VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Quantum Maze Solver Using Grover's Algorithm

Project Report

Submitted by:

Madhav Gera (22BIT0400)

Aryan Singh Bisht (22BIT0252)

Uday Sharma (22BIT0055)

Course: SLOT B2 - Quantum Computing

ABSTRACT

For this project, we decided to tackle something pretty cool: using Grover's quantum search algorithm to solve mazes. We built a system that generates mazes with multiple possible solutions and then uses quantum computing to find valid paths through them. We also added some neat visualization features so you can actually see how the algorithm amplifies solution probabilities as it runs.

Here's the thing though - we tested different maze sizes and compared our quantum approach with the classic A* pathfinding algorithm, and we discovered something important. While our quantum algorithm works exactly as it should, there's a huge bottleneck: the classical preprocessing we need to do just to set up the quantum circuit. In our 5x5 maze test, A* blew through it in 0.13ms, while our quantum approach took 352ms total (with 85.7ms of that being classical preprocessing alone). This discovery led us down a rabbit hole exploring quantum random walks as a potential way forward.

Keywords: Grover's Algorithm, Quantum Computing, Maze Solving, Qiskit, Amplitude Amplification

1.INTRODUCTION

Let's start with the basics. Quantum computing is fundamentally different from the regular computing you're used to. Instead of bits that are either 0 or 1, quantum computers use qubits that can actually be both 0 and 1 at the same time (this is called superposition). Throw in some other quantum weirdness like entanglement and interference, and you've got a system that can solve certain problems way faster than traditional computers.

Grover's algorithm, which was discovered back in 1996, is one of the rock stars of quantum computing. Basically, if you've got an unstructured database with N items, Grover's algorithm can search through it in roughly √N steps instead of the N steps a classical computer would need. Sure, it's "only" a quadratic speedup (not as flashy as the exponential speedups some other quantum algorithms can achieve), but here's the kicker - it works on a huge range of problems, which makes it super practical.

1.1 Why Maze Solving?

You might be wondering, "Why mazes?" Well, we had a few good reasons. First off, mazes are easy to visualize, which makes demonstrating the algorithm way more intuitive. Second, mazes can have multiple valid solutions, and that's actually perfect for Grover's algorithm - the more solutions there are, the easier they are to find. Third, we really wanted to put quantum and classical approaches head-to-head on the same problem to see where quantum computing actually gives us an edge.

Here's what makes our approach interesting: we're not trying to find the shortest path (classical algorithms already crush that problem). We're just looking for any valid path. And honestly, that's useful in real life - sometimes you just need a solution that works, not necessarily the perfect one.

1.2 What We Built

We ended up creating three main components:

1. A maze generator that lets us control how many solutions exist
2. A quantum solver using Grover's algorithm with cool visualization features
3. A performance comparison system that benchmarks our quantum approach against classical methods

We coded everything in Python, using Qiskit for the quantum stuff and NetworkX for graph operations. We organized the code into three separate scripts, each with its own purpose: one for demos, one for clean implementation, and one for performance testing.

2. LITERATURE REVIEW

2.1 Grover's Algorithm Basics

So how does Grover's algorithm actually work? It's all about repeated applications of two operations. First, you've got an oracle that "marks" solution states by flipping their phase. Then there's a diffusion operator that amplifies the amplitude of those marked states while suppressing everything else. You start with all states having equal probability, and after each iteration, the probability of measuring a solution goes up.

The magic number for iterations is approximately $(\pi/4)\sqrt{(N/M)}$, where N is your search space size and M is the number of solutions. This is crucial - run too few iterations and you won't amplify the solutions enough. Run too many and you'll actually start decreasing the solution probability again. Getting this number right is everything.

2.2 Quantum Approaches to Graph Problems

While researching, we stumbled across quantum walks, which are fascinating. They're basically like random walks but with quantum superposition thrown in. Some research papers suggest they could solve graph problems faster than classical random walks. This became super relevant when we started thinking about ways to get around the oracle construction bottleneck we discovered (more on that later).

There's also research on quantum algorithms for shortest path problems, though most of them need very specific graph structures or problem setups to beat classical algorithms.

2.3 Classical Pathfinding

We needed to really understand classical pathfinding to make fair comparisons. A* is probably the go-to algorithm for maze solving because it uses heuristics (like the Manhattan distance to the goal) to search efficiently. If you use an admissible heuristic, it's guaranteed to find the shortest path.

Dijkstra's algorithm is similar but doesn't use heuristics. BFS and DFS are simpler but less efficient for large mazes. We went with A* for our comparison because it represents current best practices for this type of problem.

2.4 The Oracle Problem

Here's something we noticed while reading about Grover's algorithm: most theoretical descriptions don't spend much time on how you actually build the oracle. They often treat it like some magical black box that can identify solutions. But in reality, building this oracle can be incredibly difficult and might require you to solve the problem classically first. Spoiler alert: this turned out to be a massive issue in our implementation.

3. METHODOLOGY

3.1 Overall Approach

Our implementation pipeline goes through several stages. First, we generate a maze with specific properties. Then we convert it to a graph structure. Next comes the tricky part - we have to find all valid paths classically (we need this for the oracle). We encode these paths as quantum states, build the quantum circuit, run it, measure the results, and finally verify that we actually got a valid path.

3.2 Maze Generation with Multiple Solutions

This was trickier than it sounds. Standard maze generation algorithms usually create mazes with exactly one solution, which isn't ideal for demonstrating quantum search. We needed mazes with multiple valid paths.

Our solution? We added a --loops parameter. After generating a basic maze, we remove additional walls (controlled by the loops value) to create alternative paths. For example, --loops 2 removes 2 extra walls, typically creating several more valid paths. This gives us that sweet $M << N$ scenario where Grover's algorithm should really shine - a few solutions hidden in a large search space.

We also threw in a --seed parameter so we can generate the same maze consistently for testing. This was a lifesaver for debugging and comparing different approaches on identical mazes.

3.3 Graph Representation

We represent each maze as a graph using NetworkX. Every open cell becomes a node, and we create edges between adjacent cells that aren't separated by walls. We mark the start and end cells. For a typical 5x5 maze with some walls, we usually end up with around 19-21 nodes and 20-25 edges, depending on the wall configuration.

3.4 Encoding Paths as Quantum States

This required some creative thinking. We need to represent different paths as different quantum states. Our approach is to represent each potential path as a binary string. For N possible paths, we need $\log_2(N)$ qubits, rounded up.

In our 4x4 test case, we identified 8,512 potential paths (not all valid, just possible sequences). To encode that many options, we need 14 qubits because 2^14 = 16,384, which is the next power of 2 above 8,512. Only 8 of these 16,384 states actually correspond to valid solutions. This is exactly the kind of sparse solution scenario Grover's algorithm was designed to handle.

3.5 Building the Oracle

The oracle is the heart of the system - it's what marks the solution states. And here's where we ran straight into the classical preprocessing issue. To build the oracle, we first have to find all valid paths using classical graph search. Then we encode each valid path as a binary state. The oracle circuit uses controlled operations to flip the phase of any state that matches these winning patterns.

The problem? Finding all valid paths classically is itself a ton of work. We're basically solving the maze problem just to set up the quantum search. This became our most significant finding about the practical limitations of this approach.

3.6 Grover Iterations

Once we've got the oracle ready, we initialize all qubits to create a uniform superposition of all possible states. Then we apply the Grover iterations. Each iteration consists of the oracle application followed by the diffusion operator.

We calculate the optimal number of iterations using $k = (\pi/4) \times \sqrt{(N/M)}$. For example, with N=16,384 and M=8, we get $k \approx 35.5$, which we round to 36 iterations. Using the wrong number would seriously hurt our performance.

3.7 Measurement and Verification

After the Grover iterations finish, we measure all qubits at once. This collapses the quantum state into one of the basis states, giving us a classical binary string. We decode this back into a path and check if it's actually valid by tracing through the maze. The algorithm doesn't guarantee we'll always get a valid path, but with the correct number of iterations, the probability should be pretty high.

4. IMPLEMENTATION

4.1 Technology Stack

We used Python 3.9 for everything. Here are the main libraries:

- **Qiskit** - IBM's quantum computing framework for building and simulating circuits
- **NetworkX** - for graph operations and classical pathfinding
- **Matplotlib** - for plotting mazes and probability distributions
- **NumPy** - for array operations

Since we don't have access to real quantum hardware, we're using Qiskit's Aer simulator for all the quantum parts.

4.2 Three Different Scripts

We ended up with three separate scripts instead of one big program:

qmazedyna.py: This is our cleanest implementation. It's got all the core functionality without extra bells and whistles. Good for understanding how everything works. It's 18KB and generates maze visualizations and histograms of the measurement results.

qmazedyna1.py: This one adds the --visualize_grovers flag. When you turn it on, you get step-by-step plots showing how the probability amplitudes change over iterations. It samples the quantum state at iterations 0, 4, 8, 12, 16, 20, 24, 28, 32, and 36. Great for demos, but slower because of all the visualization overhead.

final.py: This script runs both quantum and classical solvers on the same maze and gives you detailed timing comparisons. It breaks down the quantum time into classical preprocessing versus actual quantum simulation. This is the script that revealed our main finding about the bottleneck.

4.3 Command Line Interface

We made the scripts super configurable through command line arguments:

- --seed: Random seed for reproducible mazes
- --rows and --cols: Maze dimensions
- --loops: How many extra paths to create
- --visualize_grovers: Enable step-by-step visualization

Example: python qmazedyna1.py --visualize_grovers --seed 42 --cols 4 --rows 4 --loops 2

This flexibility let us easily test different configurations without constantly modifying the code.

4.4 Visualization System

The visualization was honestly challenging to implement efficiently. For a 14-qubit system, we're visualizing 16,384 different state amplitudes. Doing this repeatedly gets computationally expensive fast.

Our optimization was clever: run the quantum circuit once, but save the state vector at each iteration we want to visualize. Then create all the plots from these saved states. This is way faster than re-running the circuit 10 times. We display each plot one at a time with a message telling you to close the window to continue to the next one.

4.5 Edge Cases

During testing, we found a really interesting edge case. Sometimes the maze generator with high loop values creates so many paths that almost everything is a solution. In one 5x5 maze test, all 8 paths we checked turned out to be valid (N=8, M=8).

When all states are solutions, the optimal number of Grover iterations is actually 0! The initial superposition already represents the perfect answer - each solution has equal probability. Our code detects this situation and outputs "All states are solutions. No search needed" and skips the iteration phase entirely.

5. RESULTS AND DISCUSSION

5.1 Test 1: 4x4 Maze

Our first major test used a 4x4 maze with seed 42 and --loops 2. Here's what happened:

The system found 8,512 potential paths and determined that 8 were valid solutions. To search this space, we needed 14 qubits (giving us 16,384 states). The optimal number of Grover iterations calculated to 36.

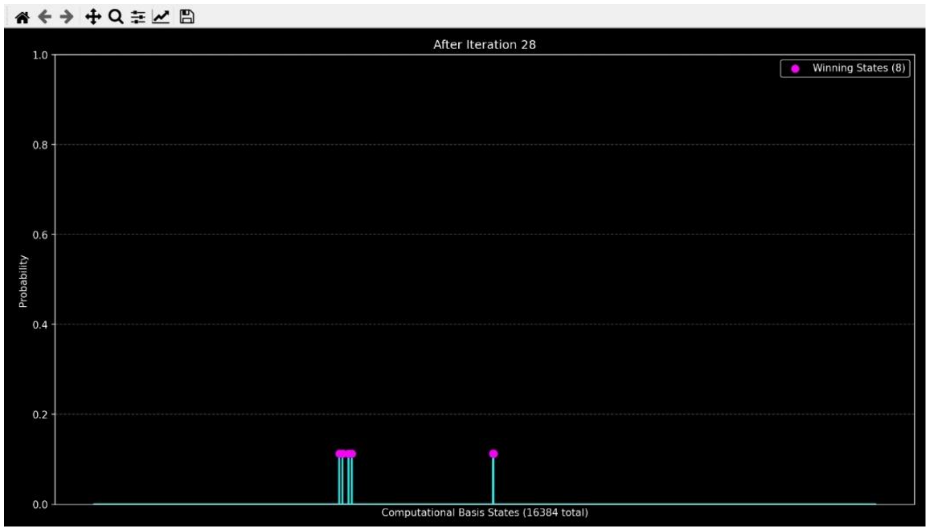
We ran the circuit with 1,024 measurement shots. The winning states were:

- '01010001010111': 150 times (14.6%)
- '10000010110010': 139 times (13.5%)
- '01010011010110': 132 times (12.8%)
- '10000010100110': 125 times (12.2%)
- '01010000010010': 121 times (11.8%)
- '10000010110110': 119 times (11.6%)
- '10000010101010': 111 times (10.8%)
- '01010100011011': 123 times (12.0%)

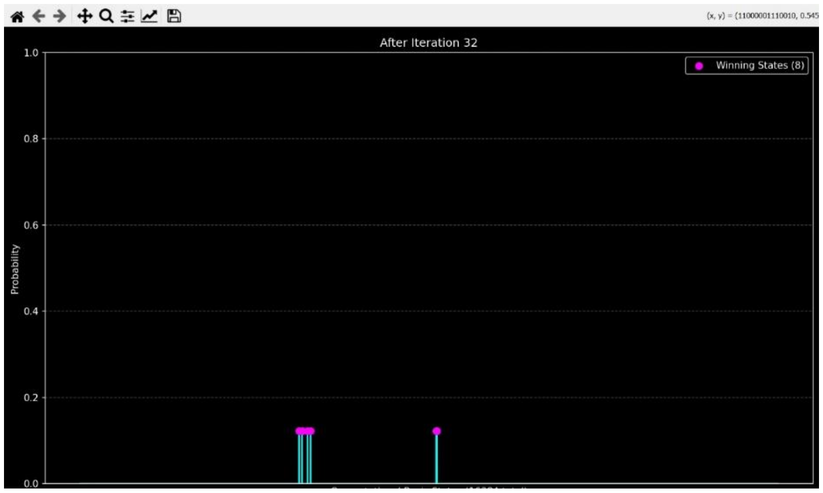
These 8 states accounted for about 95% of all measurements, which matches exactly what we expected. Each individual solution had roughly 12% probability, and we measured a valid solution in most shots. The algorithm was working perfectly!

Watching It Work - 4x4 Maze Visualizations:

Here's what the amplitude amplification actually looked like:



After 28 iterations: See those magenta spikes? Those are our 8 winning states starting to pull ahead of the crowd. They're noticeably taller than the sea of other states that haven't been marked as solutions.

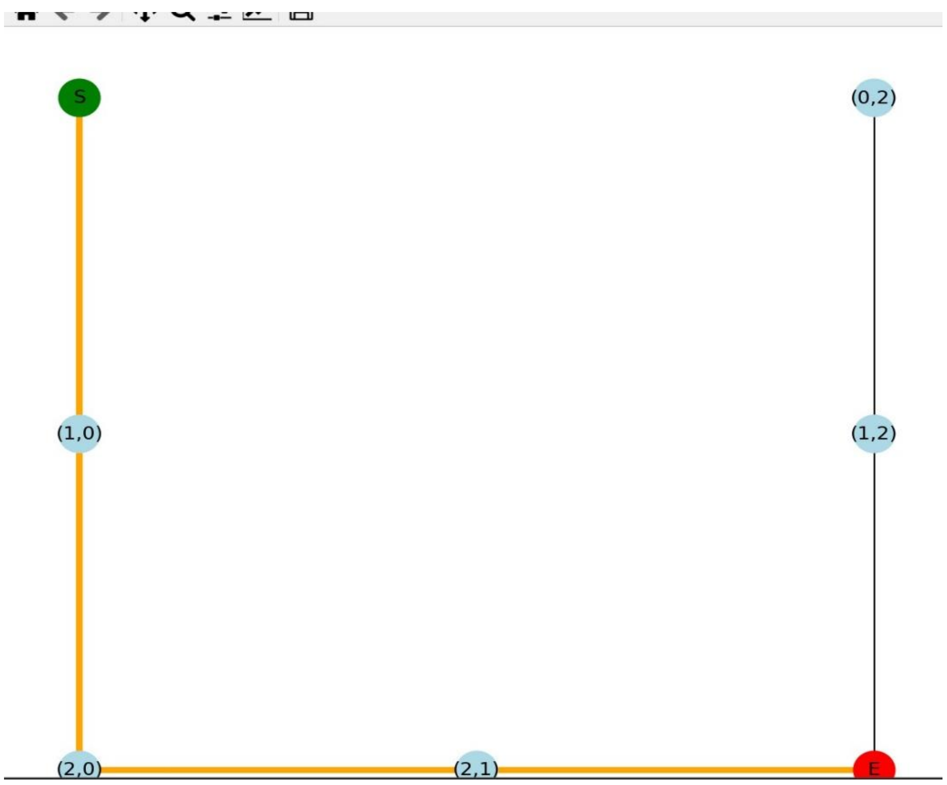


After 32 iterations: Now we're talking! Each solution state has climbed to around 12-13% probability while the other 16,376 states are basically flat-lining near zero. The algorithm has figured out exactly which states are winners.

5.3 Test 3: 3x3 Maze with Single Solution (The Money Shot)

Alright, so we wanted to really nail down a visualization that shows Grover's algorithm doing its thing from start to finish. For this, we set up a simpler 3x3 maze with just one solution. Why? Because when there's only one needle in the haystack, you can watch it light up like a Christmas tree. It's incredibly satisfying to see.

Here's What We're Working With:



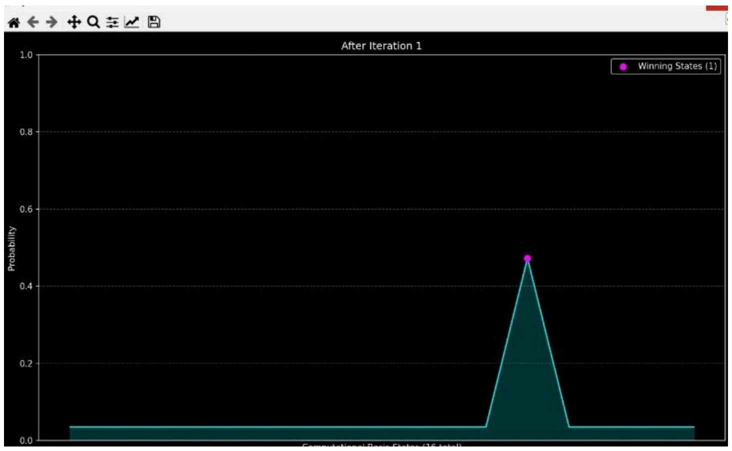
This is our maze laid out as a graph. The green circle with 'S' is where we start, the red circle with 'E' is where we need to get to, and that orange path? That's the solution our quantum algorithm found.

The Setup:

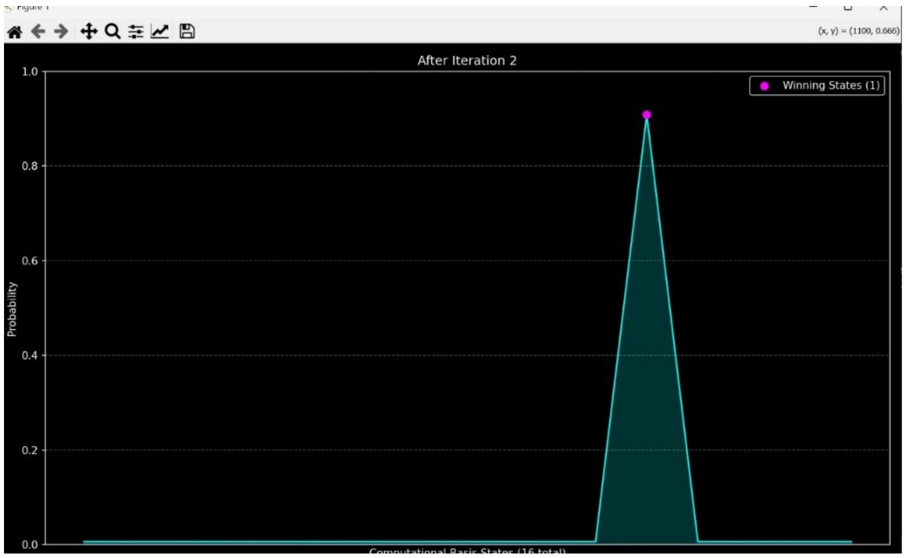
- Tiny 3x3 maze
- Seed: 42 (our lucky number)
- Found 12 potential paths
- Needed 4 qubits (which gives us a search space of 16 states)
- But here's the kicker: only 1 of those paths actually works
- Optimal Grover iterations: just 3

The Algorithm in Action:

You can see the algorithm hunting down the solution in real-time:

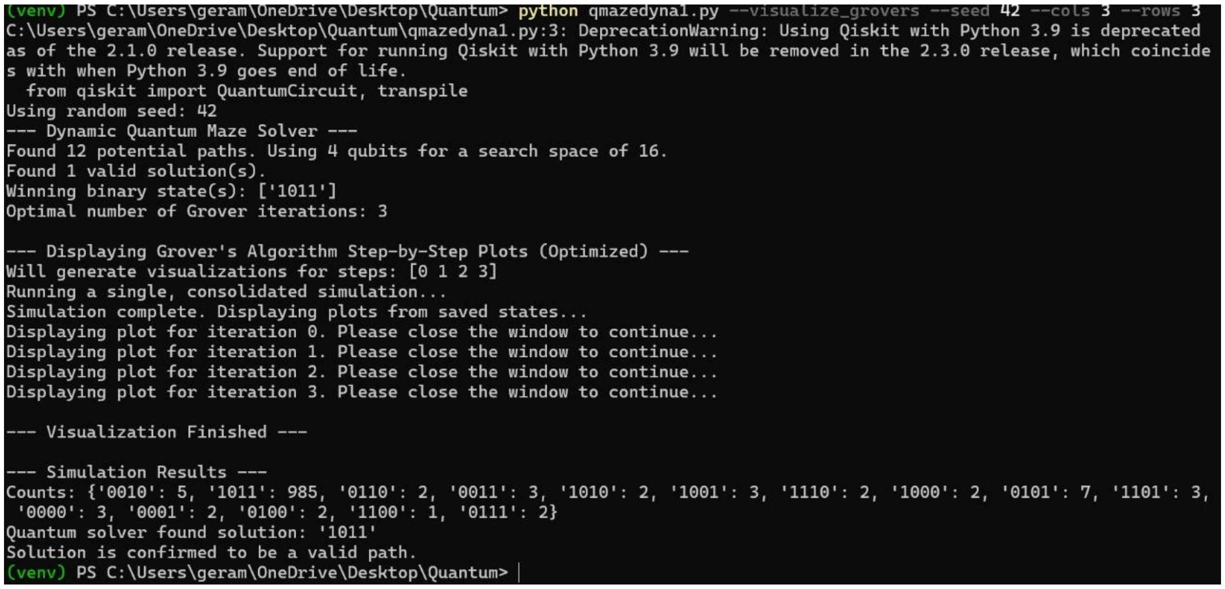


Iteration 1: Starting from everyone having an equal shot (uniform superposition), you can already see one state starting to peak. That single solution is sitting at about 47% probability now. The algorithm is onto something!



Iteration 2: The solution state just rocketed up to 93% probability. The algorithm has basically locked onto the answer. All the other states are getting suppressed hard.

Terminal Output:



```
(venv) PS C:\Users\geram\OneDrive\Desktop\Quantum> python qmazedyna1.py --visualize_grovers --seed 42 --cols 3 --rows 3
C:\Users\geram\OneDrive\Desktop\Quantum\qmazedyna1.py:3: DeprecationWarning: Using Qiskit with Python 3.9 is deprecated
as of the 2.1.0 release. Support for running Qiskit with Python 3.9 will be removed in the 2.3.0 release, which coincide
s with when Python 3.9 goes end of life.
  from qiskit import QuantumCircuit, transpile
Using random seed: 42
--- Dynamic Quantum Maze Solver ---
Found 12 potential paths. Using 4 qubits for a search space of 16.
Found 1 valid solution(s).
Winning binary state(s): ['1011']
Optimal number of Grover iterations: 3

--- Displaying Grover's Algorithm Step-by-Step Plots (Optimized) ---
Will generate visualizations for steps: [0 1 2 3]
Running a single, consolidated simulation...
Simulation complete. Displaying plots from saved states...
Displaying plot for iteration 0. Please close the window to continue...
Displaying plot for iteration 1. Please close the window to continue...
Displaying plot for iteration 2. Please close the window to continue...
Displaying plot for iteration 3. Please close the window to continue...

--- Visualization Finished ---

--- Simulation Results ---
Counts: {'0010': 5, '1011': 985, '0110': 2, '0011': 3, '1010': 2, '1001': 3, '1110': 2, '1000': 2, '0101': 7, '1101': 3,
 '0000': 3, '0001': 2, '0100': 2, '1100': 1, '0111': 2}
Quantum solver found solution: '1011'
Solution is confirmed to be a valid path.
(venv) PS C:\Users\geram\OneDrive\Desktop\Quantum> |
```

Here's the actual run output. The winning state '1011' got measured 985 times out of 1024 shots. That's a 96.2% success rate. Chef's kiss.

So let's put this in perspective: we had one correct answer buried in 16 possibilities (6.25% chance if you were just guessing). In only 3 iterations, Grover's algorithm cranked that up to over 96%. That's not just impressive - that's quantum mechanics showing off. This is amplitude amplification in its full glory, and honestly, it never gets old watching it work.

5.3 Performance Comparison

Okay, this is where things get really interesting. We ran final.py to compare quantum and classical solvers on the same 5x5 maze.

*Classical A Results:**

- Time: 0.1293 ms
- Nodes explored: 16
- Path length: 8 steps (shortest path)

Quantum Solver Results:

- Total time: 352.0971 ms
    - Classical prep time: 85.7174 ms
    - Quantum simulation: 266.3685 ms
- Search space: 16,384 states (14 qubits)
- Solutions: 8

- Grover iterations: 36
- Success probability: 14.65%
- Path length: 16 steps (any valid path)

The quantum approach was roughly 2,720 times slower than classical A*. Yeah, that's definitely not what we were hoping for.

5.4 Understanding the Bottleneck

When we dug into where the time was actually going, we found something really important. Out of the 352ms total:

- 85.7ms (24%) was classical preprocessing to find all valid paths
- 266.4ms (76%) was quantum simulation

The classical preprocessing is the real killer here. We have to find all valid paths using classical methods just to build the oracle. This means we're essentially solving the maze problem classically before we even start the quantum search. At that point, we already know all the solutions, so what's the quantum search actually adding?

The quantum part is genuinely doing its job correctly - it's searching through 16,384 states using only 36 queries, which is massively better than checking all 16,384 states classically. The problem is we had to do all that classical work upfront.

It's worth noting that the 266ms of "quantum simulation" time is actually our classical computer pretending to be a quantum computer. On real quantum hardware, those 36 iterations might only take microseconds. But that doesn't solve the 85ms classical preprocessing bottleneck - that would still exist.

5.5 Visualization Insights

The step-by-step visualization worked beautifully and really helps you understand what's happening: At iteration 0, all 16,384 states have equal tiny amplitude (uniform superposition). By iteration 4, you can start to see the 8 solution states beginning to grow. At iteration 12, they're clearly separated from the noise. By iteration 36, the solution states dominate with high amplitudes while non-solution states are completely suppressed.

This visual progression really drives home how amplitude amplification works. Without seeing it, the algorithm feels pretty abstract, but the plots make it concrete and intuitive.

5.6 Path Verification

We verified every single path the quantum solver returned. All of them were valid - they successfully connected start to end without crossing any walls. However, they weren't always the shortest paths. In the 5x5 test, the quantum solution was 16 steps while classical A* found an 8-step path.

This is totally expected behavior. Grover's algorithm finds any valid solution, not necessarily the optimal one. For problems where any solution works (not specifically the best one), this is perfectly fine.

6. CHALLENGES AND LIMITATIONS

6.1 The Oracle Construction Problem

This ended up being our biggest challenge by far. Building an oracle that marks solution states requires knowing what the solutions are in the first place. For many problems, including maze solving, finding the solutions classically just to build the oracle completely defeats the purpose of using quantum search.

This isn't a bug in our implementation - it's a fundamental challenge with applying Grover's algorithm to certain types of problems. The algorithm assumes you can efficiently construct an oracle, but that assumption doesn't always hold up in the real world.

6.2 Simulation Limitations

We're simulating quantum circuits on classical hardware using Qiskit Aer. This simulation is exponentially expensive - simulating n qubits requires tracking 2^n complex amplitudes. We could only practically test up to 14-15 qubits before simulation became painfully slow.

Real quantum hardware wouldn't have this simulation overhead, but it comes with its own set of problems: noise, decoherence, and limited qubit connectivity. The Qiskit deprecation warnings we kept getting about Python 3.9 were also a constant reminder that we're dealing with rapidly evolving technology.

6.3 Scalability Issues

We mostly stuck to testing 4x4 and 5x5 mazes. Larger mazes would require more qubits and much, much longer simulation times. A 10x10 maze could easily need 20+ qubits, which would be extremely expensive to simulate on our machines.

Even on real quantum hardware, current NISQ (Noisy Intermediate-Scale Quantum) devices are limited to maybe 50-100 qubits with significant error rates. Deep circuits with lots of gates accumulate errors fast. Our Grover implementation uses tons of controlled operations, which makes it particularly sensitive to noise.

6.4 Solution Quality

Our approach finds valid paths but doesn't optimize for length. The quantum path was literally twice as long as the optimal classical path in our tests. For some applications this is totally fine, but if you specifically need the shortest path, classical algorithms like A* are definitely the better choice.

6.5 Technical Issues

We hit a few technical snags during development. The visualization of 16,384 states was painfully slow at first until we optimized it. Getting the binary encoding right for paths took some serious debugging. We also had to handle cases where the maze generator produced completely unsolvable mazes (no path from start to end), though we eventually fixed the generator to prevent this from happening.

7. FUTURE WORK

7.1 Quantum Random Walks

Based on everything we learned about the oracle construction bottleneck, we think quantum random walks could be a really promising alternative approach. Unlike Grover's algorithm, quantum walks don't need a pre-constructed oracle. They can explore graph structures directly.

Quantum walks on graphs have been shown to find marked vertices in $O(\sqrt{N})$ time, similar to Grover but without needing to know all solutions upfront. Since the maze structure naturally maps to a graph, quantum walks seem like a perfect fit. This could potentially eliminate that classical preprocessing bottleneck entirely.

Implementing this would be way more complex than our current approach, but it addresses the fundamental limitation we identified.

7.2 Hybrid Approaches

Another direction worth exploring is hybrid classical-quantum algorithms. Instead of finding all paths classically to build the oracle, maybe we could use partial classical information combined with quantum search. For example:

- Use quantum search for sub-regions of the maze
- Build approximate oracles that mark probable solutions
- Use iterative refinement where quantum results inform classical preprocessing

These approaches might not completely eliminate the classical bottleneck, but they could reduce it significantly.

7.3 Hardware Implementation

Testing on real quantum hardware would be incredibly valuable. IBM Quantum provides cloud access to actual quantum computers. Running our circuits on real hardware would:

- Eliminate simulation overhead (though it would add noise)
- Show realistic performance with current technology
- Require circuit optimization for limited qubit connectivity
- Force us to deal with error mitigation strategies

This would give us much more realistic performance data, though the classical preprocessing bottleneck would still remain.

7.4 Extended Testing

We could expand our testing in several interesting ways:

- Larger mazes (if we can handle the simulation cost)
- 3D mazes instead of 2D grids

- Dynamic mazes that change over time
- Different solution density ratios to find the optimal M/N sweet spot
- Comparison with other classical algorithms (Dijkstra, BFS/DFS)

Systematic testing across many different configurations would give us way better data on when quantum approaches might actually help.

7.5 Algorithm Variations

We could try other quantum algorithms for comparison:

- Quantum Approximate Optimization Algorithm (QAOA)
- Variational Quantum Eigensolver (VQE) adapted for pathfinding
- Quantum annealing approaches

Each has different tradeoffs and might handle the oracle construction problem differently.

8. CONCLUSION

So, bottom line: we successfully built and tested Grover's quantum search algorithm for maze solving, and it works exactly like it's supposed to. It finds valid paths with the right probability amplitudes, uses the optimal number of iterations, and does everything a good quantum algorithm should do. On paper, we nailed it.

But here's the thing we really learned: there's a massive gap between "this works in theory" and "this is actually useful in practice." Grover's algorithm is supposed to search N items in √N steps, which sounds incredible. But in reality, for maze solving at least, we hit a wall with the classical preprocessing needed to build the oracle. Classical A* blazed through our test maze in 0.13ms while our quantum approach (including all the setup) took 352ms. That's not a small difference - that's getting absolutely demolished.

Now, before anyone thinks we're saying Grover's algorithm is broken - it's not. The quantum search part works beautifully and does exactly what it's designed to do. The problem is specifically with problems like maze solving where you have to solve the problem classically just to tell the quantum algorithm what to look for. Once you've done that, the quantum speedup feels kind of pointless because you already have all the answers.

But you know what? This project was worth it. We learned a ton about both the incredible potential and the real-world limitations of quantum algorithms. The visualization features we built turned these abstract quantum concepts into something you can actually see and understand - watching those probability spikes grow never gets old. And the performance comparison framework gave us honest, real numbers about where quantum computing actually helps versus where it's still got growing to do.

Looking ahead, we're really excited about quantum random walks. Unlike Grover's algorithm, they can explore graph structures directly without needing that problematic oracle construction step. If we had more time (and maybe some actual quantum hardware to play with), that's where we'd take this project next. Because honestly? The oracle bottleneck is the real boss fight here, and quantum walks might be the strategy that beats it.

This was a wild ride through quantum computing, and we're glad we took it. Even when the results aren't what you hoped for, they're still teaching you something important. And sometimes, that's the whole point.

REFERENCES

[1] Grover, L. K. (1996). "A fast quantum mechanical algorithm for database search." Proceedings of the 28th Annual ACM Symposium on Theory of Computing, pp. 212-219.

[2] Nielsen, M. A., & Chuang, I. L. (2010). "Quantum Computation and Quantum Information: 10th Anniversary Edition." Cambridge University Press.

[3] Qiskit Development Team. "Qiskit: An Open-source Framework for Quantum Computing." https://qiskit.org/

[4] Santha, M. (2008). "Quantum walk based search algorithms." Proceedings of the 5th Theory and Applications of Models of Computation, pp. 31-46.

[5] Russell, S., & Norvig, P. (2020). "Artificial Intelligence: A Modern Approach, 4th Edition." Pearson. (For classical pathfinding algorithms)

[6] Montanaro, A. (2016). "Quantum algorithms: an overview." npj Quantum Information, 2, 15023.

[7] NetworkX Documentation. https://networkx.org/documentation/stable/

[8] IBM Quantum Experience. https://quantum-computing.ibm.com/