quWaveFunctionCollapse

1 Project Description

We proposed our own project, based on the WaveFunctionCollapse algorithm, which can be seen here. The original algorithm is a procedural generation algorithm to generate bitmaps that are locally similar to the input bitmap. The general idea is that based on this input bitmap, we are given a set of tiles with neighboring constraints as input, where we want to randomly generate a larger board of a given size that satisfies the neighboring constraints from this. The original algorithm is based on quantum mechanics, where it initializes all tiles in a superposition of all states in the input and then observes the tile with the lowest entropy to collapse it. Following this, the changes caused by the collapse are propagated through the rest of the board until all constraints are satisfied. This repeats until all tiles are collapsed into a single state, in which we have the board we wanted to generate.

Given this clear underlying quantum structure to the problem, our main goal is to incorporate our knowledge of quantum programming and algorithms from this course into the project, replacing certain parts of the algorithm with quantum ideas rather than this classical simulation of it. We aimed to do so in two different ways.

The first was to keep the mainly classical approach to the algorithm and analyze the effect of utilizing a quantum random number generator to select the state for collapse. In the original algorithm, this state is selected using the numpy random.choice function. However, we were interested in seeing what difference, if any, it would make to adjust this to use a quantum random number generator. Furthermore, this was the first step in allowing us to begin incorporating more quantum ideas into the algorithm. To implement this, we did so in Q#, where we generated a single random bit by applying the Hadamard gate to a qubit in the zero state. This causes the qubit to be in an equal superposition, such that upon measuring in the computational basis, we obtain a completely random result. Thus, we then obtained a random bit string by concatenating many of these random bits, which we converted into an integer to obtain our final random integer to pass to the classical program. Furthermore, the classical wrapper itself is largely a Python version of the original algorithm, which we will discuss in the following section.

Our second method for incorporating quantum computation into the WaveFunctionCollapse algorithm was through an algorithm similar to VQE from lecture. From our input image, we obtain frequencies for each of the tiles as well as a table of constraints on neighboring tiles. We then pass these frequencies into the quantum circuit, where the states are then encoded as one-hot vectors with each tile in superposition with the respective probabilities of being in each state as the amplitudes. Furthermore, by using a variational approach, we can then arrive at the correct entanglement scheme in order to obtain a board where the constraints are satisfied. This involves a quantum and classical hybrid algorithm very similar to VQE. Our idea for this is to have a parameterized quantum circuit, which strongly entangles the tiles and counts the number of conflicts with the constraints found earlier. In addition, this quantum circuit only deals with a section of the board at a time, given a center tile, the one to its right, and the one below it, when thinking about the arrangement of the board. Thus, we start from the top left tile and progressively apply the quantum circuit and count the number of conflicts for every possible center tile. Furthermore, we can pass this count to the classical machine in which we have a loss function that sums the total number of conflicts for the whole board. Our classical minimizer then selects new frequencies for the tiles, where this choice should minimize the number of conflicts that occur. Finally, when this minimizer converges on the board with the minimal number of conflicts with the constraints, we have the board we want. Classically, we select the tile states with the highest probability, since the minimizer had the most success when those states occurred most often.

2 Implementation

We had 2 options for quantum implementations. One simple implementation was to replace the classical random number generation in the classical propagation algorithm with a quantum random number generator. The other, much more complicated algorithm, was to attempt a VQE approach to generating a board which mostly satisfies the constraints.

First, we will detail the quantum random number generation (QRNG) portion of the algorithm and how it fits in to the original classical WaveFunctionCollapse algorithm. The logic behind this implementation is that we replace the numpy random.choice function, which chose a random initial position and random state to collapse with our quantum random number generator, with a quantum random number generator. In this method, we leave most of the original WaveFunctionCollapse algorithm as classical but then add in this small quantum element to see how it affects the runtime or results. The implementation of the random number generator is as follows. We use the Hadamard gate to put a qubit in an equal superposition and then measure this in the computational basis. Because the $|+\rangle$ state is an equal superposition of the computational basis states, measuring it will result in a perfectly random bit. Furthermore, we then loop to create a bit string made from concatenating these random bits together and convert this into an integer. Thus, we have obtained a random integer using quantum random number generation. Furthermore, we can assert an upper bound on the random number generated by only creating bit strings of a certain length and repeating this process until arriving at a suitable integer. Using this quantum technique for generating random integers, we then choose states in the wave function that we want to collapse as well as our initial state using this random number generator rather than the classical numpy version.

Now, we will discuss the variational approach to incorporating quantum computation into the WaveFunctionCollapse algorithm. As discussed earlier, the general idea for this implementation is to adapt the classical WaveFunctionCollapse to ensure there are as few conflicting tiles as possible utilizing entanglement of qubits which represent these tiles. The logic behind this implementation is that from the input image, we classically obtain a table of probabilities as well as a table of constraints for each of the neighboring tiles. From the table of probabilities, we have a vector of probabilities for each tile, where the probabilities are the chance that this tile is in a specific state. Each state in the board is represented by a one-hot vector. Thus, when we encode the tiles as a superposition of these states in the quantum machine, the amplitudes of the superposition equal to the square roots of the probabilities of that tile being in a specific state. Note that these should be square roots since in quantum mechanics, we have that the probability of being in a certain state should be the amplitude of that state in the superposition squared. Specifically, this encoding was created similarly to the W state, where we used controlled rotations as well as trigonometry to get the correct amplitudes. Furthermore, we use a table of values which stores intermediate values that we have as our amplitudes in order to make the computation more efficient. To explain this method more thoroughly, we will go through one step in the algorithm for the test given of $x = [0.4, 0.1, 0.2, 0.3]$. Then, we want a superposition with amplitudes $0.5477, 0.447, 0.316, 0.632$ (approximately) as these are the square roots of the x values given. Our algorithm takes in these square root values and then computes the angle needed such that the rotation results in such an amplitude on the $|1\rangle$ state. This can be accomplished with the y rotation gate, which has the effect

$$R_y(\theta)|0\rangle = \cos\frac{\theta}{2}|0\rangle + \sin\frac{\theta}{2}|1\rangle$$

Thus, we can clearly see that in order to get the amplitude, 0.5477 for example, we take $\theta_1 = 2\arcsin(0.5477)$. Then, when we apply the rotation gate to the first qubit, we get the state

$$0.5477|1000\rangle + \sqrt{1 - 0.5477^2}|0000\rangle$$

Then, the next angle we have to rotate by is $\theta_2 = 2\arcsin\frac{0.447}{\sqrt{1-0.5477^2}}$. This denominator will get rid of the left over amplitude we have on the $|0000\rangle$ state. Thus, we continue in this way and store each of the leftover amplitudes we have on the $|0000\rangle$ state after each iteration to avoid recomputing each time. In addition, to avoid changing the states we already have with correct amplitudes, we control this rotation on the first qubits (however many are before the current basis state we want to fix the amplitude of) being in the zero state. In this way, we encode the probabilities into the amplitudes of the superposition.

In terms of the variational form used, we construct a parametrized circuit that counts the number of

conflicting states for a board state under the neighboring constraints derived from the WaveFunctionCollapse classical algorithm. To achieve this minimizer approach, the encoded qubit arrays, prepared as described earlier, are then entangled in accordance with an input array of acceptable neighboring states. Thus, exhaustively looping over all pairs of states, we only entangle qubits corresponding to states that conflict in pattern. Note that while the input array shows all acceptable neighboring states for every center tile, we only need to check the right and bottom states. To determine if two tiles conflict, we entangle the conflicting qubits in the center and right or bottom qubit arrays with one of two reference qubits (one for checking with the right tile and the other for checking with the bottom tile) at the from of the center qubit array using the CCNOT gate. Measuring the two reference qubits then allows us to determine whether or not a conflict has occurred since there can only be maximally one conflicting pair due to one-hot encoding. However, by only checking the right and bottom tiles for each center tile, we gain additional edge cases to care for. Specifically, for tiles on the bottom or rightmost edge of the output space, the missing tile will not pose any conflicts and thus is accounted for in the final Boolean array of conflict counts passed back to the classical loss function.

In the loss function, we iterate through all possible center tiles in the board and pass the center, right, and bottom tiles to the quantum variational circuit to get the counts of the number of conflicts for this specific center tile. Then, the loss function sums up the total number of conflicts for the entire board in this way. Furthermore, when we generate the actual board, our classical minimizer then selects the probabilities for each tile that minimize the expected number of conflicts. Then, we measure the board with these probabilities and generate the output image. Thus, in summary for the variational approach, we are using the quantum programming language to enforce the constraints on neighboring tiles through entanglement where we then use Python to minimize the number of conflicts with these constraints to arrive at an optimal board.

Finally, we will describe generally how the classical algorithm works. The classical code for all ways is essentially the same, but with the addition of the loss function for our variational method. First, we take in an image and break it up into the different tiles, which are then passed to the WaveFunctionCollapse algorithm. From this image, we also retrieve the neighboring constraints, which are also passed to the WaveFunctionCollapse algorithm. Then, we initialize the tiles for the output image each into a superposition of all states in the input and procedurally collapse each tile into a single state. Specifically, we perform a measurement on the tile with the lowest entropy to collapse it. Then, we can propagate the changes through the entire board until no changes are left. Next, we select the next tile of lowest entropy to collapse, and repeat. At the end of this loop, we are left with the board which satisfies all of our neighboring constraints.

3 Results

First, we will display the results comparing the quantum random number generator against the completely classical WaveFunctionCollapse. After testing on several input images, a sample of the results we obtained are below. We only included the Red Maze and Bricks input images as tests for the sake of space in this report. First, in Figure 1, the original input patterns for the Red Maze and Brick patterns are shown. Following this, in Figures 2 and 3, we display different results from the completely classical WaveFunctionCollapse algorithm implemented in Python versus the WaveFunctionCollapse algorithm with quantum random number generation included when we choose which state we want to collapse. Ideally, we would use a higher pattern dimension of 3×3, but the quantum algorithm took too long to run with this (surprisingly vastly) increased number of parameters. Thus, we stuck with a dimension of 2. Although the generated images are not as "nice" it is still valid for discussion.

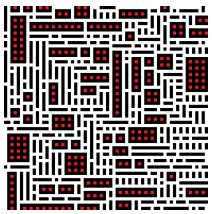
Obviously, these results will be slightly different due to the randomness in each of them, but the result overall is still what we expected. Both of these output images are locally similar to their respective inputs. Furthermore, note that for the results obtained here, the QRNG method took longer to run than the purely classical method. This was expected, as simulating quantum computer to generate the random numbers which we then have to pass to the classical machine would clearly increase the runtime, rather than dealing with only the classical computer.

The VQE approach took so long to run that the algorithm would not finish by the deadline for this report. It is disappointing that we could not see if it was able to converge correctly, but at least we have an algorithm that makes use of non-classical computation to compute this algorithm.

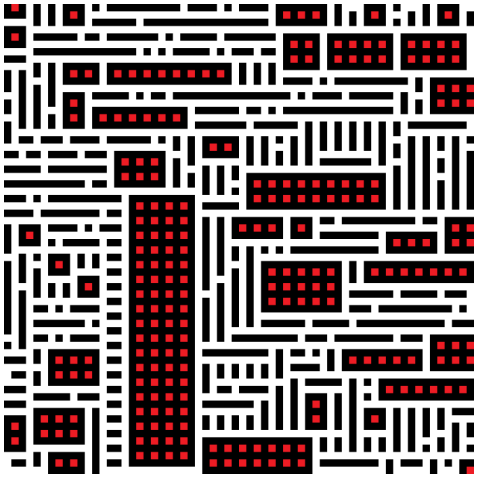
(a) Red Maze Original Pattern

(b) Bricks Original Pattern

Figure 1: Original Input Patterns



(a) Wave Function Collapse with QRNG



(b) Classical Wave Function Collapse

Figure 2: Comparison of Classical and QRNG Wave Function Collapse for Red Maze Pattern

(a) Wave Function Collapse with QRNG

(b) Classical Wave Function Collapse

Figure 3: Comparison of Classical and QRNG Wave Function Collapse for Brick Pattern

4 Tests

The tests we considered were based off of input images provided in the original implementation of the WaveFunctionCollapse algorithm. Given these input images, we can output what our board generated and observe whether or not this output image is locally similar to the original input. The input images are small and indicate which pattern we expect to see in the much larger output image. We considered in total eight input images to test this on. These tests can be used to test all of the classical, quantum random number generation, and variational approaches. The output of the tests should simply be a larger image that is locally similar to the input image, where by locally similar, colloquially we mean that the output image should depict a pattern that resembles the input.

In addition to these overall tests for our implementation, we also have some smaller tests that demonstrate the program is working properly. For instance, the testEnc() operation in the variational method outputs the result of encoding our probabilities into a superposition. In this operation, we have a vector of classical probabilities, given by x and our encodeState operation is meant to make the square root of these classical probabilities the amplitudes of the superposition. Furthermore, the superposition is only over basis states which have Hamming weight 1, similar to the W state, but here it is an unequal superposition. Also, note that the amplitudes should be the square root of the classical probabilties since we operate in the L_2 norm for quantum mechanics.

In the example provided in the test, we have $x = [0.4, 0.1, 0.2, 0.3]$. When we run this test, the result is a simple DumpRegister call which displays the superposition that our qubits are in. Thus, this should be in the superposition given by $|1\rangle$ with amplitude .547722557505166, $|2\rangle$ with amplitude 0.4472135954999579, $|4\rangle$ with amplitude 0.31622776601683794, and $|8\rangle$ with amplitude 0.6324555320336759, with all other amplitudes equal to zero. Note that these are indeed the square roots of the amplitudes given and the basis states have Hamming weight 1. Also, a note is that the superposition produced may have a small (on the order of 10^{-8} amplitude in the $|0>$ state, but this is expected due to rounding errors and other such numerical precision issues. For further tests, this x vector can be replaced with other decimals, under the condition that the entries of x add up to 1, as expected of classical probabilities.

We also include a test for the variational circuit. We define two states A and B, whose only constraints are that they cannot be next to another tile of the same state. Thus, any A tile can only be neighbors with a B tile and vice versa. We tried 3 different test cases. An A surrounded with very high probability by A's, an A surrounded with very high probability by B's, and a perfect superposition of A's and B's. The results are not entirely deterministic since there is a probability, in the first case for example, that the A becomes next to a B. However, we observe that our variational circuit generally returns 1 or two conflicts for the first case and 0 conflicts for the second case. In the third case, our encoding ran into issues, which we did not have time to investigate before the deadline.