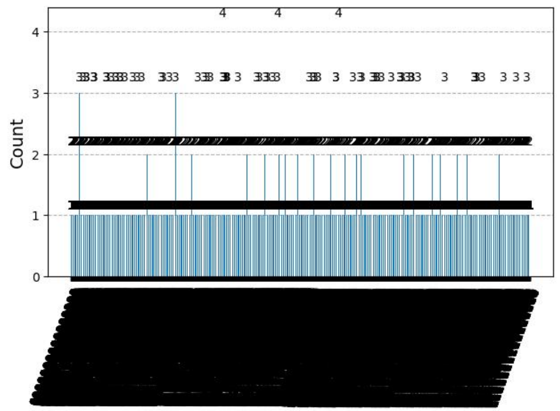


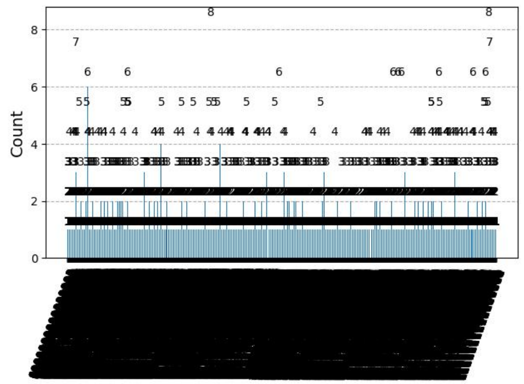
Results

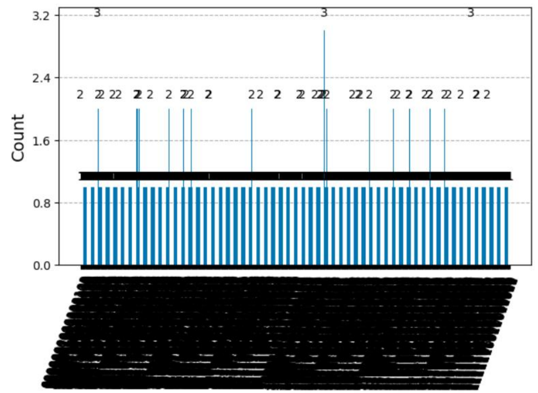

Nelder-Mead
With damped
jumps and
bounds


Bound COBYLA


Default COBYLA
on a QPU
(aws_sv1)


Count


Count

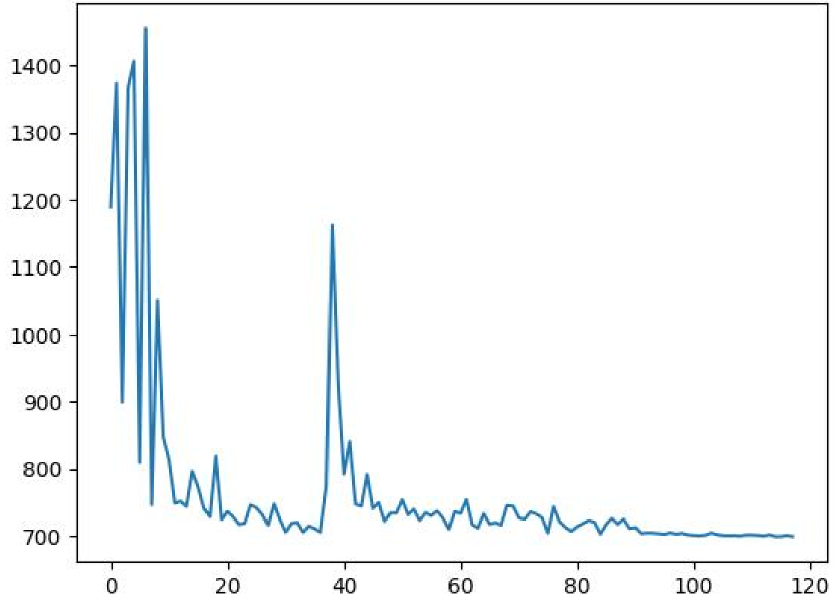

Count

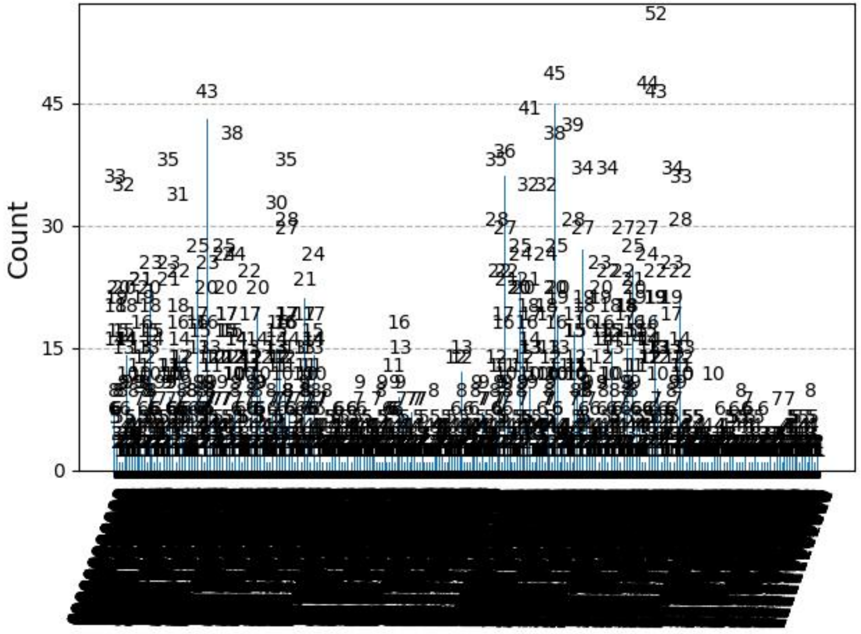
None of them are perfect

- Nelder - very localized
- COBYLA - spread out
- Analysing more probable wrong outputs suggested that some penalties were just not enough and some were outweighing others.
- Example: if the system accepted one rule-violation to persist the other violations could be easily taken care of
- Also P was needed to be kept small and bounds needed to be made to not let search space explode

Best Approach so far - Hybrid + Tailored Energy-Cost function

```
# COBYLA goes haywire and tries to find everything everywhere so get a ballpark of solutions there the use NELDER to hone in
```





Modified Cost Function

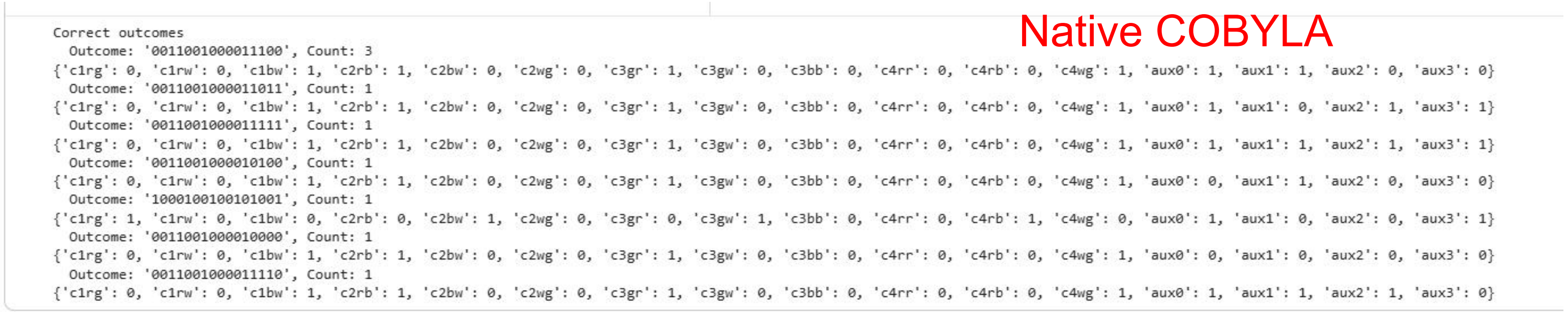
```
def compute_expectation(counts):
      total = 0
      for bitstring, count in counts.items():
        obj = 0

        sample_values_as_ints = [int(bit) for bit in bitstring]
        sample = dict(zip(bqm.variables, sample_values_as_ints))
        total += bqm.energy(sample)

      all_total_E.append(total)
      return total
```

Before

```
def compute_expectation(counts):
      total = 0
      total_shots = sum(counts.values())
      for bitstring, count in counts.items():
        obj = 0
        sample_values_as_ints = [int(bit) for bit in bitstring]
        sample = dict(zip(bqm.variables, sample_values_as_ints))
        sample['aux0'] = 1
        sample['aux1'] = 0
        sample['aux2'] =1
        sample['aux3'] = 1
        for i, constraint in enumerate(csp1.constraints):
          passed = constraint.check(sample)
          if (not passed):
            total += 30*bqm_rule1.energy(sample)

        for i, constraint in enumerate(csp2.constraints)
          passed = constraint.check(sample)
          if (not passed):
            total += 13*bqm_rule2.energy(sample)

        for i, constraint in enumerate(csp3.constraints):
          passed = constraint.check(sample)
          if (not passed):
            total += 25*bqm_rule3.energy(sample)

      all_total_E.append(total/total_shots)
      return total/total_shots
```

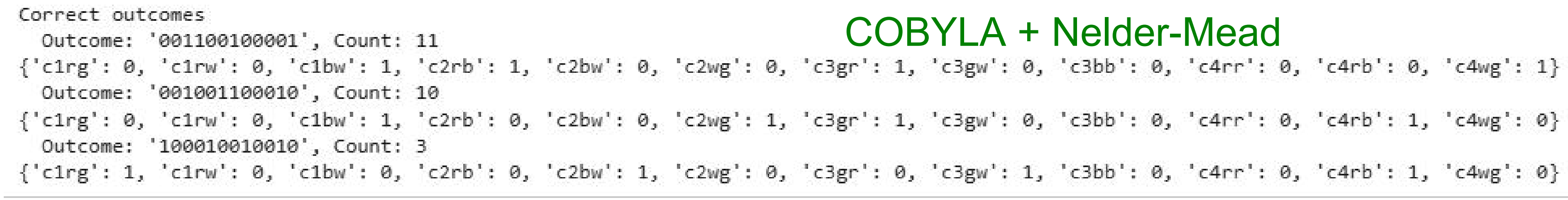
After

Benefit - Correct answers were little more probable now



Native COBYLA

```
  Correct outcomes
    Outcome: '0011001000011100', Count: 3
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 1, 'aux1': 1, 'aux2': 0, 'aux3': 0}
    Outcome: '0011001000011011', Count: 1
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 1, 'aux1': 0, 'aux2': 1, 'aux3': 1}
    Outcome: '0011001000011111', Count: 1
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 1, 'aux1': 1, 'aux2': 1, 'aux3': 1}
    Outcome: '0011001000010100', Count: 1
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 0, 'aux1': 1, 'aux2': 0, 'aux3': 0}
    Outcome: '1000100100101001', Count: 1
  {'c1rg': 1, 'c1rw': 0, 'c1bw': 0, 'c2rb': 0, 'c2bw': 1, 'c2wg': 0, 'c3gr': 0, 'c3gw': 1, 'c3bb': 0, 'c4rr': 0, 'c4rb': 1, 'c4wg': 0, 'aux0': 1, 'aux1': 0, 'aux2': 0, 'aux3': 1}
    Outcome: '0011001000010000', Count: 1
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 0, 'aux1': 0, 'aux2': 0, 'aux3': 0}
    Outcome: '0011001000011110', Count: 1
  {'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1, 'aux0': 1, 'aux1': 1, 'aux2': 1, 'aux3': 0}
```



COBYLA + Nelder-Mead

```
Correct outcomes
  Outcome: '001100100001', Count: 11
{'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 1, 'c2bw': 0, 'c2wg': 0, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 0, 'c4wg': 1}
  Outcome: '001001100010', Count: 10
{'c1rg': 0, 'c1rw': 0, 'c1bw': 1, 'c2rb': 0, 'c2bw': 0, 'c2wg': 1, 'c3gr': 1, 'c3gw': 0, 'c3bb': 0, 'c4rr': 0, 'c4rb': 1, 'c4wg': 0}
  Outcome: '100010010010', Count: 3
{'c1rg': 1, 'c1rw': 0, 'c1bw': 0, 'c2rb': 0, 'c2bw': 1, 'c2wg': 0, 'c3gr': 0, 'c3gw': 1, 'c3bb': 0, 'c4rr': 0, 'c4rb': 1, 'c4wg': 0}
```

Both 10,000 shots

Our Conclusions

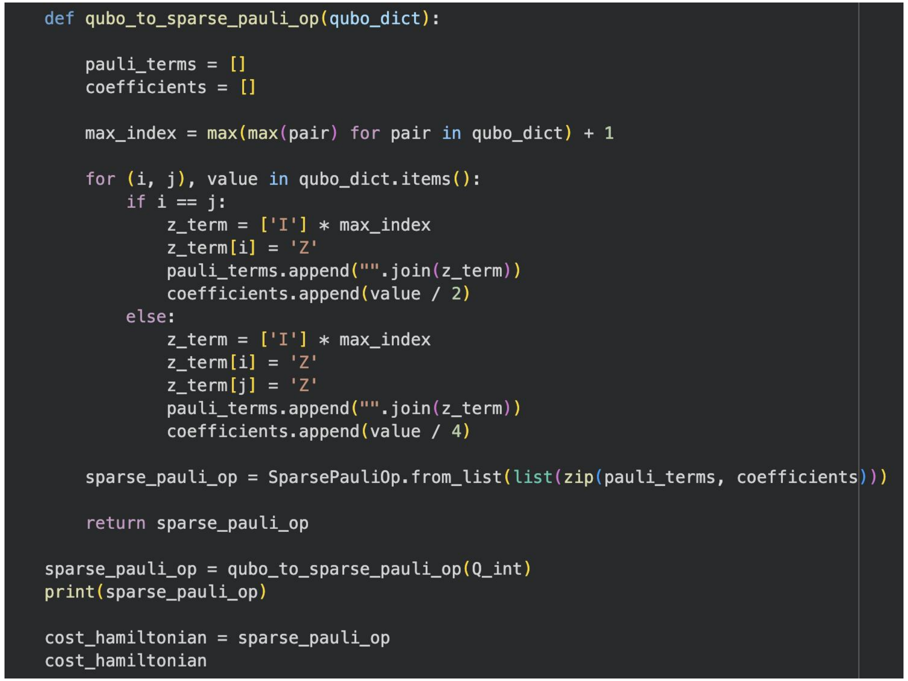
- The “Quantum Advantage” with QAOA can be effectively utilised only if the classical minimization approach is chosen properly based on the problem.
- To limit the search space even moreSome sources suggest that Mixer architecture can be tailored to the problem. (This is probably a good starting point to continue our work)
- Weights and Penalties need to be tailored to the problem as well to lead the system in the correct direction.

References

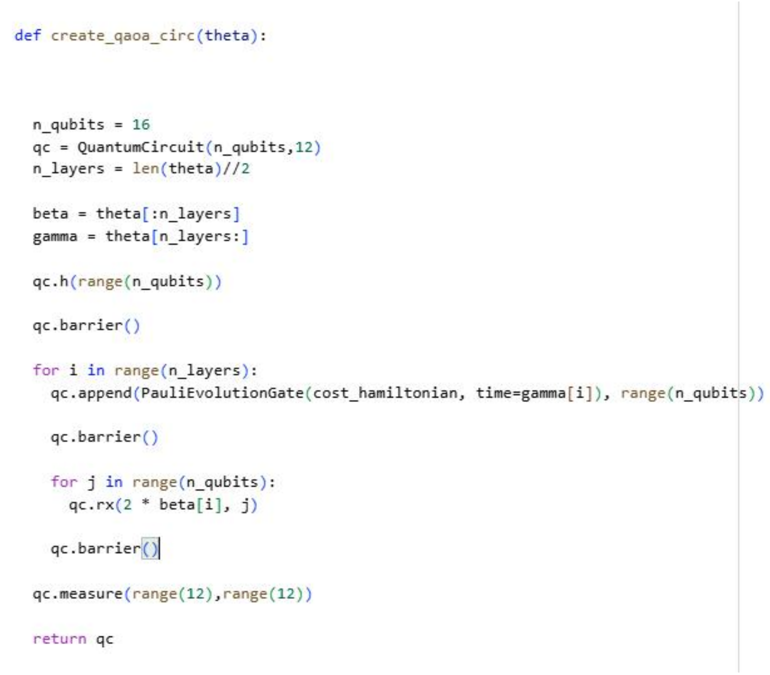
- https://www.math.ksu.edu/outreach/skday/2023/slides/FourCubesSlides.pdf
- https://secquoia.github.io/QUBOBook/QUBO%20and%20Ising%20Models.html

BACKUP SLIDES

Solving Using Universal Quantum Computer



```
def qubo_to_sparse_pauli_op(qubo_dict):

    pauli_terms = []
    coefficients = []

    max_index = max(max(pair) for pair in qubo_dict) + 1

    for (i, j), value in qubo_dict.items():
        if i == j:
            z_term = ['I'] * max_index
            z_term[i] = 'Z'
            pauli_terms.append("".join(z_term))
            coefficients.append(value / 2)
        else:
            z_term = ['I'] * max_index
            z_term[i] = 'Z'
            z_term[j] = 'Z'
            pauli_terms.append("".join(z_term))
            coefficients.append(value / 4)

    sparse_pauli_op = SparsePauliOp.from_list(list(zip(pauli_terms, coefficients)))

    return sparse_pauli_op

sparse_pauli_op = qubo_to_sparse_pauli_op(Q_int)
print(sparse_pauli_op)

cost_hamiltonian = sparse_pauli_op
cost_hamiltonian
```



```
def create_qaoa_circ(theta):



 n_qubits = 16
 qc = QuantumCircuit(n_qubits,12)
 n_layers = len(theta)//2

 beta = theta[:n_layers]
 gamma = theta[n_layers:]

 qc.h(range(n_qubits))

 qc.barrier()

 for i in range(n_layers):
  qc.append(PauliEvolutionGate(cost_hamiltonian, time=gamma[i]), range(n_qubits))

  qc.barrier()

  for j in range(n_qubits):
   qc.rx(2 * beta[i], j)

  qc.barrier()

 qc.measure(range(12),range(12))

 return qc
```