Determining the Minimum Surface Code Distance Required for Effective QAOA (Work In Progress)

Tanner Smith

Abstract

The Quantum Approximate Optimization Algorithm (QAOA) is a hybrid quantum-classical algorithm used to approximate solutions to combinatorial optimization problems such as Max-Cut, Ising energy minimization, and general constraint satisfaction problems. However, current quantum algorithms suffer from noise produced by the hardware, including decoherence of qubits and imperfect gate operations. In this work, I investigate how the required code distance scales for reliably executing QAOA and how much protection algorithms like QEC require. I simulate various logical error rates based on surface-code distances on a variety of graphs and analyze the output to answer this question. To achieve this goal, I introduce a weighted average relative error rate (WARAR) that emphasizes the layers that meaningfully contribute to the algorithmic performance and demphasizes those that are either too shallow to achieve useful AR, or so deep that noise inevitably dominates.

1 Introduction

Quantum computing promises to solve a number of problems in a wide variety of fields such as business, physics, and chemistry. However, current NISQ-ero quantum algorithms suffer from noise produced by the hardware, including decoherence of qubits and imperfect gate operations. These errors make obtaining meaningful results from quantum circuits difficult. Of the many quantum algorithms proposed, the Quantum Approximate Optimization Algorithm (QAOA) is on that has been more heavily examined recently due to its near-term applications. QAOA is used to approximate optimization problems by minimizing an equation of the form

$$\sum_i h_i z_i + \sum_{i,j} J_{ij} z_i z_j$$

by tuning γ and β parameters within a circuit composed of controlled rotations. It is widely believed that QAOA is able to produce useful results even amidst high error rates that would be unacceptable for other algorithms. However, there is yet to be any research done to systematically quantify the extent of the algorithms error tolerance. This gap in the literature is particularly important because QAOA is a promising candidate for achieving quantum advantage during the NISQ era.

With the era of fault-tolerant quantum computing approaching, error detection and correction have become increasingly effective. Quantum error-correcting methods such as surface codes promise to reduce the effects of errors on quantum algorithms and increase the fidelity of operations. In the case of surface codes, as the code distance d increases, the logical error rate (i.e. aparent error rate) of quantum systems decreases. If QAOA is able to produce meaningful output amidst reasonable noise levels, and surface codes are able to decrease the logical error rates of circuits, a question arises - "at what point is error correction sufficient for QAOA to produce meaningful results?" Or, more formally, "what is the minimum surface-code distance required for effective QAOA?" Answering this question would provide insight not only into the practical feasibility of QAOA on early fault-tolerant machines, but also into how resources should be allocated when designing quantum architectures intended to run optimization workloads.

In this project, I aim to simulate various logical error rates based on surface-code distances on a variety of graphs and analyze the output to answer this question.

2 Background

Before diving into the main question of this project, we provide a brief overview of the main topics discussed in this work: QAOA and quantum error correction, with a particular focus on surface codes.

2.1 Quantum Approximate Optimization Algorithm (QAOA)

The Quantum Approximate Optimization Algorithm (QAOA) is a hybrid quantum-classical algorithm used to approximate solutions to combinatorial optimization problems such as Max-Cut, Ising energy minimization, and general constraint satisfaction problems. The algorithm works by minimizing a cost function of the form

$$C(z) = \sum_i h_i z_i + \sum_{i,j} J_{ij} z_i z_j \tag{1}$$

where $z \in \{0,1\}^n$ is a bitstring encoding assignments to problem variables. here, h_i denotes the linear coefficient of variable z_i, and J_{ij} is the quadratic coefficient represents the coupling between variables z_i and z_j. QAOA problem

instances are commonly represented as graphs, where each node corresponds to a variable z_i, each node is assigned with a weight h_i, and each edge (i, j) is assigned a weight J_{ij}. This formulation is consistent with the quadratic unconstrained binary optimization (QUBO) and Ising models often used in optimization and physics. To approximate the minimizer of this cost function, QAOA employs tunable parameters γ and β that adjust the weighting and relative phase of different candidate solutions across the problem space respectively. These parameters are iteratively optimized by a classical routine to steer the quantum circuit toward high-quality solutions.

The QAOA circuit is constructed from the structure of the problem instance, with one qubit assigned to each node in the graph. The circuit consists of p layers, each following the same basic template and characterized by its own pair of parameters (γ_l, β_l). Initially, an ansatz is selected by assigning random or heuristic values to the γ_l and β_l parameters. The algorithm begins by preparing all qubits in the $|+\rangle$ state via a Hadamard gate. Then for each layer, two things happen. (1) A phase-seperation operator is applied, consisting of single-qubit R_z rotations implementing the h_i terms and two-qubit controlled R_z rotations implementing the J_{ij} terms on edges of the graphs. These operations are parameterized by γ_l and encode the cost Hamiltonian into the circuit. (2) A mixing operator is applied by performing R_x rotations on all qubits, parameterized by β_l in order to explore the solution space. Finally, the results of the circuit are sent to a classical optimizer which tunes the γ and β parameters to favor configurations yeilding lower costs. This process repeats until a z is found such that the objective function is minimized.[2]

2.2 Surface Codes

Quantum error correction (QEC) is a method for mitigating the effects of noise on quantum computations. Of the many error-correcting codes developed, one of the most promising is the surface code, due to its compatability with the two-dimensional grid layouts used in many leading quantum hardware platforms. Surface codes encode a single logical qubit using many physical qubits arranged on a 2D lattice. There are two types of physical qubits: data qubits and measurement qubits. Data qubits store the encoded (logical) quantum information. Measurement qubits on the other hand are used to measure stabelizers. Stabelizers are operators that commute with logical operations, but anticommute with errors and allow the system to detect errors without collapsing the logical state. These stabelizer measurements determine whether the configuration of data qubits corresponds to a valid codework. If a stabelizer measurement yields an unexpected outcome, it signals that an error has occured.

Surface codes rely on stabilizer operators to detect both bit-flip (X) and phase-flip (Z) errors. In the standard planar or rotated surface code, X-type stableizers correspond to checks involving products of Pauli-X operators on neighboring data qubits, and Z-type stabilizers correspond to similar products of Pauli-Z operators. These stabelizers define a subspace of valid codewords: the logical information resides in the simultaneous +1 eigenspace of all stabelizers. Thus, deviations from this subspace indicate that hte system has left the valid codeword space - i.e. an error in the data qubits has occured.

A key property of a surface code is its code distance d, which is the minimum number of physical-qubit errors required to go from one valid logical state to another. On surface codes, this d corresponds to the side length of the lattice, and oftentimes this code distance is odd due to the checkerboard pattern of X and Z stabelizers. More importantly, d also directly relates to the number of errors that can be corrected by a surface code - namely, a surface code with code distance d can detect up to $d - 1$ errors and correct up to $\lfloor (d-1)/2 \rfloor$ errors.

Physically, a distance-d surface code requires $O(d^2)$ physical qubits to encode one logical qubit. So while larger code distances suppress logical error rates exponentially, they increase hardware overhead quadratically which creates a tradeoff between error protection and resource requirements.

Error detection and correction occur continuously through the execution of a quantum algorithm, and these iterations are known as surface-code cycles and can be thought of as rounds of stabelizer measurements. During each cycle, all stabilizers are measured and errors are detected. Multiple cycles are often needed to reliably detect errors, however, the exact frequency at which surface-code cycles are executed depends on the code structure and algorithm being run. More frequent cycles increase overhead, but offer higher fault tolerance by catching errors sooner, before they can propagate.

As quantum processors continue approaching the fault-tolerant era, understanding how logical error rates scale with code distance as well as how much protection algorithms like QAOA require is essential for estimating realistic resource needs. Surface codes currently represent a leading candidate for achieving these low logical error rates due to their high thresholds and compatibility with existing hardware architectures. [3]

3 Project

In order to tackle the question of what code distance is required to reliably run QAOA, I investigated several factors that influence both the performance of QAOA and the effectiveness of logical error suppression. These factors include minimum layer requirement, minimum size requirement, and modeling LER of surface codes with distance d, which are explained in the sections below.

3.1 Identifying Minimum Layer Requirement

Since QAOA can be run with multiple layers, and the quality of the output generally improves with additional layers, it was necessary to choose an appropriate number of layers for the simulations. Too few layers can prevent the algorithm from reaching near-optimal solutions, while too many layers can increase computational cost and simulation time. Additionally, at higher layers, error becomes

exceedingly harder to correct and thus, in non ideal simulators, there is a point at which increasing layers results worse results.

To determine a reasonable lower bound on the number of layers, I referred to [1]. This work shows that, theoretically, with too few layers it is impossible to guarantee that the optimal cost can be achieved. As the author states, the algorithm must "see the entire graph" to adequately explore the solution space. Intuitively, the locality of QAOA requires that there must be enough layers for the circuit to encode interactions across the full graph and for the classical optimizer to access the corresponding solution landscape. The necessary number of layers p can be estimated using the formula:

$$2p \geq \frac{w \log n}{\log(d/\ln 2)}, \quad w < 1, d \text{ sufficiently large} \tag{2}$$

Here, n is the number of nodes in the graph, d represents the average degree of the nodes in the graph [1]. I chose $d = 3$ since I will be testing on 3 regular graphs, and $w = 0.999$ to be as aggressive as possible. Table 1 shows the minimum number of layers required for several example graph sizes. In practice, since QAOA circuits cannot have fractional layers, the ceiling of the calculated value is shown in parenthesis.

Table 1: required layer count p for graphs of size n

Nodes (n)	**min layers (p)**
4	0.472 (1)
6	0.611 (1)
10	0.785 (1)
12	0.847 (1)
18	0.985 (1)
20	1.02 (2)
25	1.09 (2)
50	1.33 (2)
100	1.57 (2)
500	2.11 (3)
1000	2.35 (3)

This analysis provided a lower bound for the number of layers to simulate which I will label l_{th}. From Table 1, we see that for small graphs, the minimum required layers is fewer than 3. It is important to note that l_{th} here is the necessary number of layers for accurate QAOA, but NOT sufficient. Having l_{th} layers does not guaranty accuracy, but having fewer than l_{th} layers guarantees innacuracy. More specifically, you cannot obtain an AR greater than 0.854 [1]. To ensure I tested well beyond this threshold, I ran on circuits up to 9 layers.

This threshold was considered when evaluating the performance of the noisy simulations to the ideal baseline. As will be explained further in section 4.2, the closeness of the noisy simulator relative to the ideal simulator was considered more heavily for layers beyond this threshold.

3.2 Determining Number of Nodes required for Advantage

While quantum computing promises computational advantages over classical methods thanks to quantum phenomena such as entanglement and superposition, these advantages only appear once sufficiently large problem instances can be executed. In the case of QAOA, which is a hybrid quantum-classical algorithm, the total runtime is influenced not only by quantum circuit execution but also by classical post-processing, communication overhead between hardware layers, and repeated circuit evaluations required for parameter optimization. Because of this hybrid overhead, at small scales QAOA has been shown to be slower than classical brute-force methods [4].

Thus, even if we identify a code distance d that performs well on small problem sizes, such results will not necessarily reflect performance at larger scales. We want to figure out what code distance is required for *practically* useful QAOA, and so we must also determine the problem size at which QAOA begins to outperform classical algorithms. The authors of [4] compare classical optimization to QAOA for various depths, and by extending their results through a simple extrapolation of their time-per-node scaling trend, we estimate that QAOA begins to outperform classical methods at about 600 nodes for $p = 4$ and about 650 nodes for $p = 8$. This threshold is far beyond the limits of computational hardware available to me, so any analysis of effective logical error rates must be scaled to the graph sizes at which QAOA becomes efficient from smaller graphs. Taken together, these considerations imply that the useful code distance is not the one that works for small benchmark graphs, but rather one that supports beyond 3-layer QAOA on graphs with 700+ nodes.

3.3 Determining LER based on d

Since implementing error-correcting codes within an algorithm is complex and has significant overhead, I opted instead to simulate error-correcting codes. To do this I needed a way to relate the surface code distance d to the resulting logical error rate (LER). If I could compute the LER for each code distance, then I could construct a noisy simulator whose effective noise strength mimics that of a device protected by a surface code of distance d.

To model this I abstracted away the specific syndrome extraction and decoding process and focused only on the logical error rate defined as the probability that a logical qubit fails after one error correction cycle. The authors of [3] find that the relationship between code distance and logical error rate of standard surface codes follows Equation 3.

$$LER \approx 0.03 \left(\frac{p}{p_{th}}\right)^{\lceil d/2 \rceil} \tag{3}$$

In this equation, the 0.03 applies an overall scaling to fit the formula to the observed trend when using standard surface codes. The parameter p represents the base physical error rate of the machine per surface code cycle, and

the parameter p_{th} is a threshold error rate based on the surface code implemented.

To incorporate this into my simulator, I chose to add noise to gate operations and then use this formula to tailor the error of the gates using p as the base noise level of the gate. In a way, this simulates as if error correction was carried out after every gate operation, but it allows me to simulate code distances by simply updating the physical error rates of the gates using the LER formula and choosing a p based on the gates themselves. In my experiments, I chose $p = 0.003$ based on the IBM hardware stats from lab 2 to model current-day superconducting hardware. However, this value may vary between hardware implementations. For the threshold value, I used $p_{th} = 0.0057$, as reported in [3], which is lower than previous thresholds because it incorporates additional error sources present in realistic surface code implementations. Substituting these values into the formula yields

$$LER \approx 0.03 \left(\frac{0.003}{0.0057} \right)^{\lceil d/2 \rceil} \tag{4}$$

This is the expression I used to assign effective noise levels to the physical gates in my simulations. Table 2 lists several code distances and their corresponding logical error rates.

Table 2: Code Distance (d) vs LER

d	**LER**
3	0.01145
5	0.00603
7	0.00317
9	0.00167
11	0.000879
13	0.000463
15	0.000243
17	0.000128
25	0.0000098

3.4 Experiment Setup

To investigate how the required code distance scales for reliably executing QAOA, I constructed a series of experiments on randomly generated 3-regular graphs. For each experiment I varied the following parameters:

- Graph weight distribution:
  - random choice weighting $\{-1, 1\}$
  - uniform distribution $[-1, 1]$
  - gaussian distribution $[-1, 1]$
- Graph sizes: $n \in \{4, 6, 8, 10, 12, \}$ (graph sizes beyond 14 nodes were infeasible with my implementation, even on Alpine)
- QAOA Layers: $p \in \{1, 2, 3, 4, 5, 7, 9\}$ (ensuring every graph size tested included sufficent depths based analysis from Section 3.1)
- Noise level / code distance: $d \in \{0 \text{ (no QEC)}, 5, 7, 9, 13, 15, 17\}$. I also ran the ideal simulator as a baseline.

Each $(size, topology, weighting)$ combination defines a unique base graph, which I treat as a single experimental instance. For each instance, all simulations acrosss code distance and layer count were performed on the same graph and the same QAOA Hamiltonian coefficients. This isolates the effect of depth and noise, preventing variation due to graph randomness which could affect trends.

For every configuration of $(n, topology, weight, d, p)$, I performed five optimization runs, each with randomly initialized γ and β parameters, and recorded the best achieved approximation ratio (AR) among the five runs. This mitigates issues with local minima and poor optimizer initialization, both of which are common in QAOA landscape as the number of nodes increases.

Along with AR, I also stored cirucit depth, runtime, and optimized parameters per (p, d) pair.

3.5 Running Experiments

To run my experiments, I created a Jupyter Notebook using Python. I used Qiskit for quantum circuit generation and simulation with custom noise models to model LER for each code distance.

For graphs of size 4 to 8 nodes, I ran on my local machine which has an 11th gen Intel(R) Core(TM) i7-1165G7 2.80Ghz processor and 16GB of RAM. For graphs of size 10+ nodes, I used CU Boulder's Alpine supercomputer. I created one job for each experiment $(n, topology, weight)$. I ran these jobs on Alpine's Amilan cluster, allocating 16-32 CPUs each with 4-8GB of memory.

4 Results

4.1 layer vs AR tables

Figures 1 to 9 present the relationship between QAOA layer count and approximation ratio (AR) across different code distances, graph sizes, and weight distributions. Each graph corresponds to a specific graph type (topology, weight pair) and within the graph, each line represents a different code distance and the shape of the line shows the AR achieved (y-axis) when using a certain number of layers (x-axis). As expected, the general trend is that increasing the code distance leads to improved AR, which reflects the fact that adding layers reduces the chance of qubit errors within the circuit. One thing to note is that due to the time the simulations took as the number of nodes increased, I only did random choice weighting for beyond 12 nodes and was only able to do a couple code distances for the graph with 14 nodes.

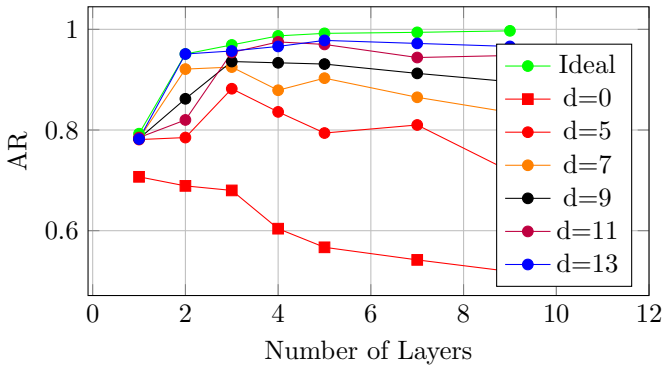

Figure 1: 4 node, Random Choice

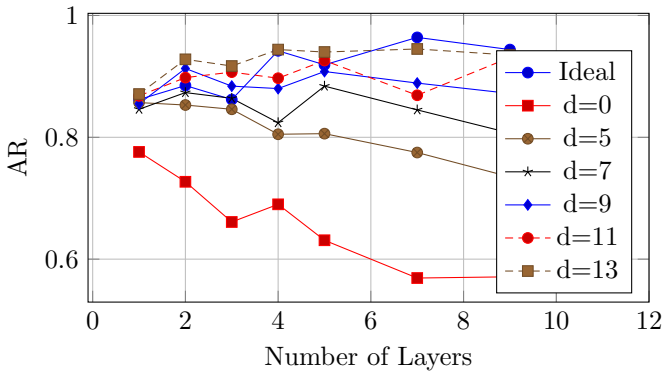

Figure 2: 4 node, Uniform Distribution

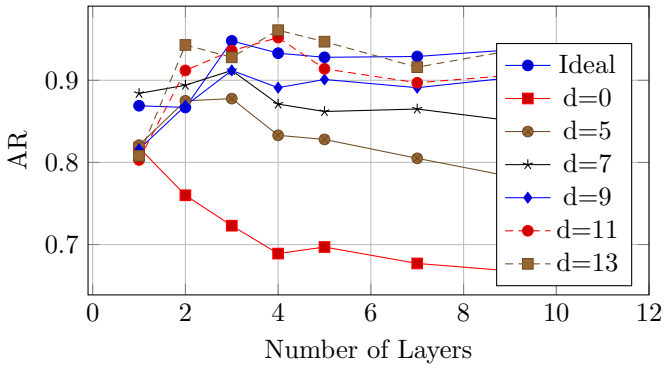

Figure 3: 4 node, Gaussian Distribution

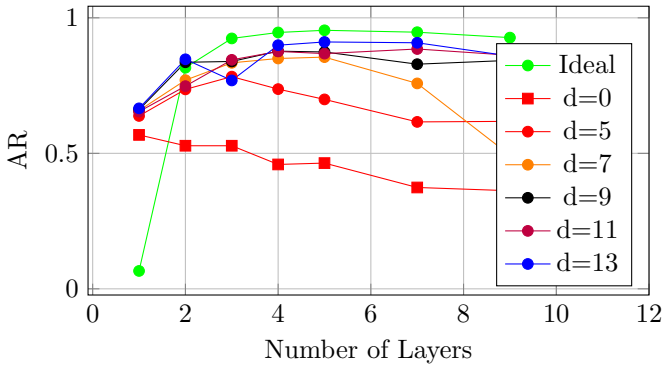

Figure 4: 6 node, Random Choice

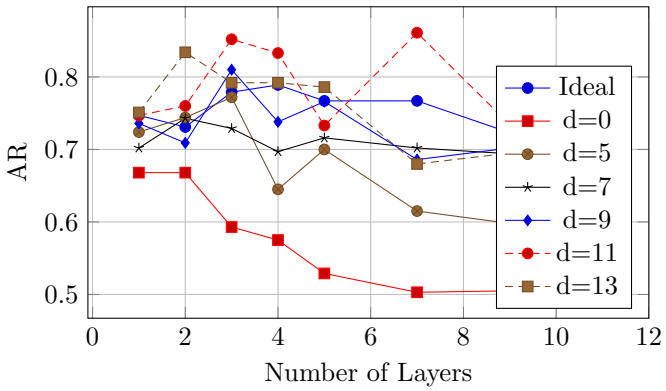

Figure 5: 6 node, uniform Dist

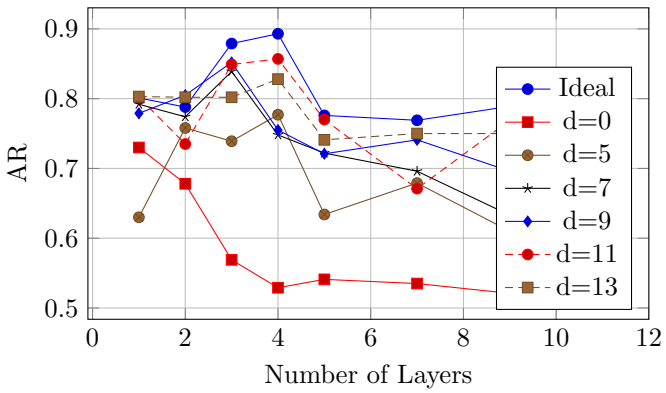

Figure 6: 6 node, Gaussian Dist

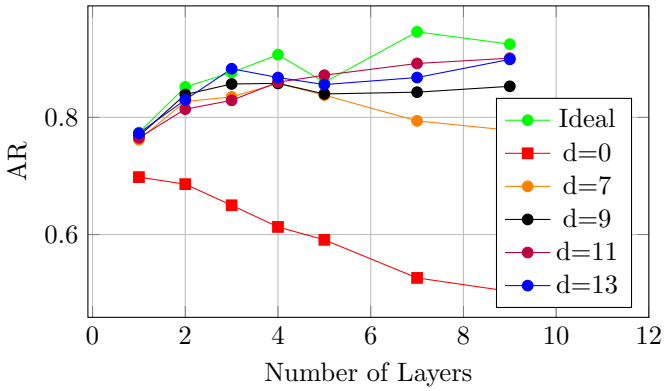

Figure 7: 8 node, Random Choice

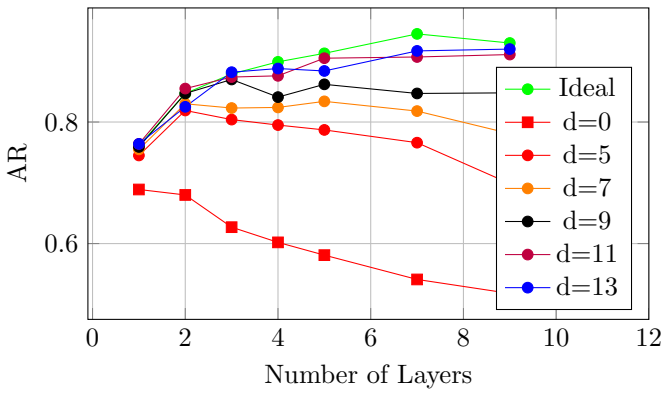

Figure 8: 10 node, Random Choice

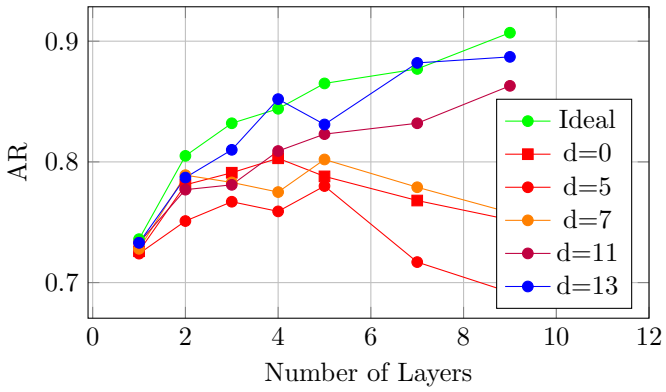

Figure 9: 12 node, Random Choice

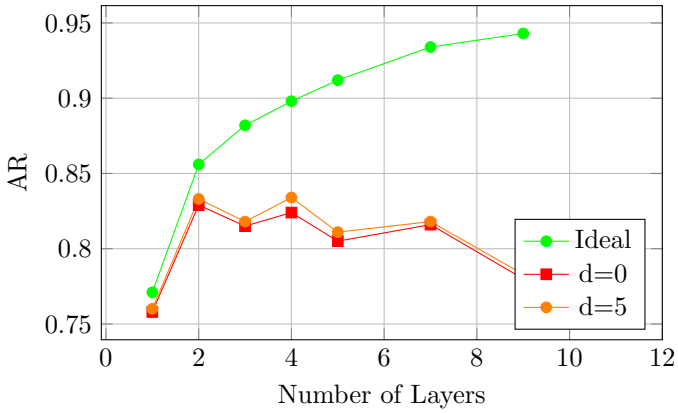

Figure 10: 14 node, Random Choice

4.2 Weighted Average Relative AR

To determine the code distance required for QAOA to run effectively, we compare the relative approximation ration between the noisy error-corrected simulations and the ideal noiseless simulations. Naively we could compute the relative AR at each layer l and simply average these values. However, this isn't quite optimal in practice because while the ideal AR increases logarithmically with the number of layers, noise accumulates exponentially. As $l \to \infty$, any noisy simulation will eventually collapse in performance. This would mean that a simple average, when considering many layers, would eventually be dominated by these late-layer failures. However, realistically, the ideal simulator will achieve an AR that is acceptable and any additional layers are unnecessary, thus it is less important that the noisy simulator stays as accurate as the ideal beyond this point. To address this, I introduce a weighted average relative AR (WARAR), where the relative AR of each layer l is assigned a weight $f(l)$ that emphasizes the layers that meaningfully contribute to the algorithmic performance and demphasizes those that are either too shallow to achieve useful AR, or so deep that noise inevitably dominates. First, we define a minimum meaningful layer l_{th} using equation 2, which gives the smallest l for which QAOA can reach $AR > 0.854$. Since QAOA is incapable of producing accurate results (I consider 0.854 not accurate enough) with fewer than l_{th} layers, we set

$$f(l < l_{th}) = 0$$

Next, we identify the point where the ideal QAOA instance reaches the AR we consider "acceptable". Let l_{ideal} be this smallest layer at which the ideal simulation attanins an acceptable AR. If no layer run on the ideal simulator achieves your desired AR, set l_{ideal} equal to the maximum layer tested plus one because later layers are required to get close to your ideal AR. For this project I will consider 0.9 to be acceptable but this can be adjusted. Our weight function should place maximal emphasis on this layer and smoothly reduce the weight for layers that are significantly smaller (too shallow) or larger (increasingly dominated by noise). To achieve this I went with a Gaussian (bell curve) centered at l_{ideal}, scaled so that $f(l_{th}) = 0.1$. This construction satisfies the desired properties that $f(l) > 0$ for all l, $f(l) \to 0$ as $l \to \infty$, and maximum weight is given to l_{ideal}. The full resulting WARAR and weight function is given in Equation 5

$$WARAR(n,d) = \frac{\sum_{l \in Layers} f(l,n) \frac{AR_{ideal}(l)}{AR_{noisy-d}(l,n)}}{\sum_{l \in layers} f(l,n)} \quad (5)$$

where

$$f(l,n) = \begin{cases} 0 & \text{if} \quad l < l_{th}(n) \\ \exp(-\frac{(l-l_{ideal})^2}{S}) & \text{otherwise} \end{cases}$$

where in order to satisfy that $f(l_{th}) = 0.1$, we set

$$S = \frac{-(l_{th}(n) - l_{ideal})^2}{\ln(0.1)}$$

In the equations above, l_{th} is given a parameter of n (number of nodes in the graph) because Equation 2 is based on the size of the graph. Additionally, $AR_{ideal}(l)$ is the AR of the ideal simulator with l layers, and $AR_{noisy-d}(l)$ is the AR of the noisy simulator implementing code distance d with l layers.

This weighted average gives more import to layers required for accurate QAOA and removes import from layers beyond what is necessary, which could negatively affect the average in the long run. Figure 11 plots the relationship between the size of the graph (x-axis) and the WARAR (y-axis) for each code distance. This shows how WARAR changes as the size of the graph increases. We can see that as we increase d, our WARAR increases, which we expect. Without implementing surface codes we obtain a WARAR around 0.6 to 0.8, while code distances of $d = 7$ have a WARAR around 0.9, and $d = 13$ obtains a WARAR of a high 0.95.

4.3 Scaling WARAR to answer the research question

Figure 11 shows the WARAR for up to 12 nodes. However, from section 3.2 we know that we require at least 600 to 700 nodes for quantum advantage. Therefore, we must identify a trend in the WARAR and extrapolate to determine the required code distance at 600+ nodes. Due to the small number of graphs I was able to test, this isn't optimal, but the WARAR of each code distance seems to exhibit a relatively flat trend. I will be honest, this isn't

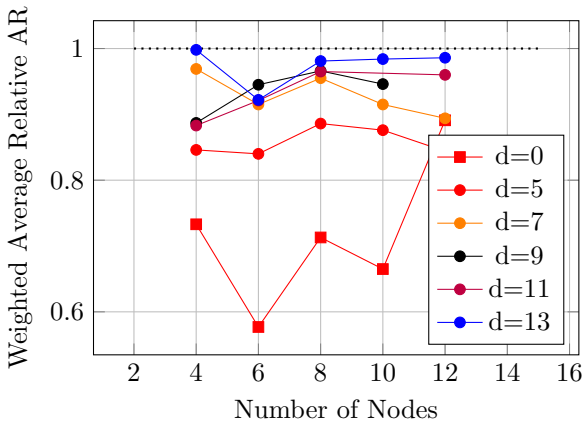

Figure 11: random Choice Weighted Average Relative AR

quite what I was expecting. I thought that the WARAR would begin to decrease as we increased the number of nodes as the required layer increased and circuits required more noisy operations to reach a layer in which the ideal simulator would even achieve a high enough accuracy. It is worth noting, though, that the WARAR of a few of the code distances, $d = 5, 7, 9$, do begin to drop slightly on the higher end of the node count, but I would require tests up to 16 or so nodes to be sure.

For accurate QAOA, we will require the noisy simulator to be at least 95% the ideal because even the ideal simulator only obtains an AR a little over 0.9. With the data presented, if the trend is that the WARAR stays constant, we can expect the d required to be around 11 or 13. However, I would reckon that once I run QAOA on a few more larger graphs, we will see a drop in WARAR so I presume the required d is actually larger. Though how much larger will depend on the trend.

5 Future Work

While this project was a good starting point that allowed me to provide a rough estimate for the number of nodes required to run QAOA effectively, given more time, I would like to further refine some of the metrics and experiments I conducted. Firstly, I would like to try to run simulations on more graph topologies. Currently, I only use 3-regular graphs, as they provide the clearest results for QAOA. However, I would like to extend this to random graphs, complete graphs, and power-law graphs. Additionally, I would like to test beyond the 12 nodes I did in this paper. QAOA only provides an advantage beyond 600 nodes, and I feel that the long-term trends are not very recognizable from the small sample set of graphs I could run. However, due to the exponential growth of quantum algorithms, and the possibility that my functions are not currently fully optimized, the runtime became to great, even on Alpine (not to mention the continuous roadblocks I faced along the way). Along with scaling the quantity and size of experiments, I would also like to take a deeper look into the calculation of LER and improve the implementation. I currently model LER using the generic formula provided by [3] and adjust the error rates of gates, but this may not be entirely accurate for real-world scenarios. Instead, I would like to use a surface code simulating software, such as Stim, to do the calculations instead, to get a more accurate model. This way I could more accurately incorporate errors like decoherence as well.

References

[1] Edward Farhi, David Gamarnik, and Sam Gutmann. The quantum approximate optimization algorithm needs to see the whole graph: A typical case, 2020.

[2] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[3] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3), September 2012.

[4] G. G. Guerreschi and A. Y. Matsuura. Qaoa for max-cut requires hundreds of qubits for quantum speed-up. *Scientific Reports*, 9:6903, 2019.