QAOA-GPT: Efficient Generation of Adaptive and Regular Quantum Approximate Optimization Algorithm Circuits

Algorithms of Quantum Information 2025

ECE TUC

Athanasios Karakos 2021030011

Table of Contents

QAOA

(Quantum Approximate Optimization Algorithm)

Framework

The Quantum Approximate Optimization Algorithm is a hybrid quantum-classical algorithm designed to solve QUBO problems. The set of parameters to approximate the state |ψ∗⟩ is γ = (γ1, γ2, ...) and β = (β1, β2, ...) and the corresponding ansatz is U(β, γ) is inspired from the quantum annealing time- evolution Hamiltonian along the trotterizetion theorem.

The variational parameters γ and β are tuned to optimize the final measurement outcomes, driving the system towards the optimal solution.

The quantum state is:

$$|\psi(\beta,\gamma)\rangle = \prod_{k=1}^{P} U_M(\beta_k) U_C(\gamma_k) |\psi_0\rangle$$

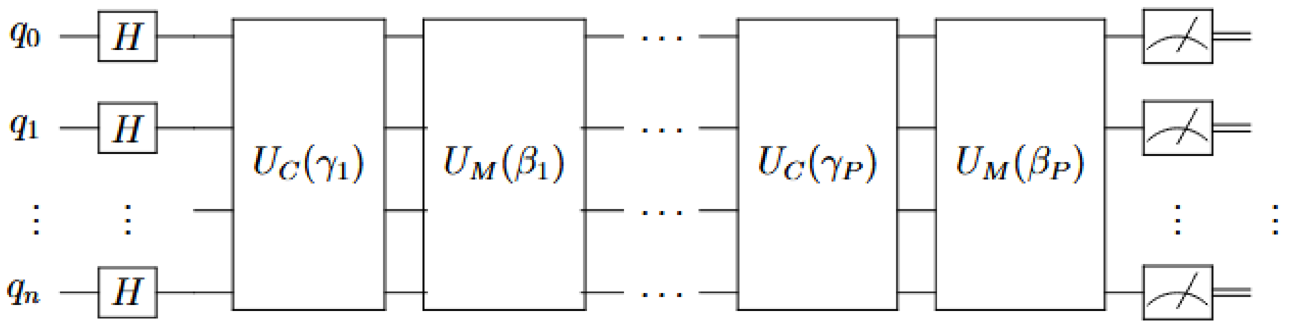
Where P is fixed and equal to how many times the $U_M U_C$ is applied (layers)

The operator $U_M(\beta_k) = e^{-i\beta_k H_M}$, H_M is the mixing Hamiltonian equal to $X_0 X_1 \dots X_{qubits}$

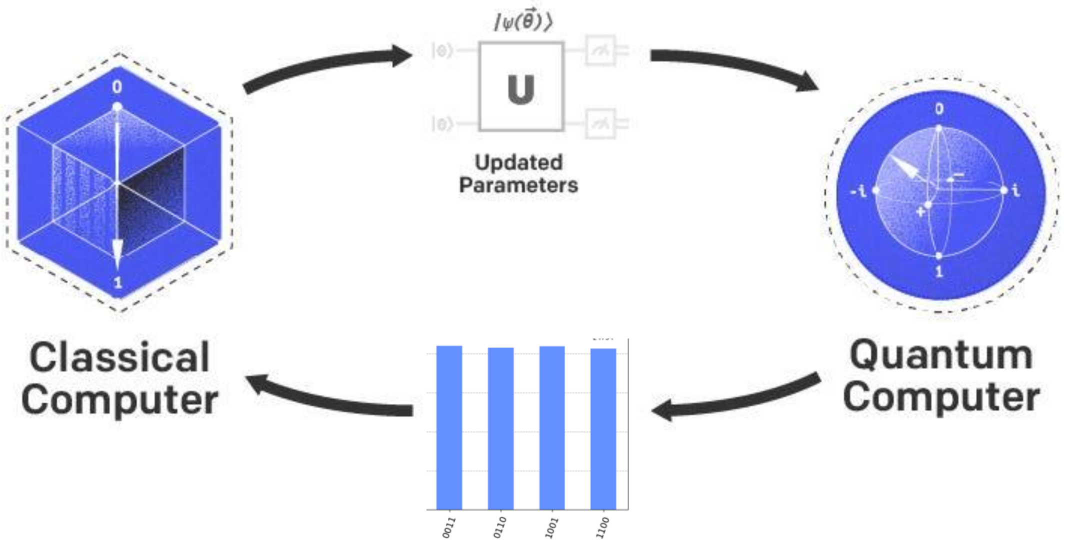
The operator $U_C(\gamma_k) = e^{-i\gamma_k H_C}$, H_C is the cost Hamiltonian also known as the problem Hamiltonian

The initial state is prepared with a set of Hadamard gates $|\psi_0\rangle = |+\rangle$

A Multi Layer (P layers) QAOA ansatz with alternating $U_M(\beta)$ and $U_C(\gamma)$ operators is shown below:



The circuit is measured, the measurement counts are passed to a classical computer and the parameters γ, β are optimized and passed to the quantum circuit again.



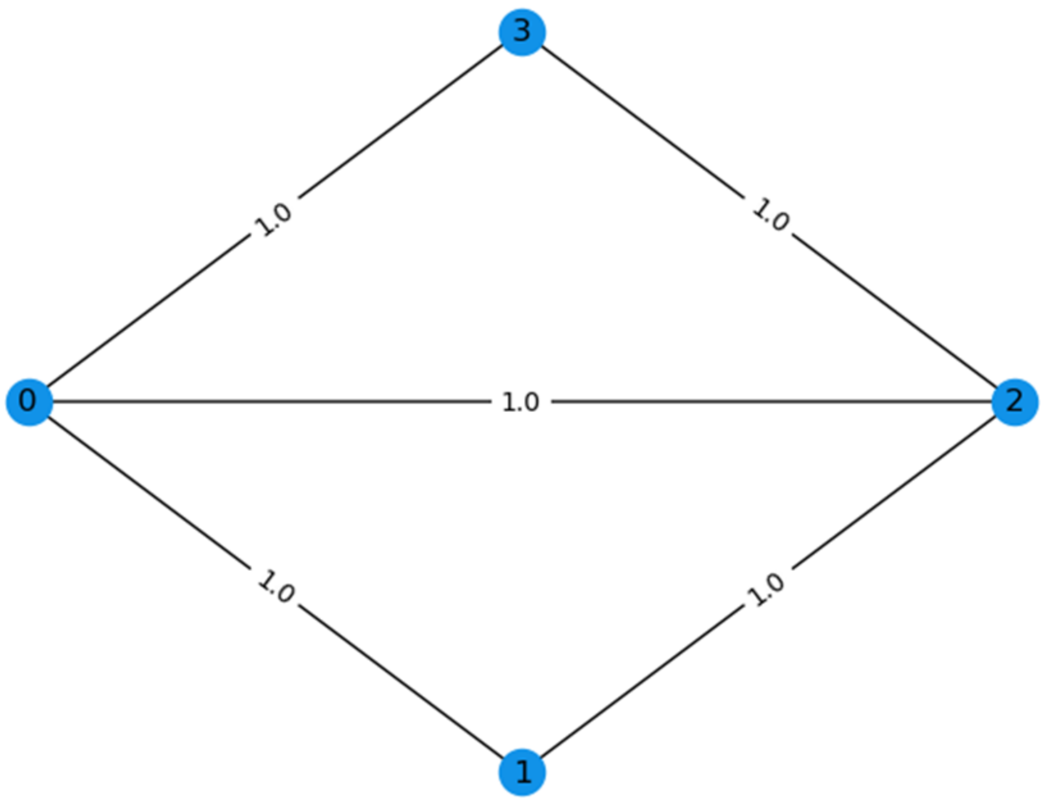
Max-Cut

Objective Function

The maximum cut (Max-Cut) problem is a combinatorial optimization problem that involves dividing the vertices of a graph into two disjoint sets such that the number of edges between the two sets is maximized. More formally, given an undirected graph **G=(V,E)** where **V** is the set of vertices and **E** is the set of edges, the Max-Cut problem asks to partition the vertices into two disjoint subsets, **S** and **T**, such that the number of edges with one endpoint in **S** and the other in **T** is maximized.

We can apply Max-Cut to solve a various problems including: clustering, network design, phase transitions, etc.

We'll start by creating a problem graph:



This problem can be expressed as a binary optimization problem.

For each node $0 \leq i \ < n$, where n is the number of nodes of the graph (in this case n = 4), we will consider the binary variable x_i. This variable will have the value 1 if node *i* is one of the groups that we'll label 1 and 0 if it's in the other group, that we'll label as 0. We will also denote as w_{ij} (element (i,j) of the adjacency matrix *w*) the weight of the edge that goes from node *i* to node *j*. Because the graph is undirected, $w_{ij} = w_{ji}$. Then we can formulate our problem as maximizing the following cost function:

$$C(x) = \sum_{i,j=0}^{n} w_{ij} x_i \left(1 - x_j\right)$$

$$= \sum_{i,j=0}^{n} w_{ij} x_i - \sum_{i,j=0}^{n} w_{ij} x_i x_j =$$

$$= \sum_{i,j=0}^{n} w_{ij} x_i - \sum_{i=0}^{n} \sum_{j=0}^{i} 2\, w_{ij} x_i x_j$$

To solve this problem with a quantum computer, we are going to express the cost function as the expected value of an observable. However, the observables that Qiskit admits natively consist of Pauli operators, that have eigenvalues 1 and −1 instead of 0 and 1. That's why we are going to make the following change of variable:

Where $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$. We can use the adjacency matrix *w* to comfortably access the weights of all the edge. This will be used to obtain our cost function:

$$z_i = 1 - 2x_i \ \rightarrow x_i = \frac{1 - z_i}{2}$$

This implies that:

$$x_i = 0 \ \rightarrow z_i = 1$$

$$x_i = 1 \rightarrow z_i = -1$$

So the new cost function we want to maximize is:

$$C(z) = \sum_{i,j=0}^{n} w_{ij}(\frac{1-z_i}{2})(1-\frac{1-z_j}{2})$$

$$= offset - \sum_{i=0}^{n}\sum_{j=0}^{i}\frac{w_{ij}}{2}z_i z_j$$

Instead of maximizing the Cost function, we are going to minimize the following:

$$-C(z) = \sum_{i=0}^{n}\sum_{j=0}^{i}\frac{w_{ij}}{2}z_i z_j + offset$$

Now that the cost function is ready, we can make the analogy with the Pauli-Z operator:

$$z_i = Z_i = \overbrace{I}^{n-1} \otimes \ldots \otimes \overbrace{Z}^{i} \otimes \ldots \otimes \overbrace{I}^{0}$$

It is equivalent to applying the Z gate on the i-th qubit

$$Z(a\,|0\rangle + b\,|1\rangle) = a\,|0\rangle - b\,|1\rangle$$

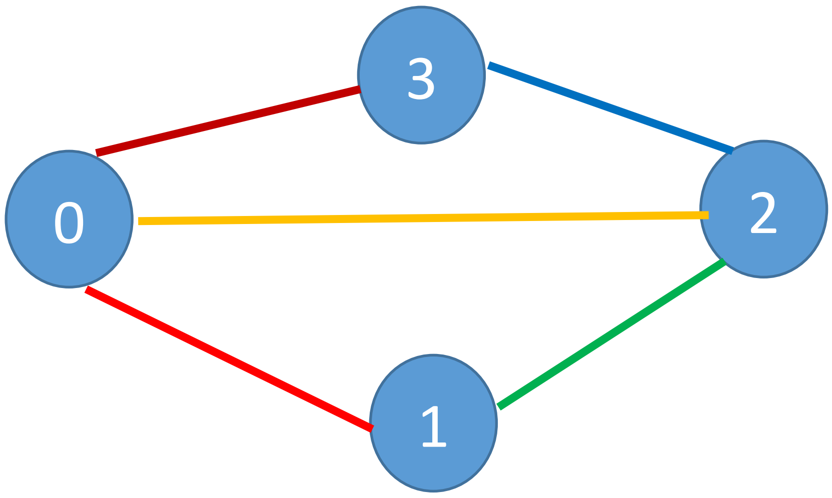
Finally the Hamiltonian that is extracted from the cost function is:

$$\hat{H} = \sum_{<i,j>} w_{ij}\frac{Z_i Z_j}{2}$$

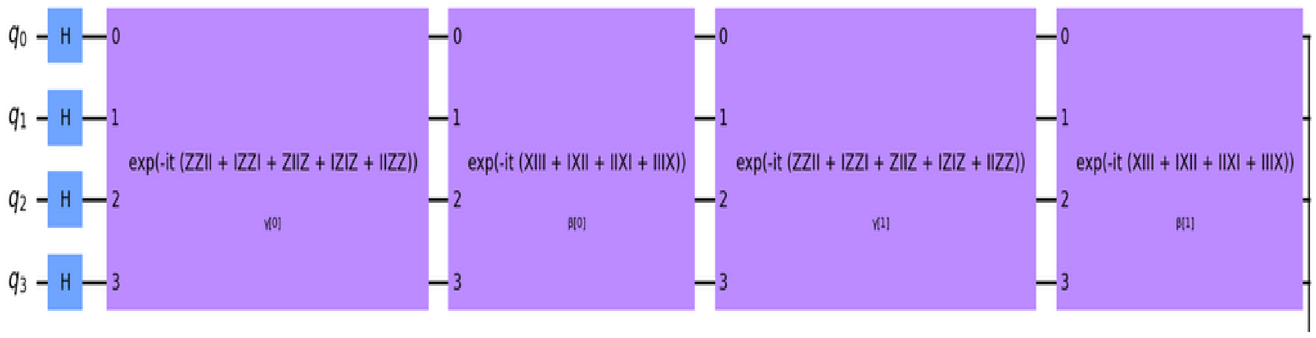
Max-Cut with QAOA

In our example:

$$\widehat{H} = IIZZ + IZZI + ZIIZ + ZZII + IZIZ$$



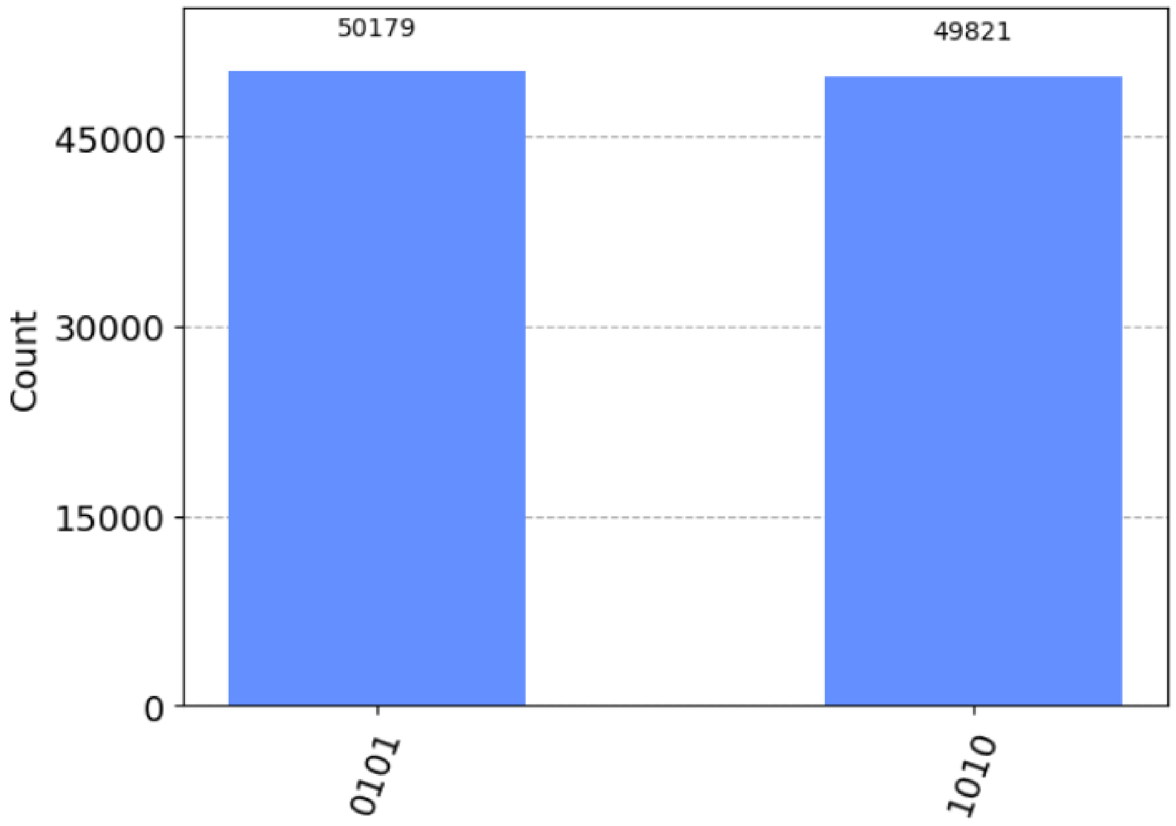
We will use a quantum circuit with 2 QAOA layers to find the optimal cut.


q_0
q_1
q_2
q_3
H
H
H
H
exp(-it (ZZII + IZZI + ZIIZ + IZIZ + IIZZ))
γ[0]
exp(-it (XIII + IXII + IIXI + IIIX))
β[0]
exp(-it (ZZII + IZZI + ZIIZ + IZIZ + IIZZ))
γ[1]
exp(-it (XIII + IXII + IIXI + IIIX))
β[1]

The initial parameters are randomly selected from U(0, 2π)

The classical optimizer is a gradient free optimizer, COBYLA

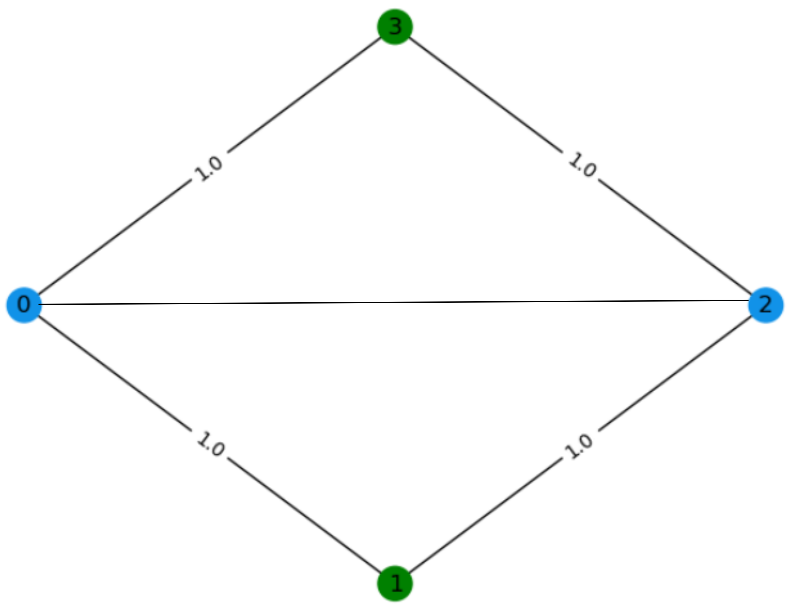
After the optimization ends we plot the probability distribution



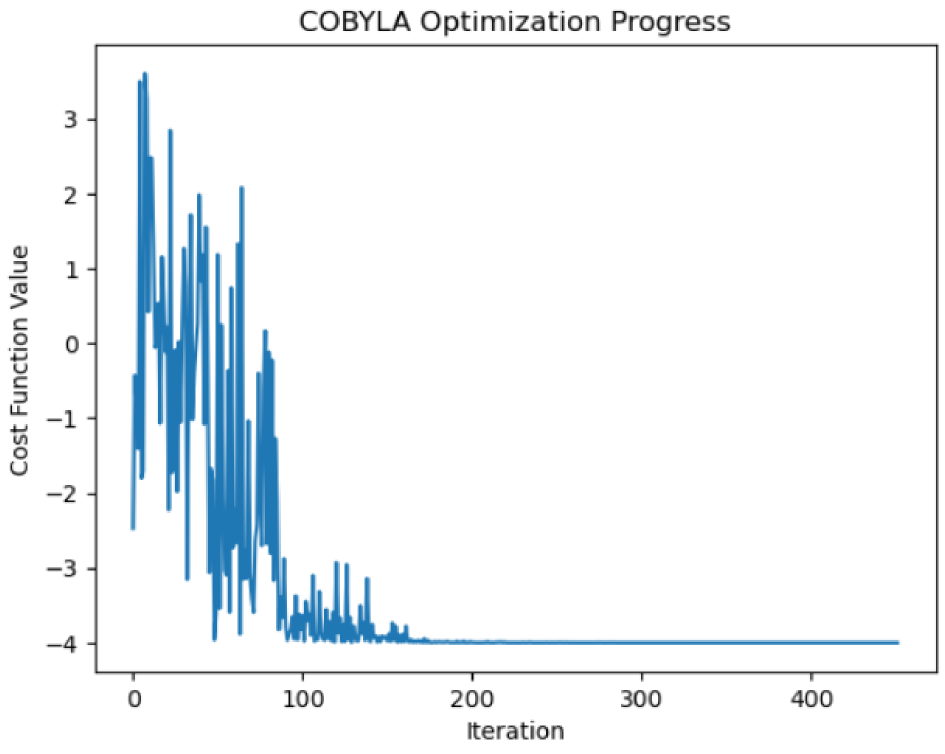
The optimal parameters where found:

- ParameterVectorElement(β[0]): 3.7009079201772983
- ParameterVectorElement(β[1]): 3.5938508537102445
- ParameterVectorElement(γ[0]): 2.689335690118754
- ParameterVectorElement(γ[1]): 4.153129710694556

We can derive the optimal cut from the optimal solution given by the histogram and divide the nodes to two subgroups. One group contains green nodes and the other blue nodes. It is easily observed that the 2 groups are {0, 2} & {1, 3}.



The optimization process is shown in the figure below:



The cost function converged in 200th iteration

ADAPT QAOA

(Adaptive Derivative Assembled Problem Tailored - Quantum Approximate Optimization Algorithm)

Framework

In QAOA the variational ansatz consists of p layers, each containing the cost Hamiltonian H_C and a mixer, H_M :

$$\left|\psi_p(\vec{\gamma},\vec{\beta})\right\rangle = \left(\prod_{k=1}^{p}\left[e^{-iH_M\beta_k}e^{-iH_C\gamma_k}\right]\right)|\psi_{\text{ref}}\rangle$$

Where $\left|\psi_{ref}\right\rangle = |+\rangle^{\otimes n}$, n is the number of qubits. The initial state is prepared by applying Hadamard gate to all the n qubits.

The variational parameters $\boldsymbol{\gamma_k}$ & $\boldsymbol{\beta_k}$ are chosen such that the $\langle\boldsymbol{\psi_p}(\boldsymbol{\gamma},\boldsymbol{\beta})|\boldsymbol{H_C}|\boldsymbol{\psi_p}(\boldsymbol{\gamma},\boldsymbol{\beta})\rangle$ is minimized, then the resulting energy and state provide an approximate solution to the optimization problem.

The accuracy of the result and the efficiency with which it can be obtained depend sensitively on H_M .
In the standard QAOA ansatz, the mixer is chosen to be a single-qubit X rotation applied to all qubits.

In ADAPT-QAOA the ansatz consists of the standard cost Hamiltonian H_C and the mixing Hamiltonian is replaced by one operator extracted from an *operator pool.*

Operator Pool

The mixer operator is selected from a collection of operators. In order to define this collection, **suppose a set of qubits Q**. The mixer operator of the standard QAOA is a single operator, thus the pool for the standard QAOA is $P_{QAOA} = \left\{\sum_{i\in Q} X_i\right\}$. If we want to enrich our collection, we need to include two additional operator pools. one consisting entirely of single-qubit mixers, and one

with both single-qubit and multi-qubit entangling gates:

- $P_{QAOA} = \{\sum_{i\in Q} X_i\}$
- $P_{single} = \cup_{i\in Q} \{X_i, Y_i\} \cup \{\sum_{i\in Q} X_i\} \cup P_{QAOA}$
- $P_{multi} = \cup_{i\times j\in Q\times Q} \left\{B_i C_j | B_i, C_j \in \{X, Y, Z\}\right\} \cup P_{single}$

Each operator pool contains O(1), O(n) and O(n^2) elements respectively.

Optimization Process

The quantum ansatz is grown one layer at a time using a gradient-based selection criterion. At each iteration k, operators A_j from the predefined mixer pool are evaluated via the energy gradient:

$$-i\langle\psi^{(k-1)}|e^{i\gamma_0 H_C}[H_C, A_j]e^{-i\gamma_0 H_C}|\psi^{(k-1)}\rangle$$

where Hc is the cost Hamiltonian, γ0 is the initial variational parameter of the problem Hamiltonian. The operator A_k with the largest gradient norm is selected and appended to the circuit as $e^{-\beta_k A_k} e^{-\gamma_k H_c}$, producing the updated variational state:

$$|\psi^k\rangle = e^{-\beta_k A_k} e^{-\gamma_k H_c} |\psi^{k-1}\rangle$$

After each layer is added, all variational parameters {βm, γm} are re-optimized to minimize the cost function ⟨ψ(k)|Hc|ψ(k)⟩. The iterative process continues until the gradient norm falls below a predefined threshold.

Steps:

1) Initialize circuit, apply Hadamard gates

2) Initialize the operator pool **A**

3) Compute the gradient for each operator A_j
   $-i\langle\psi^{(k-1)}|e^{i\gamma_0 H_C}[H_C, A_j]e^{-i\gamma_0 H_C}|\psi^{(k-1)}\rangle$

4) If the max gradient is less than a predefined threshold, then return the current circuit

5) If not append the operator with the largest gradient as $e^{-\beta_k A_k}e^{-\gamma_k H_c}$

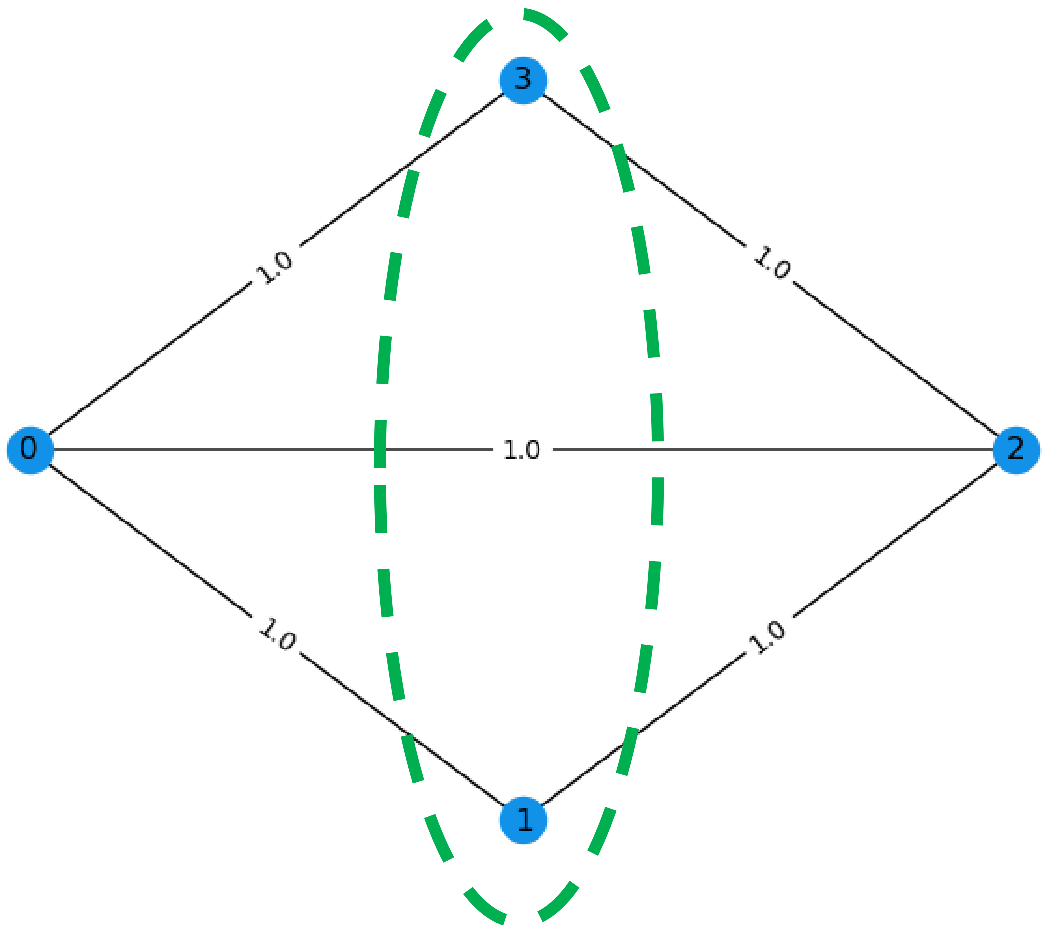
6) The updated state is $|\psi^k\rangle = e^{-\beta_k A_k}e^{-\gamma_k H_c}|\psi^{k-1}\rangle$

7) Optimize all the parameters $\{\beta_m, \gamma_m\}_{m=1}^{k}$

8) Return to step 3

Max-Cut with ADAPT-QAOA

Suppose we have the graph:



The obvious solution (- - -) is to divide the nodes to two groups {0,2}&{1,3}

The Hamiltonian is equal to

```
cost_h = SparsePauliOp.from_list([("ZZII", 1), ("IZZI",
1), ("ZIIZ", 1), ("IZIZ", 1), ("IIZZ", 1)])
```

<u>We set the threshold to stop the optimization equal to thres = 0.1</u>

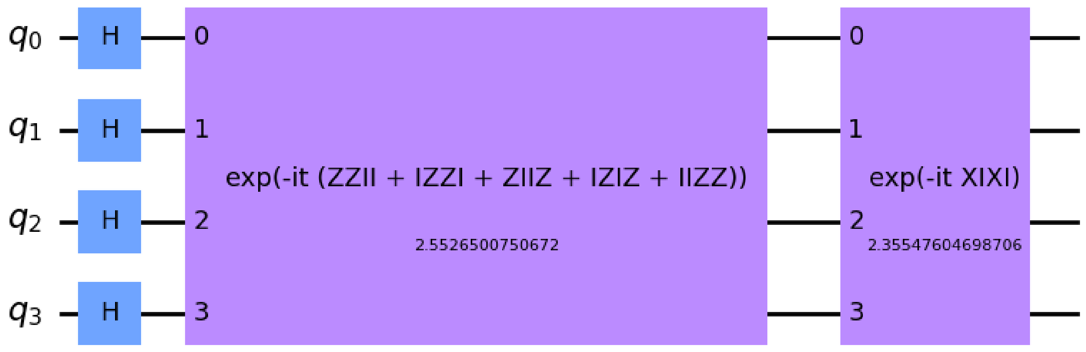
<u>The initial γ value is $\gamma_0 = 0.01$</u>

<u>The operator pool contains 64 elements</u>

Executing the algorithm, the output is:

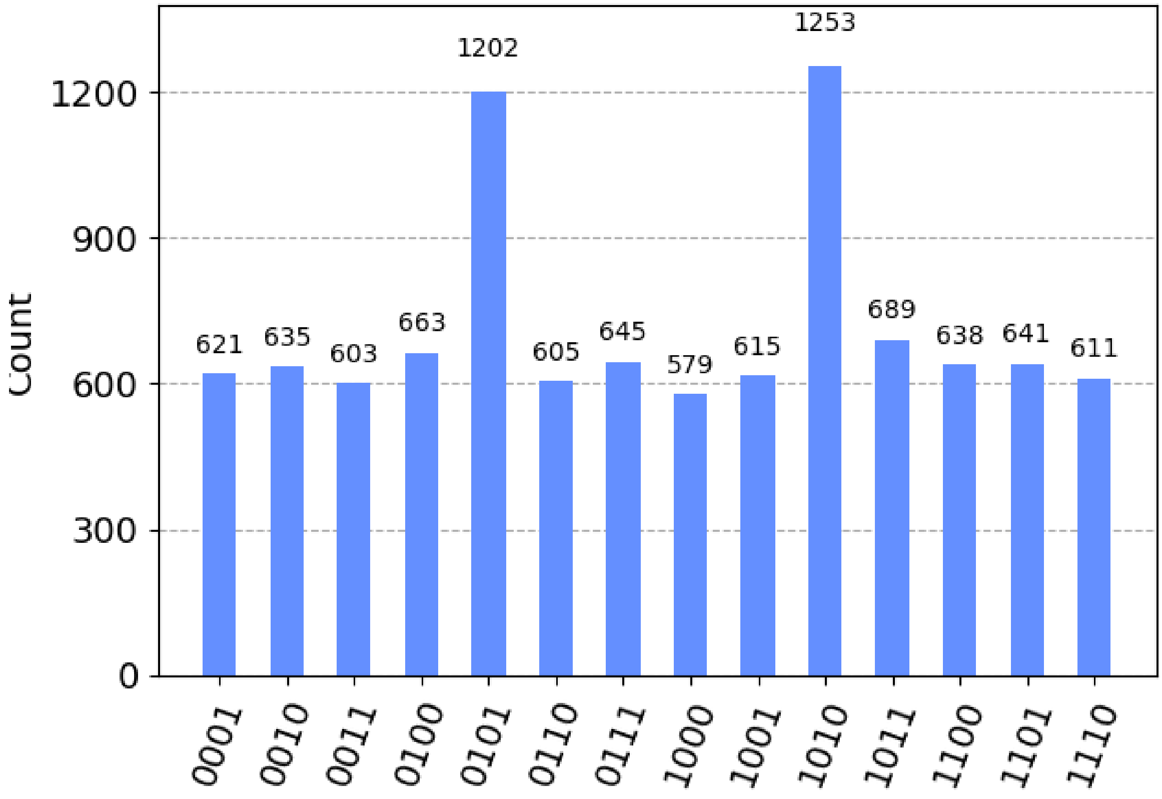
```
--- ADAPT Step 1 ---

Gradient norm: 0.477012

Adding operator
SparsePauliOp(['XIXI'],

          coeffs=[1.+0.j]) with
gradient 0.160171

Step 1 minimized energy: -
0.9999986078285611


--- ADAPT Step 2 ---

Gradient norm: 0.085417

Converged!
```

The optimal ansatz is:

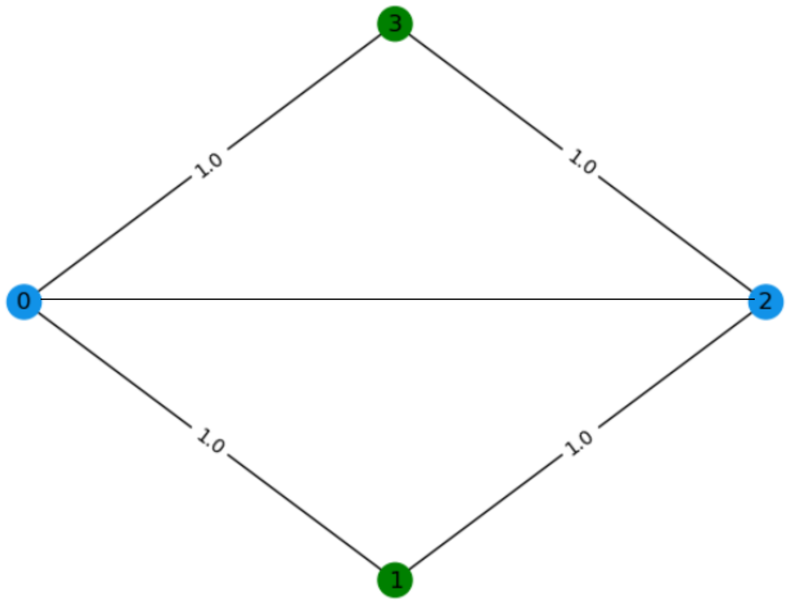


As we can see, we were lucky and the algorithm converged after 1 iteration. After the first step the algorithm found the best operator (XIXI) and in the next iteration the max gradient was lower than the thres=0.1 so the optimization stopped.

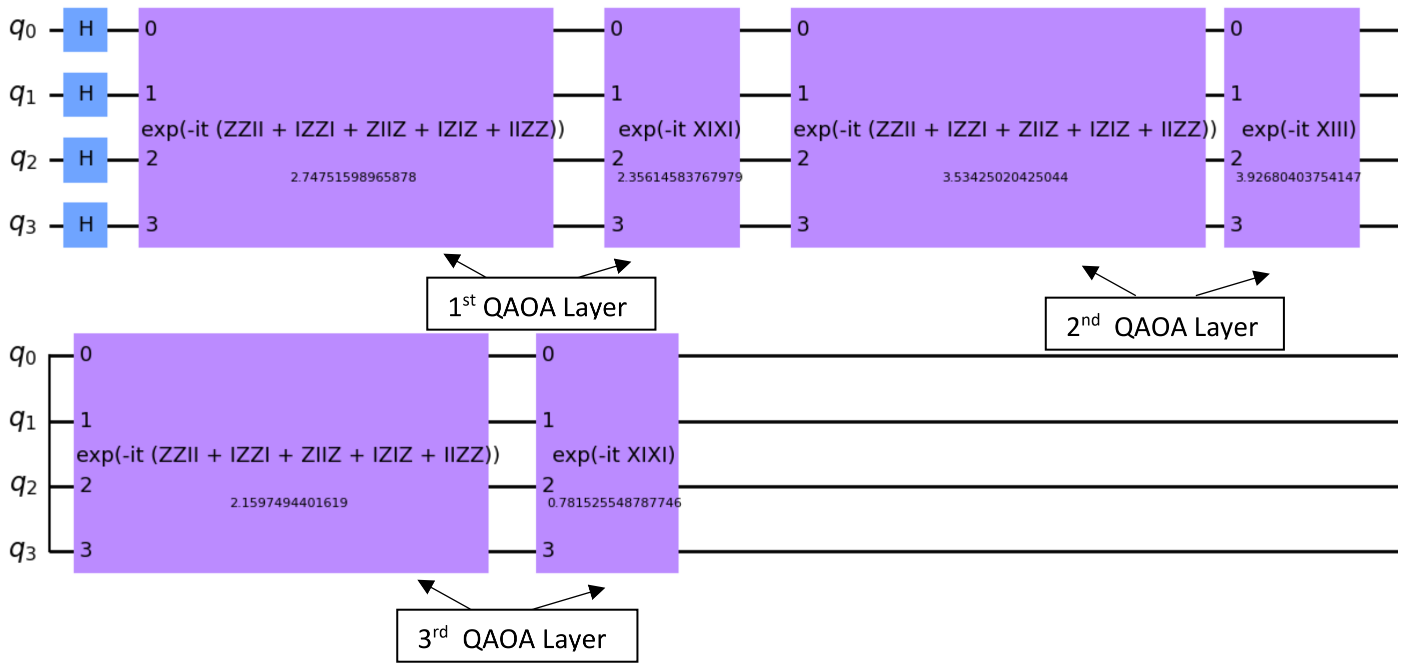
The histogram that shows the probabilities distribution of the optimal state is:

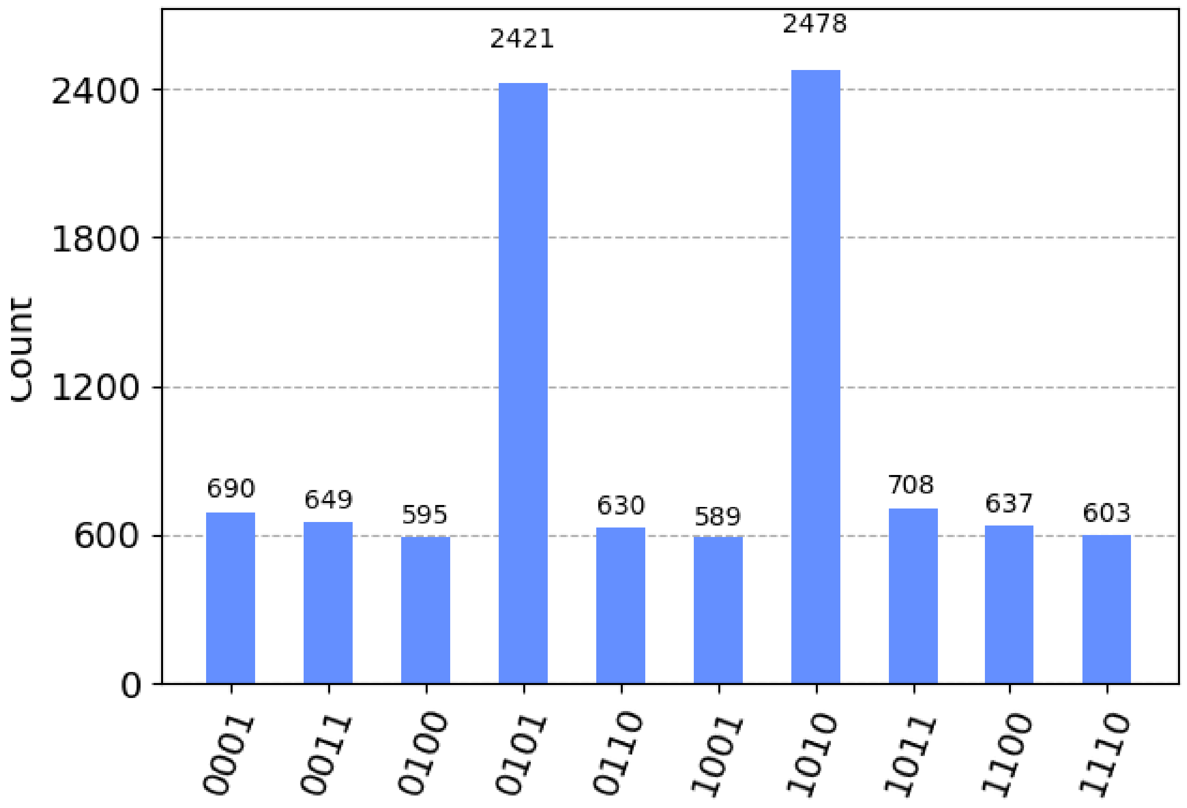


The top bitstings {0101 & 1010} indicate that the nodes should be divided to these as shown below: (green and blue group)



If we are not lucky and the process more optimization runs to converge then the optimal circuit would require more layers but the optimal bitstrings are separated more clearly from any other solution





Graph Level embedding (FEATHER)

FEATHER is a non-parametric graph embedding method based on characteristic functions that represent local distributions of node features. For each node u, the method computes the r-scale random walk weighted characteristic function:

$$\phi_u(\theta, r) = \sum_{w \in V} \hat{A}^r_{u,w} \cdot e^{i\theta x_w}$$

where $\hat{A} = D^{-1}A$ is the normalized adjacency matrix and $\hat{A}^r_{u,w}$ is the probability of reaching node *w* from u in *r* steps via a random walk. This function is evaluated at a set of d frequencies $\theta \in \Theta$ to form a complex-valued embedding that captures the higher-order structure and attribute distribution around each node. Graph-level representations are constructed by pooling node-level embeddings (e.g., using the mean).

$x^{\rightarrow} = x_1, \dots, x_{|V|}$, set of features, each node has as ingle value as a feature

$x_w = \log(\deg(w) + 1)$, the feature of each node

$\theta^{\rightarrow} = \theta_1, \dots, \theta_k$, set of k frequencies

We will compute the graph level embedding for all the u nodes and then will apply mean pooling

```
$Data$ :
$\widehat{A}$ : Normalized adjacency matrix.
$\overrightarrow{X}$ : $[x_1, x_2, \ldots, x_{|V|}]_{1\times|V|}$ vector of node features
$\overrightarrow{\theta}$ : $[\theta_1, \theta_2, \ldots, \theta_k]_{1\times k}$ vector of k eval points
$r$ : Scale of empirical graph characteristic function
-------------------------------------------------------------
1. $H = x^T\theta$
2. $H = [cos(H)|sin(H)]$
3. $Z = []$
4. for i in range(r):
5.         $H = \widehat{A} \cdot H$
6.         $Z = [Z|H]$
7. $return$ Z
```

This is a graph level embedding the node level embedding is extracted by calculating the mean value for every value of θ_i.

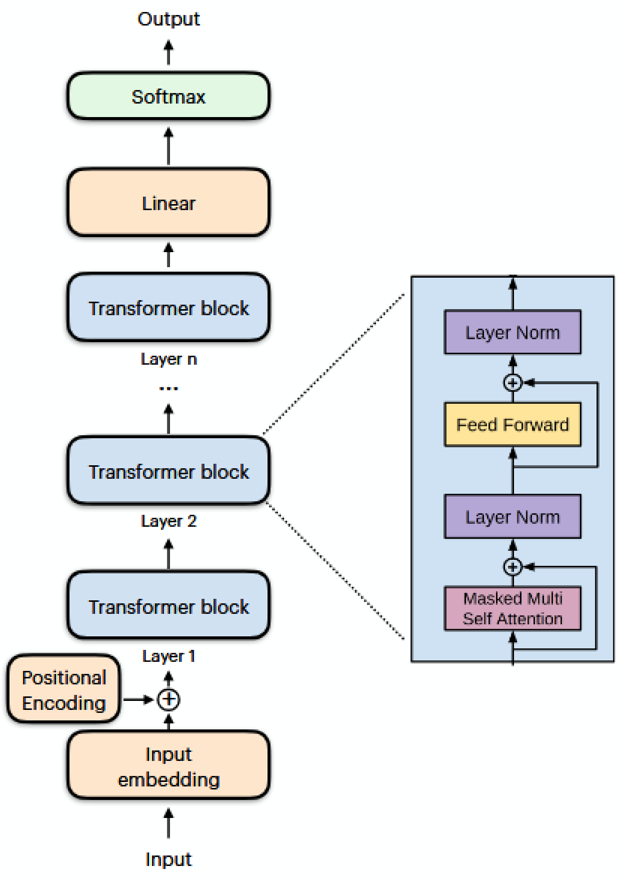
Thus the result is a vector with dimensionality $\mathbb{R}^d$, where $d = 2rk$

GPT-LLM

(Generative Pre-trained Transformer – Large Language Models)

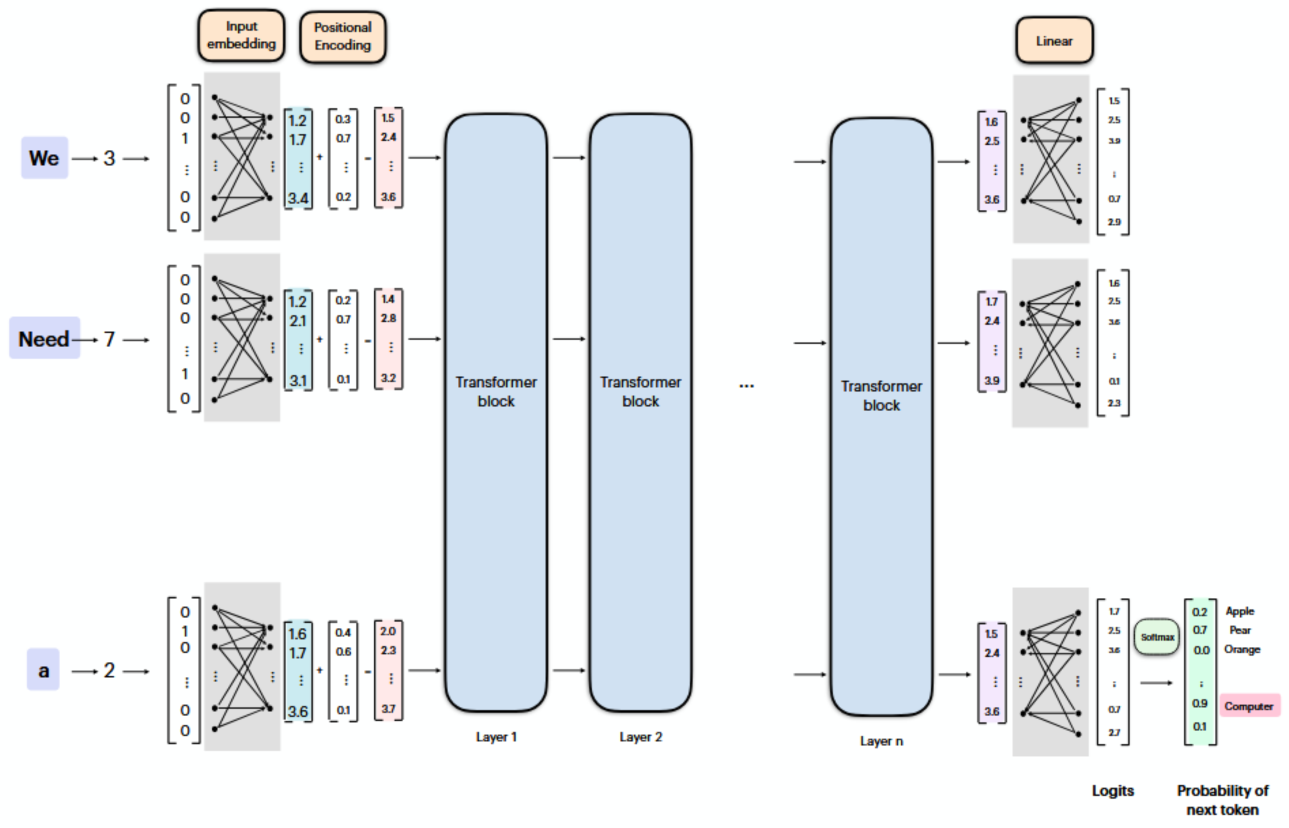
The emergence and rapid advancement of Large Language Models (LLMs) such as ChatGPT has had a significant global impact, revolutionizing our interactions with artificial intelligence and expanding our understanding of its capabilities.

GPT’s architecture is a multi-layer decoder-only Transformer. The primary part of the architecture is a stack of transformer blocks, each of which is composed of two main components: **a (masked) multi-head self-attention** mechanism followed by a position-wise fully connected **feed-forward network**. **Layer Normalization** and **Residual Connections** are placed around these two main components. The transformer blocks are stacked on top of each other, with each layer processing the output of the previous one.



Next word prediction process of GPT

In the inference stage, a language model (here, GPT) takes in a sequence of one-hot encoded tokens, and generates predictions for the next word in a sequence. The sequence of one-hot encoded tokens is first transformed through word embedding. These embeddings, after the positional encodings are added, are then input into the transformer blocks. Then, a final linear layer is applied to map the outputs from the transformer blocks back into the vocabulary space, generating a sequence of transformed vectors. The last transformed vector is passed through a softmax activation function, yielding a probability distribution across the vocabulary, indicating the likelihood of each word as the next sequence component.



Model Architecture and Training QAOA-GPT

The final training set is used to train QAOA-GPT, a decoder-only Transformer model based on the nanoGPT implementation of GPT-2. The model was trained from scratch and was not initialized from any pre-trained earlier model due to using custom circuit tokenization schema (i.e., we do not use fine tuning or prompt engineering).

The token embeddings are sequencies of graph-circuit tokens

$$graph\_token = < bos > (i, j), w, \dots, < end\ of\ graph >$$

$$circuit\_token = < new\ layer > o_k, \gamma_k, \beta_k \dots$$

$$token = [\, graph\_token \mid circuit\_token \,]$$

Given a tokenized sequence $x_{1:T}$, the model uses:

• Token embeddings $E_{tok} \in \mathbb{R}^{V \times d}$,

• Positional embeddings $E_{pos} \in \mathbb{R}^{T \times d}$,

• Graph embeddings $e_G \in \mathbb{R}^{d}$ from FEATHER.

The Transformer input is computed as:

$$X = E_{tok}(x_{1:T}) + E_{pos} + e_G \otimes 1_{1 \times T}$$

The dataset was created by generating graphs from the Erdős–Rényi random graph model G(n, p) where p was set to p = 0.5

20 ER graphs with n = 4 nodes

30 ER graphs with n = 5 nodes

27 ER graphs with n = 6 nodes

Total 77 graphs

Train/Test split: 90/10

Vocabulary size: 563 , after generating the files: train.bin, test.bin

We train using cpu (we can use gpu for accelerated performance)

```
gpt
└──➤ nanoGpt
│    └──➤ config
│    │    └──➤ train_maxcut.py
│    └──➤ data
│    │    └──➤ maxcut
│    │         └──➤ prepare.py
│    │         └──➤ meta.kpl
│    │         └──➤ train_feathers.npy
│    │         └──➤ val_feathers.npy
│    │         └──➤ train.bin
│    │         └──➤ val.bin
│    └──➤ out-qaoa-maxcut
│    │    └──➤ Ckpt.pt
│    └──➤ configurator.py
│    └──➤ model.py
│    └──➤ sample.py
│    └──➤ train.py
└──➤ adapt_qaoa.py
└──➤ feather.py
└──➤ gpt_token.py
└──➤ max-cut_adapt.py
└──➤ feather.txt
└──➤ graph_circuit_tokens.txt
```

We do only 1500 iterations because the dataset is small and the loss ratio reaches a value below 0.05 at step 1500

- Data: vocab_size=563, block_size=103
- Training with device: cpu, dtype: bfloat16
- Step 0: Train loss 6.3191, Val loss 6.3217
- Step 500: Train loss 0.7060, Val loss 4.8919
- Step 1000: Train loss 0.3566, Val loss 7.0324
- Step 1500: Train loss 0.0218, Val loss 8.0325

File structure and instructions:

```
>>python gpt_token.py
```

To generate the training dataset. Creates graph_circuit_tokens.txt and feather.txt

```
>>python nanoGpt/data/maxcut/prepare.py
```

Tokenizes the txt files. Creates .bin .npy and .kpl files which are the input to the transformer

```
>>cd nanoGpt
```

```
>>python train.py config/train_maxcut.py –device=cpu (optional)
```

Training process. The configurations are fixed in the config file

```
>>python sample.py
```

Generates new graph. The new graphs are generated from the **Barabasi-Albert model** BA(n, m) n = Number of nodes
m = Number of edges to attach from a new node to existing nodes

<u>Suppose we completed every step and we execute the last command:</u>

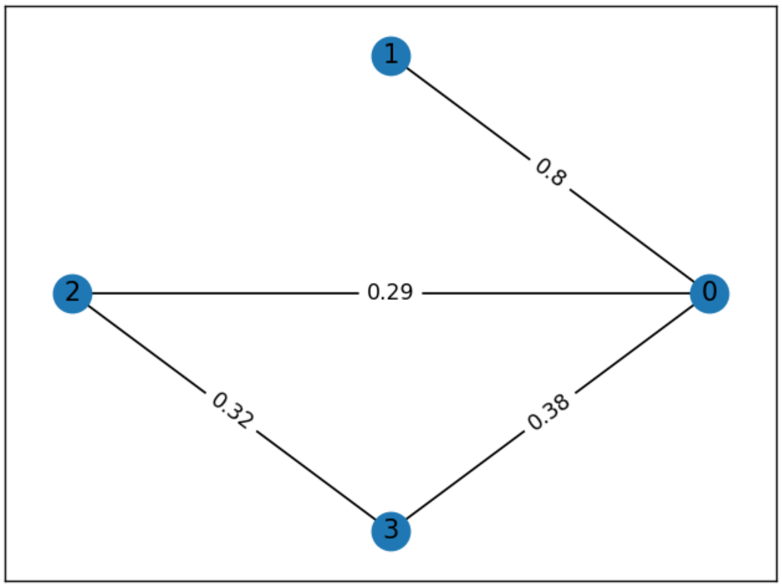
The output is the following:

Graph token for random BA graph <bos> (0, 1) 0.8 (0, 2) 0.29 (0, 3) 0.38 (2, 3) 0.32 end_of_graph

Generated tokens:

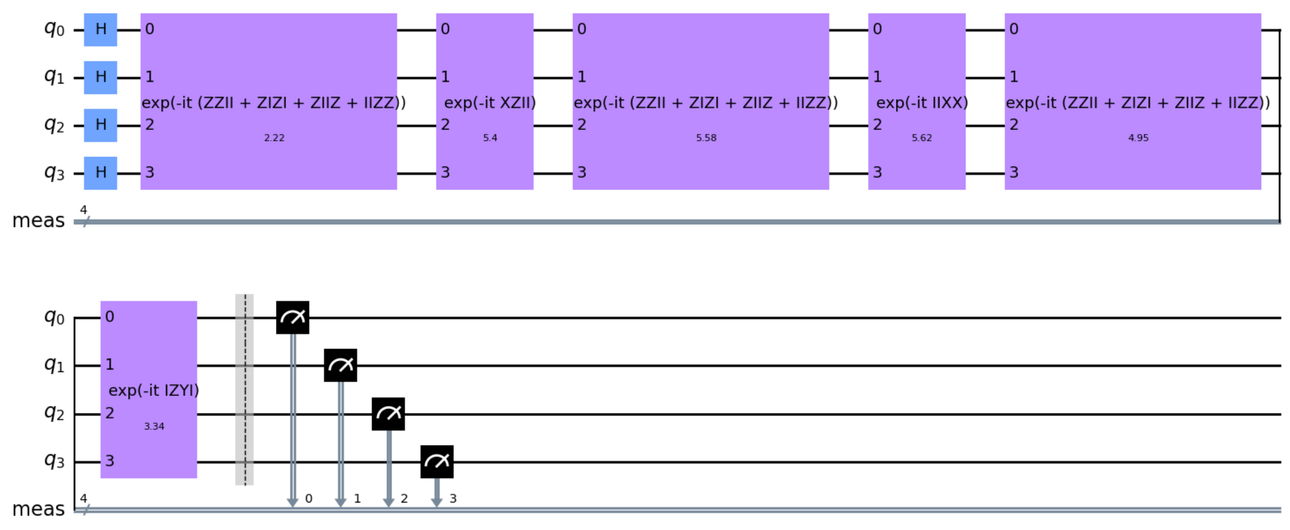
<bos> (0, 1) 0.8 (0, 2) 0.29 (0, 3) 0.38 (2, 3) 0.32 end_of_graph <new_layer> 12 2.22 5.4 <new_layer> 3.55 <new_layer> 84 2.03 3.94 <new_layer> 48 0.68 52 2.41 2 0.53 (1, 4) 0.98 (1, 4) <new_layer> 55 5.58 5.62 <new_layer> 44 4.95 3.34 <bos> (0, 1) 0.25 (1, 4) 0.81 (2, 3) 0.47 3.87 <bos> (0, 5) 0.02 (2, 3) 0.51 (2, 3)

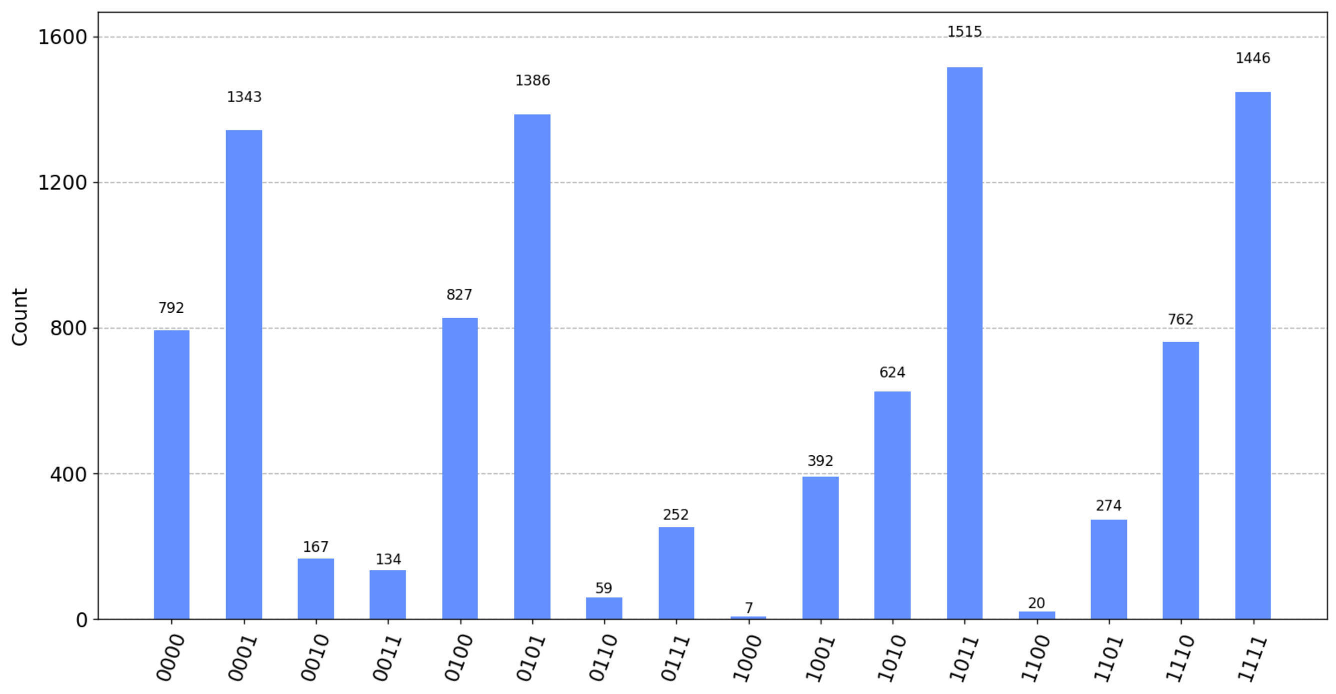
The new graph is:



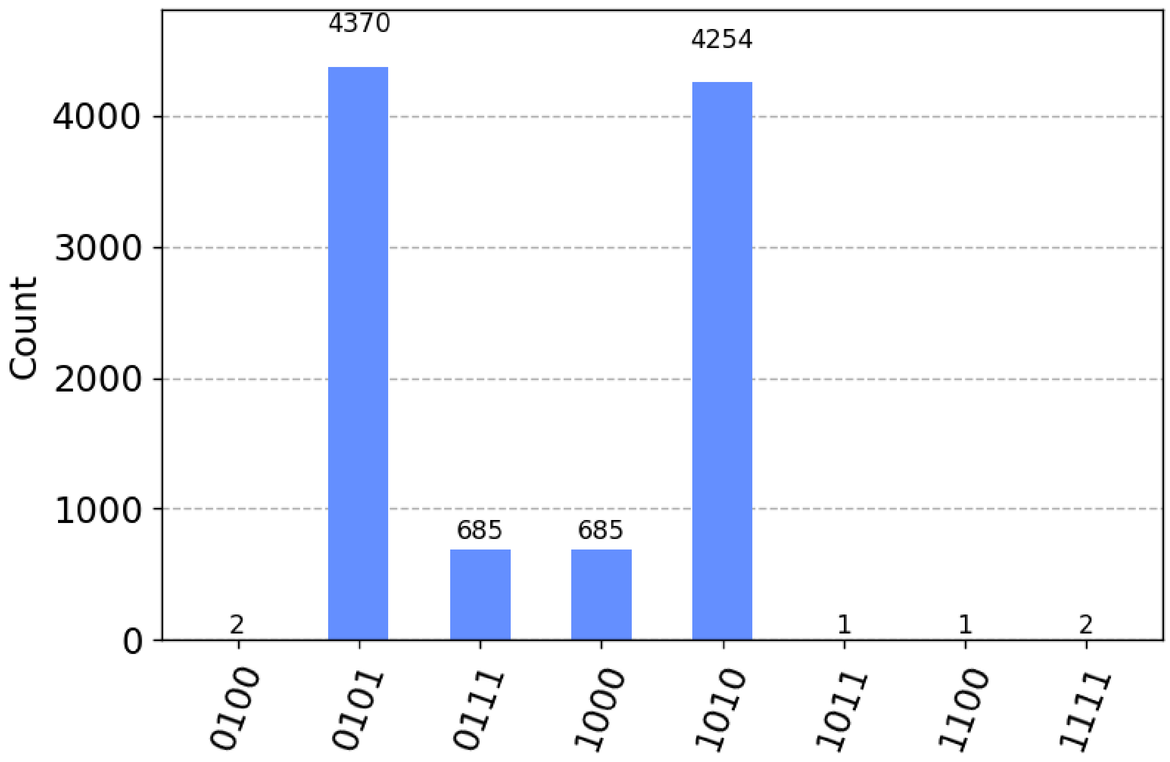
The generated circuit token contains garbage data so after data preprocessing we keep only the layers that contain $o_k \; \gamma_k \; \beta_k$.

graph <new_layer> 12 2.22 5.4 <new_layer> 55 5.58 5.62 <new_layer> 44 4.95 3.34


q_0
q_1
q_2
q_3
meas
H
exp(-it (ZZII + ZIZI + ZIIZ + IIZZ))
2.22
exp(-it XZII)
5.4
exp(-it (ZZII + ZIZI + ZIIZ + IIZZ))
5.58
exp(-it IIXX)
5.62
exp(-it (ZZII + ZIZI + ZIIZ + IIZZ))
4.95
exp(-it IZYI)
3.34



The resulted plot histogram from the generated circuit



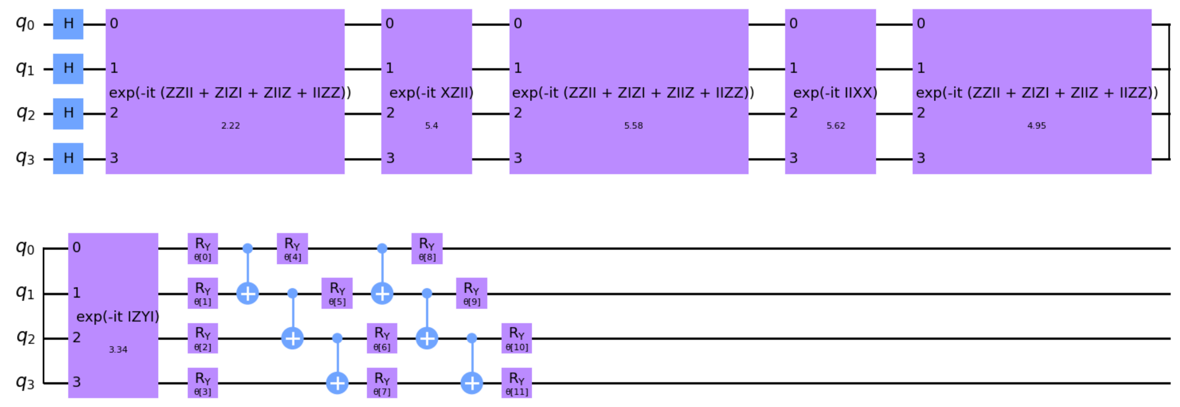
The resulted plot histogram by solving the same problem with ADAPT QAOA

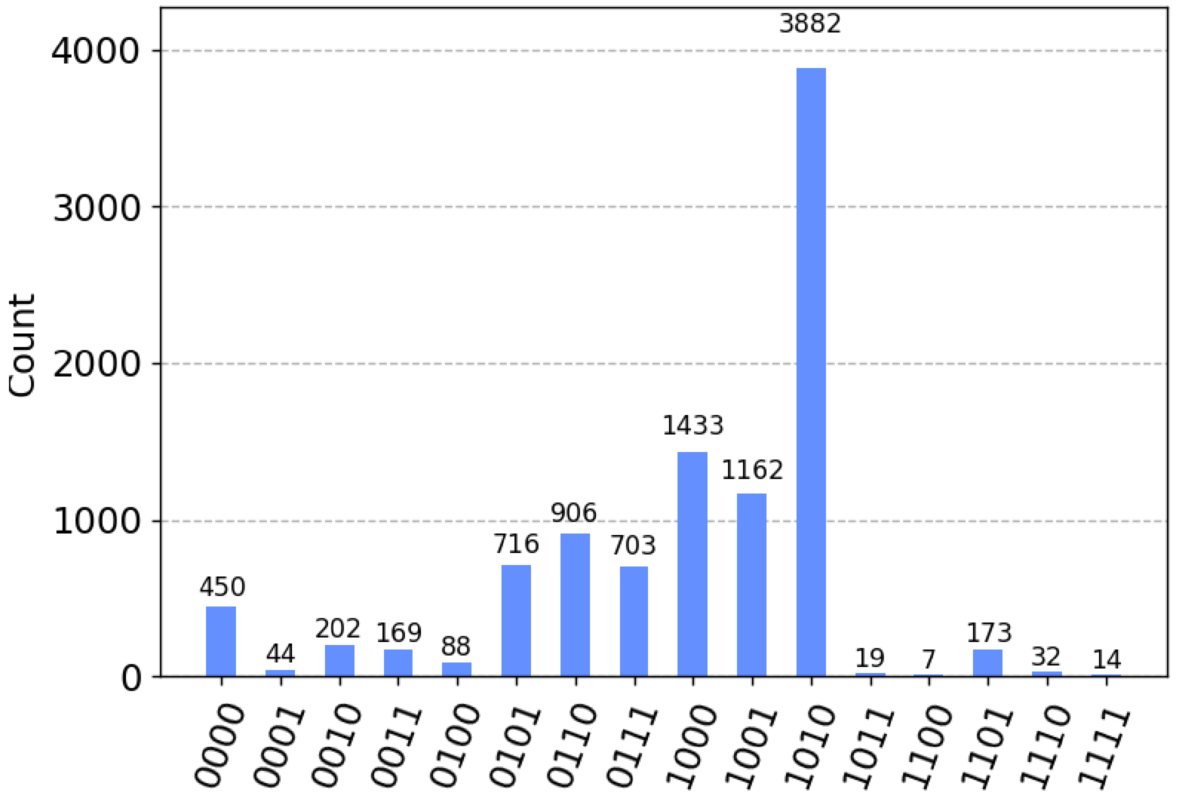
Observation:

The GPT wasn’t trained thoroughly we stopped the training process very early. The plot histogram from the generated circuit includes the 2 optimal solutions with high probability but they aren’t separated from suboptimal solutions (optimal solutions: 1010 & 0101).

Even though, we can use the generated circuit as warm start for solving the same problem with only 2 rotation layers

The idea is to create a circuit **CIRC** -> Generated | 2-layers VQA





The plot histogram with QAOA_GPT as warm start for VQA (achieved the optimal solution, 1010)

<u>**Summary:**</u>

Even though out gpt model wasn’t trained for a long time it still produced promising results and had some benefits in accelerating the optimization process for a variational algorithm with only 2 layers of rotation and entanglement

References

1. https://arxiv.org/abs/2504.16350 QAOA-GPT

2. https://arxiv.org/abs/2005.10258 ADAPT-QAOA

3. https://learning.quantum.ibm.com/course/variational-algorithm-design/variational-algorithms Variational Algorithm Design

4. https://arxiv.org/abs/1411.4028 QAOA

5. https://arxiv.org/abs/2005.07959 FEATHER

6. https://github.com/benedekrozemberczki/FEATHER/blob/master/src/feather.py FEATHER source code

7. https://github.com/karpathy/nanoGPT/tree/master nano-gpt source code

8. https://anonymous.4open.science/r/multivariateGPT_anon-4ED4/ multivariate-gpt

9. https://arxiv.org/abs/2403.09418 gpt on a quantum computer