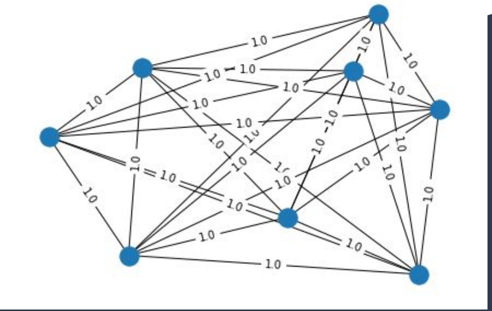
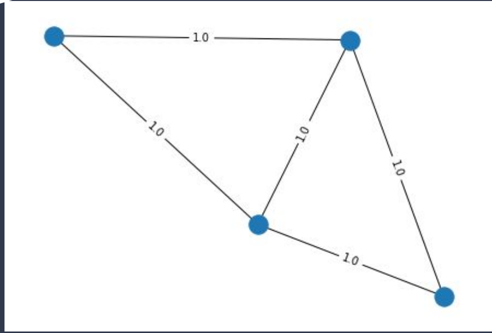
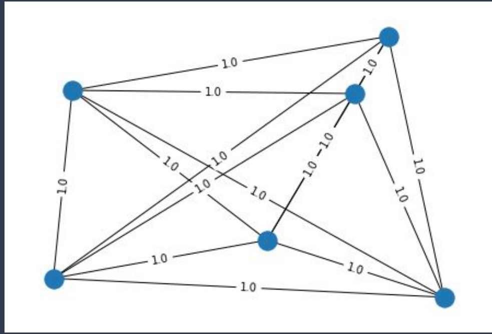
Implementing Quantum Approximation Optimization Algorithm (QAOA) with the Max Cut Problem

Anna Hsu

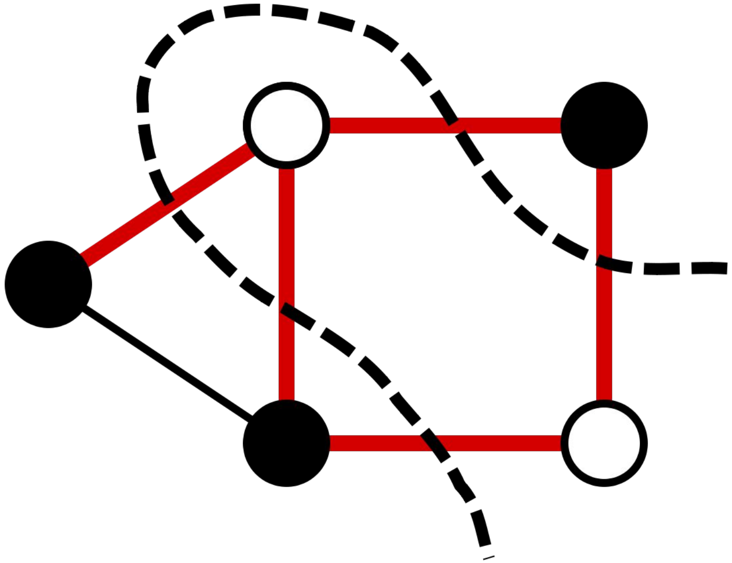






The Max Cut Problem

→ Given an undirected graph with n nodes and weighted edges connecting the nodes, find the cut that divides the nodes into two groups and maximizes the number of edges across groups



Example of the max cut of a graph
https://en.wikipedia.org/wiki/Maximum_cut

The Max Cut Problem

Can be used to model:

- Circuits with different flow levels
- Marketing model, using weighted edges to represent the potential influence of an interaction between two people

A similar QAOA approach can solve problems such as the Traveling Salesman, Graph Coloring, and Knapsack problems

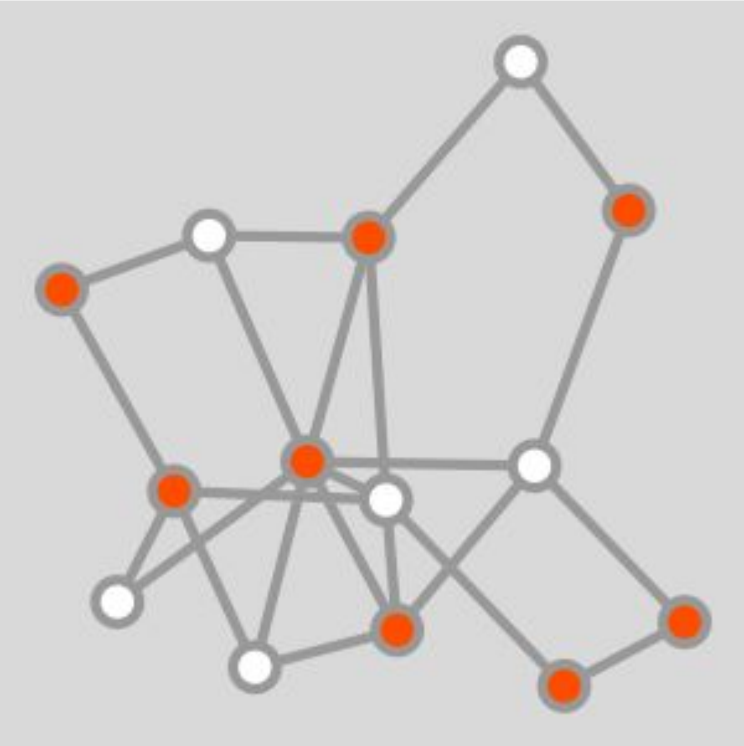


Image:https://www.wolfram.com/language/12/core-graphs-and-networks/solve-max-cut-problem.html?product=mathematica

Pauli Matrices

$$\sigma_1 = \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\sigma_2 = \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\sigma_3 = \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- Called “gates”
- These are operations that can be applied to a qubit in different combinations

In Qiskit:

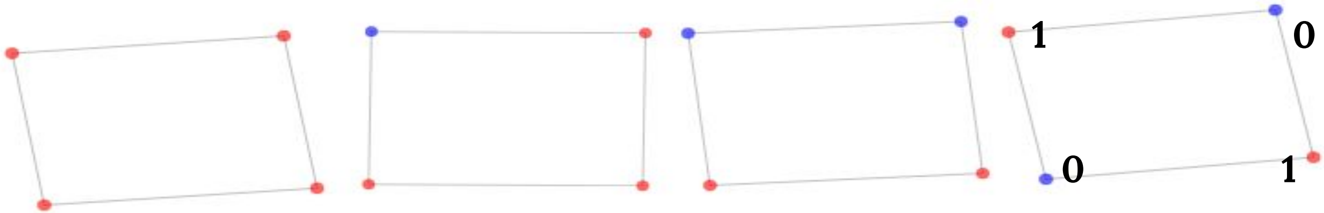
Image:https://www.wolfram.com/language/12/core-graphs-and-networks/solve-max-cut-problem.html?product=mathematica

Classical Solution

```
In [36]: def objective_value(x, w):
             X = np.outer(x, (1 - x))
             w_01 = np.where(w != 0, 1, 0)
             return np.sum(w_01 * X)

         def brute_force():
             # use the brute-force way to generate the oracle
             def bitfield(n, L):
                 result = np.binary_repr(n, L)
                 return [int(digit) for digit in result]  # [2:] to chop off the "0b" part

             L = num_nodes
             max = 2**L # max number of combinations to put L nodes into two groups
             maximum_v = np.inf
             for i in range(max):
                 cur = bitfield(i, L) #binary representation of i

                 how_many_nonzero = np.count_nonzero(cur)
                 if how_many_nonzero * 2 != L:  # checks if # of 0 and 1s is balanced, continues if not
                     continue

                 cur_v = objective_value(np.array(cur), w)
                 if cur_v < maximum_v: # replaces maximum with new maximum to find true max number of edges
                     maximum_v = cur_v
             return maximum_v

         sol = brute_force()
         print(f'Objective value computed by the brute-force method is {sol}')

         Objective value computed by the brute-force method is 3
```

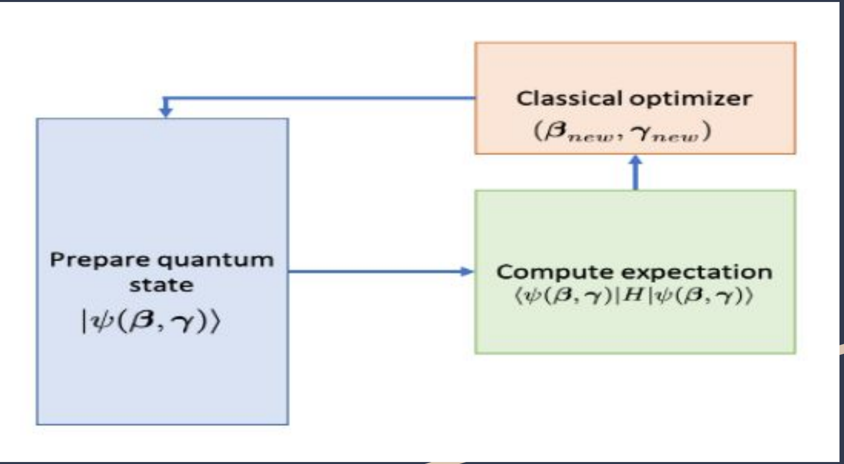
- Brute force method stores each combination of red-blue nodes as a bitstring and tests 2^L combinations
- Time to solve using brute force method increases exponentially as # of nodes increases

0 = blue
1 = red



Source: https://qiskit.org/textbook/ch-applications/qaoa.html

Quantum Solution



Solving with QAOA depends on two parameters, β and γ

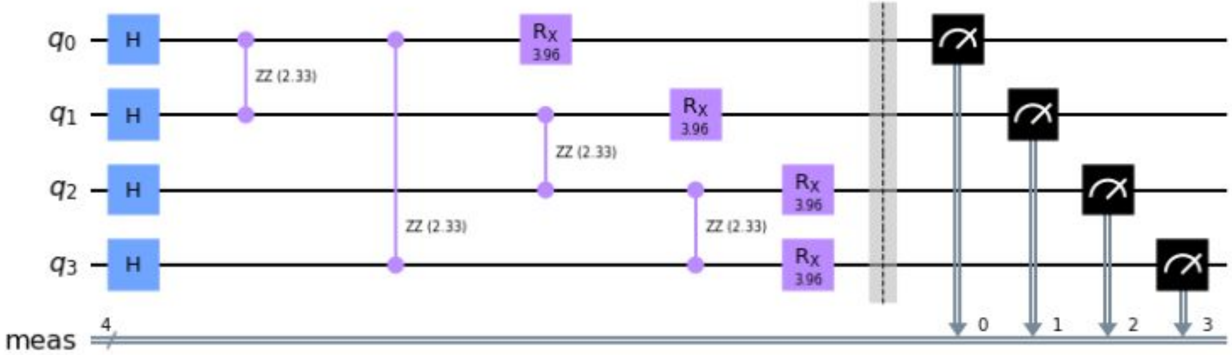
1. Prepare equal superposition state
2. Apply QAOA circuit of alternating cost and mixer layers
3. Measurement in computational basis → bitstring samples
4. Evaluate cut values
5. Optimize β and γ using classical optimizer, repeat with new parameters

Image: https://qiskit.org/textbook/ch-applications/qaoa.html#The-QAOA-circuit

The QAOA Circuit: An Overview

```
In [22]: qc_res.draw('mpl')
```

Out[22]:



Prepare initial state

Cost & mixer layers

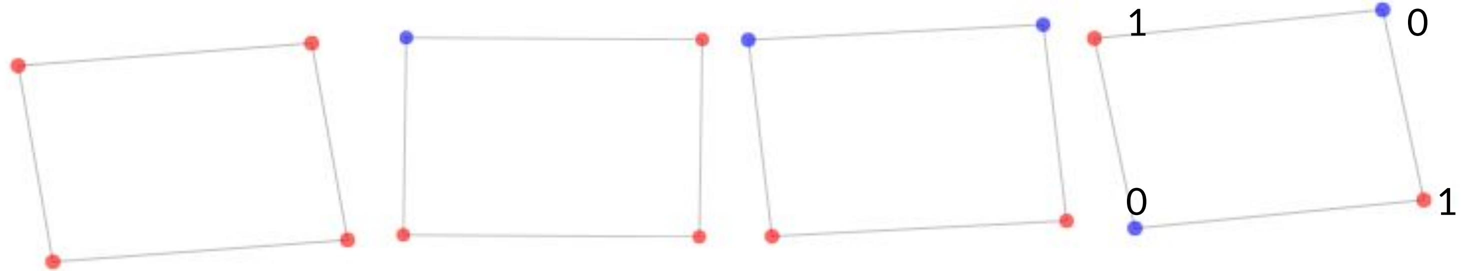
Measurement

Quantum Solution

Construct the Hamiltonian for the problem

- Assigns edge weight of 1 when i =1 , j = 0

$$C(\mathbf{x}) = \sum_{i,j=1}^{n} w_{ij} x_i (1 - x_j)$$



Source: https://qiskit.org/textbook/ch-applications/qaoa.html

The Cost Layer (γ)

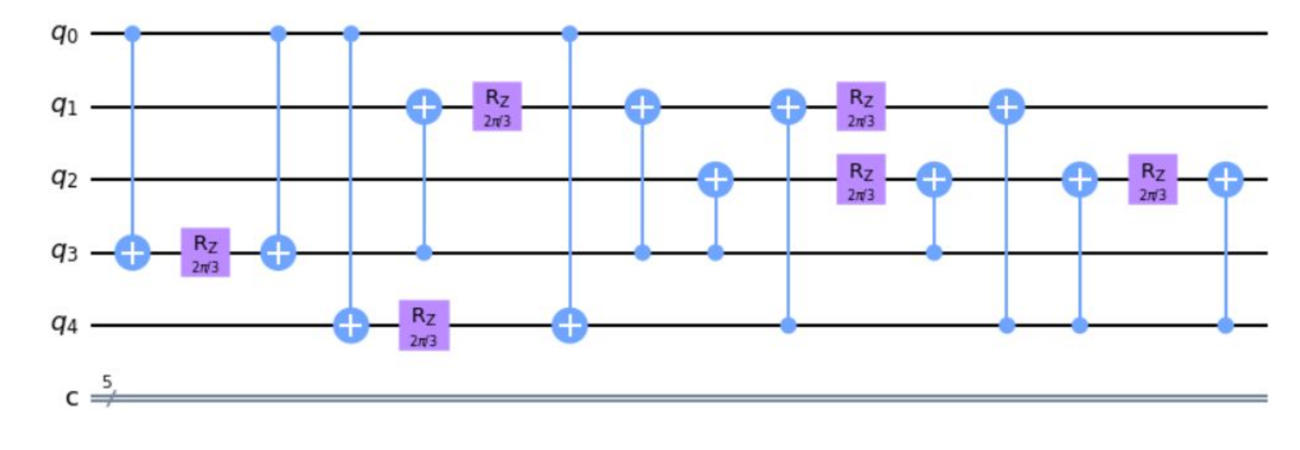
The cost layer is constructed with the exponentiation of the cost Hamiltonian, which decomposes to a combination of CNOT gates and R_{ZZ} rotation gates

Cost Hamiltonian:

$$H_C = \sum_{i,j=1}^{n} \frac{1}{4} Q_{ij} Z_i Z_j - \sum_{i=1}^{n} \frac{1}{2} \left(c_i + \sum_{j=1}^{n} Q_{ij} \right) Z_i$$

The Cost Layer (γ)

The cost layer is constructed with the exponentiation of the cost Hamiltonian, which decomposes to a combination of CNOT gates and R_{ZZ} rotation gates



The Mixer Layer (β)

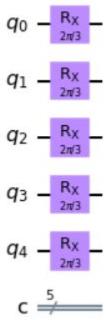
- The mixer layer "mixes" the quantum state so that the cost layer can be applied again to test more quantum states, parameterized by β.
- Constructed by the exponentiation of the mixer Hamiltonian
- X rotation gate by 2β

$$H_M = \sum_{i=1}^{n} X_i$$

In [8]:
```
def append_x_term(qc, q1, beta):
    qc.rx(2*beta, q1)

def get_mixer_operator_circuit(G, beta):
    N = G.number_of_nodes()
    qc = QuantumCircuit(N,N)
    for n in G.nodes():
        append_x_term(qc, n, beta)
    return qc
```

In [10]:
```
qc = get_mixer_operator_circuit(G, np.pi/3)
qc.draw('mpl')
```

Out[10]:

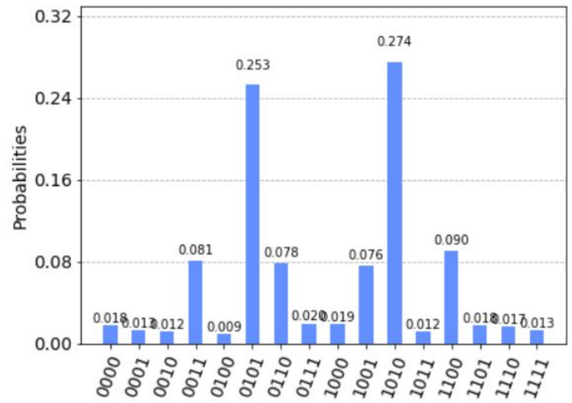


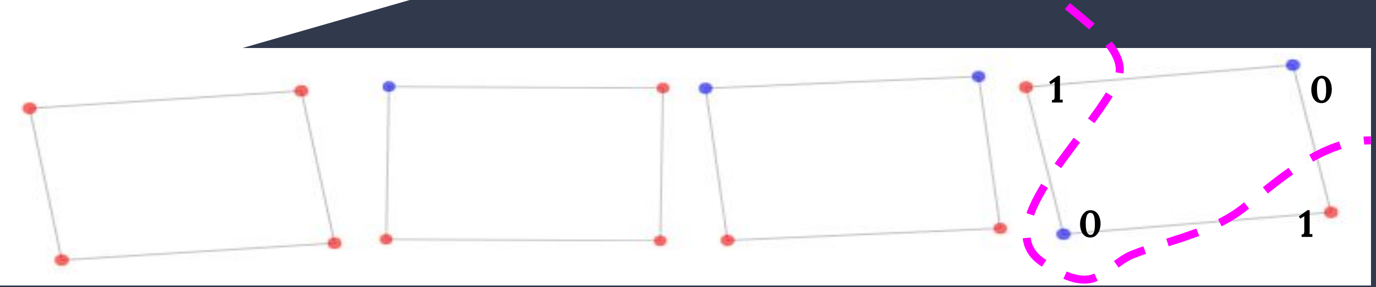
Results

In [21]:

```python
from qiskit.visualization import plot_histogram

backend = Aer.get_backend('aer_simulator')
backend.shots = 512

qc_res = create_qaoa_circ(G, res.x)

counts = backend.run(qc_res, seed_simulator=10).result().get_counts()

plot_histogram(counts)
```

Out[21]:

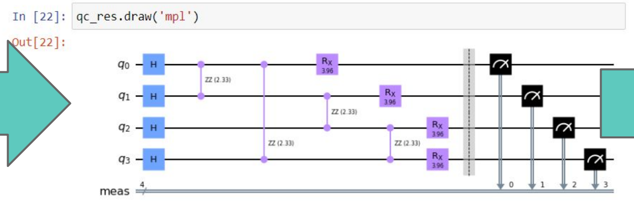




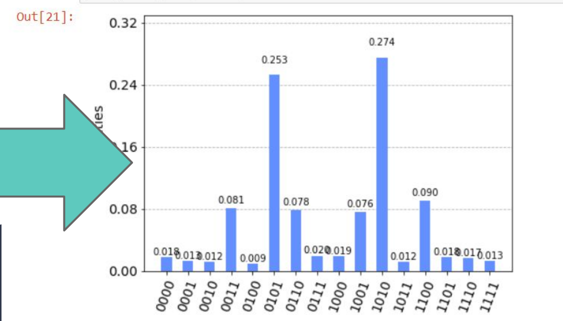
Recap

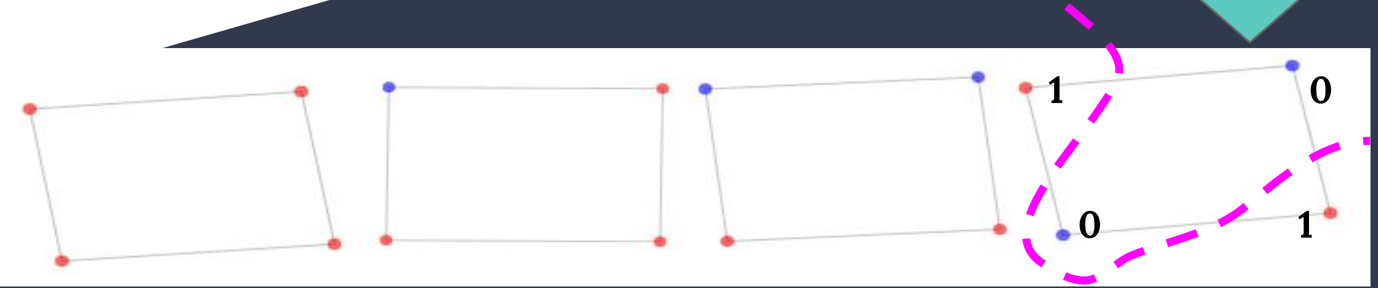
Graph

QAOA circuit



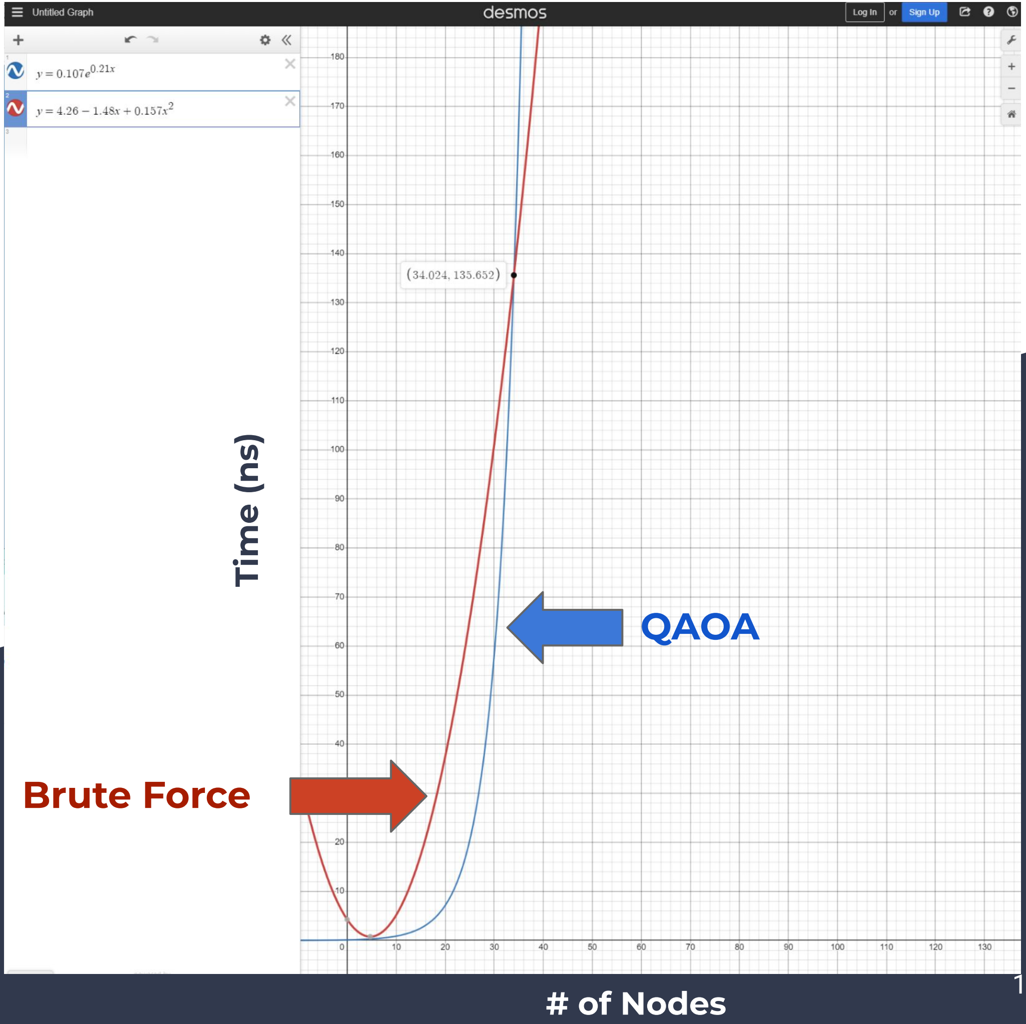
Results





Results

- Classical computing time modeled by exponential equation, QAOA time modeled by polynomial equation
- QAOA operates in polynomial time and is faster than classical approach after threshold of 34+ node graph
-



Sources

Qiskit Summer School 2021 Lab 2
https://learn.qiskit.org/summer-school/2021/lab2-variational-algorithms

Solving combinatorial optimization problems using QAOA
https://qiskit.org/textbook/ch-applications/qaoa.html

Quantum Approximation Optimization Algorithm
https://qiskit.org/documentation/locale/de_DE/tutorials/algorithms/05_qaoa.html