1 Add QAOA Functionality to PennyLane

1.1 Status

1.2 Context

The Quantum Approximate Optimization Algorithm is a variational quantum algorithm for solving combinatorial optimization problems, introduced by Farhi, Gutmann and Goldstone in 2014 [1].

To begin, we consider a cost function of the form:

$$C(z) \;=\; \sum_{n} C_n(z) \tag{1}$$

where z is some bitstring, and each $C_n(z)$ is a *clause*, which is a function that maps each bitstring to either 0 or 1. Our goal is to minimize this cost function by finding the optimal bitstring(s) that return the lowest value.

To solve this problem with QAOA, we first map the cost function to a diagonal cost Hamiltonian, where the expectated energy with respect to each basis state is its corresponding cost function value when expressed as a bitstring (for instance, $|000\rangle$ corresponds to the bitstring 000).

Next, we use a variational ansatz to prepare a parametrized quantum state. Beginning with some initial state, $|\psi_0\rangle$ (usually an even superposition over all basis states), we evolve this state to get $|\psi(\gamma, \alpha)\rangle$ with an "alternating operator ansatz", similar to a Trotterized version of adiabatic quantum evolution:

$$|\psi(\gamma, \alpha)\rangle \;=\; \prod_{n\,=\,1}^{P} e^{-i\alpha_n H_M} e^{-i\gamma_n H_C} |\psi_0\rangle$$

where H_C is the cost Hamiltonian that we are attemtping to minimize, and H_M is a non-commuting mixer Hamiltonian.

Upon execution of this circuit, we calculate the quantity $F(\gamma, \alpha) \;=\; \langle\psi(\gamma, \alpha)|H_C|\psi(\gamma, \alpha)\rangle$. We repeat this process many times, minimizing the value of $F(\gamma, \alpha)$. This in turn minimizes the cost function and gives us an optimal circuit that produces the optimal bitstring with high probability. Finally, the optimal circuit is executed multiple times and the resulting bitstring that yields the lowest value of the cost function is outputted.

QAOA has the potential to generate approximate solutions to many well-known and important optimization problems such as QUBO, Max-k-SAT, Machine Scheduling, and more. The possibility of interesting applications, combined with a variational ansatz that is relatively easy to implement, compared to other instances of VQE, makes QAOA a very attractive candidate for implementation on NISQ devices, and possibly even a demonstration of quantum supremacy.

In recent years, quantum researchers have become increasingly interested in performing benchmarking and simulations of different versions of QAOA on classical computers [2-4], making the addition of a QAOA module to PennyLane a potentiallly valuable tool for this kind of work. In addition, QAOA has received a lot of attention from the community of quantum enthusiasts. A PennyLane QAOA module would be a fantastic learning resource for many people trying to learn more about variational quantum algorithms, and may serve to attract more people to the entire PennyLane library.

1.3 Analysis

There are wour six components that are necessary to defining a QAOA workflow:

1. Specifying the optimization problem
2. Allowing the user to define a state initialization circuit
3. Allowing the user to define a mixer layer
4. Creating a cost layer from the chosen optimization problem and combining it with the mixer and an initialization circuit to create a variational circuit

5. Optimization of the variational circuit

6. Post-processing

The objective of this ADR will disucss each in detail, and outline how they will be implemented in a PennyLane QAOA module.

The QAOA module should be as **problem-centric** as possible. This means that the user should be able to define a QAOA workflow for some optimization problem without worrying too much about the exact specifics of the quantum circuits in the backend.

1.4 Location in PennyLane

We propose the creation of a new QAOA module where all of the functionality for creating QAOA workflows would be located.

1.5 Specification of the Optimization Problem

The main component of any QAOA procedure is the cost function that the user chooses to optimize. In the QAOA module, we propose that the user have access to a variety of built-in QAOA problem instances. To start, we suggest that the following optimization problems be implemented:

- MaxCut (for a user-defined graph) For this problem, we know that the cost Hamiltonian is of the form:

$$H_C \ = \ \sum_{(i,j)\in G} \frac{1}{2}\big(1 \ - \ Z_i Z_j\big) \tag{2}$$

- QUBO (for a user-defined Ising model)

After the core functionality has been built, more common optimization problems can be added to the library.

Each of these built-in optimization problems will be instances of this "Problem" class with the following attributes:

- The cost Hamiltonian, implemented as an instance of the pennylane Hamiltonian class.
- The cost layer, which can be constructed by simply exponentiating each term of the cost Hamiltonian. Converting the cost Hamiltonian into a differentiable cost unitary is thus very straightforward, and does not require any complicated decomposition.
- A *recommended* mixer layer, which is a function containing PennyLane operations. The default is simple RX layer used in the original QAOA paper.
- A *recommended* initialization circuit, which is also a function containg PennyLane operations. Generally, this will either be an even superposition over all bitstrings, or an even superposition over all bitstrings satisfying a certain constraint (for example, Hamming weight 1). The default value is a method creating an even superposition.

Note: The user isn't required to use either the recommended mixer, nor the initialization circuit, they are merely suggestions as to what works best for each built-in problem

Note: The custom cost Hamiltonian will also be defined as an instance of the of this general problem class, with the default values for the mixer and initialization attributes.

This is also one of the cases where the circuit decomposition of the cost layer corresponding to a cost Hamiltonian is unknown. Thus, we must write a subroutine decomposes anc arbitrary diagonal Hamiltonian into a sum of Pauli-Z gates. Since the Pauli gates form an orthogonal basis of the vector space of linear transformations, we will have a unique decomposition of our Hamiltonian H_C in terms of Z gates. In fact, we will have:

$$H_C \ = \ \sum_n c_n Z_1^{n_1} \otimes Z_2^{n_2} \otimes \ ... \ \otimes Z_k^{n_k}$$

with $n_j \in \{0,\ 1\}$. It follows that:

$$c_n \ = \ \text{Tr}(H_C Z_1^{n_1} \otimes Z_2^{n_2} \otimes \ ... \ \otimes Z_k^{n_k})$$

This means that with knowledge of H_C, its full decomposition in terms of powers of Z gates can be calculated. Since each term of the Hamiltonian commutes, we will have as the cost unitary:

$$U_C \ = \ \prod_n e^{-i\gamma c_n Z_1^{n_1} \otimes Z_2^{n_2} \otimes \ ... \ \otimes Z_k^{n_k}}$$

Each exponentiated Z in this unitary can be implemented as a MultiRZ gate in PennyLane, and is thus differentiable.

1.5.1 Further Notes on Mixer Layers

There are a wide variety of mixers that can be used for QAOA. A good mixer Hamiltonian should be non-commuting with the cost and constrain the search space of different bitstrings to those that satisfy a certain *hard-constraint* (for instance, Hamming weight 1). As a result, the correct choice of a mixer can vary greatly between different QAOA workflows.

Since the user is not constrained to using the recommended mixer that is built-into each problem instance, we must specify convenient ways for a user to utilize a variety of different mixers in their QAOA workflows.

Similar to defining a cost Hamiltonian, we let the user either:

- Define their own method containing PennyLane operations, which can be used as the mixer layer.
- Choose from a variety of built-in mixers. To begin, we propose the following mixer layers are implemented:
  - The RX mixer
  - The XY mixer
  - The Hamming weight-preserving generalized SWAP mixer, defined as:

$$U_M \ = \ \prod_{(i,j)\in G} \exp\Big[-\frac{\beta}{2}(X_i X_j \ + \ Y_i Y_j)\Big]$$

Note: Each of the recommended mixers in each problem instance will simply be a call to one of the built-in mixers defined above, with the default for all problem instances being the RX mixer.

1.5.2 Defining the QAOA Circuit

With all of the necessary peices of the QAOA defined, they must be combined to create a variational QAOA circuit. We outline two possible ways to do this.

The first option is to have the user take their cost and mixer layers, along with some kind of qubit initialization, and pass it into a circuit like this:

```
import pennylane as qml
from pennylane import qaoa

qubits = range(4)
dev = qml.device("default.qubit", wires=len(qubits))

# Defines the graph and the cost Hamiltonian for MaxCut

graph = [(0, 1), (1, 2), (2, 3), (3, 0)]
maxcut = qaoa.problems.maxcut(graph=graph)

# Defines the unitaries, the circuit and the cost function
```

```
initialization = maxcut.init
cost_layer = maxcut.cost
mixer_layer = maxcut.mixer

qaoa_circuit = qaoa.circuit(cost_layer, mixer_layer, depth=3)

def circuit(params):

    initialization()
    qaoa_circuit(params)

    return qml.expval(qml.Hermitian(maxcut.hamiltonian, wires=qubits))

# Defines the cost function

cost_function = qml.QNode(circuit, dev)
```

The benefit of this method is that it allows for more flexibility in what goes on inside the variational circuit and the value that is returned after execution. It does, however, place more of a burden on the user to code out the whole variational circuit themself.

The second option is to have something like this:

```
import pennylane as qml
from pennylane import qaoa

qubits = range(4)
dev = qml.device("default.qubit", wires=len(qubits))

# Defines the graph and the cost Hamiltonian for MaxCut

graph = [(0, 1), (1, 2), (2, 3), (3, 0)]
maxcut = qaoa.problems.maxcut(graph=graph)

# Defines the unitaries, the circuit and the cost function

mixer_layer = maxcut.mixer
init_layer = maxcut.init

qaoa_circuit = qaoa.circuit(maxcut, mixer_layer, init_layer, depth=3)

# Defines the cost function

cost_function = qaoa.sampling_cost(qaoa_circuit, samples=200)
```

This allows for the user to easily define the variational QAOA circuit with a couple lines of code, but without as much customizability as the former option.

Note: The reason that we choose to keep the cost function separate from the QAOA circuit in this option is due to the fact that the QAOA circuit (not necessarily the cost function) is usually needed for post-processing.

Conclusion: The second option would probably be best for users. In the case that someone wanted as much freedom and customizability as the first option provides, they may as well code the QAOA circuit completely manually, without the help of the module. The second option on the other hand is still customizeable, but it allows the user to define and run a circuit in far fewer lines of code, and with much greater ease.

1.5.3 Post-processing

The final step of any QAOA workflow is post-processing of the data after determining the parameters that yield the optimal probability distribution of bitstrings. The goal of the QAOA is to find the bitstring that optimizes the cost function. Thus, we introduce functionality that can be used to determine this value, as well as other properties of the outputted data.

Within the core functionality of the PennyLane QAOA module, there are two post-processing functions that we propose:

- A function that samples from the variational QAOA circuit, for a given set of parameters. This allows the user to sample from the circuit many times, and infer things like the most commonly occurring bitstring, the average value of the cost function, the standard deviation of samples, etc.
- A function that returns the probability of measuring a given bitstring after executing a given QAOA circuit for some collection of parameters. This would allow the users to validate that their QAOA experiment is working as expected, by ensuring that the bitstrings that correspond to optimal values of the cost function occurr with high probability.

We choose to leave the post-processing functionality of the module fairly basic for the first iteration of the QAOA module, as a user that has some experience with Python should be able to take these two functions and use them to calculate essentially anything they want.

2 Other

There are a few other features that the QAOA library should likely include that don't fall into any of the previous categories:

- Functions that initialize α and γ parameters from normal and uniform distributions, for a given circuit.

3 User Interface

All together, a general QAOA workflow might look something like this (assuming the second option for defining the QAOA circuit is chosen):

```
import pennylane as qml
from pennylane import qaoa
import math
import numpy as np

qubits = range(4)
dev = qml.device("default.qubit", wires=len(qubits))

# Defines a QUBO problem with the following structure:
#   Interaction terms --> (qubit index 1, qubit index 2, weight)
#   Bias terms --> (qubit index, bias)

interaction_terms = [(0, 1, 2.7), (1, 2, 0.43), (2, 3, 1.2), (3, 0, 0.15)]
bias_terms = [(0, 2.3), (3, 0.93)]

ising_model = [interaction_terms, bias_terms]

qubo = qaoa.problems.qubo(ising_model=ising_model)

# Defines a custom mixer, using two built-in mixers
def mixer_layer(beta):

    qaoa.mixers.XYLayer(beta, qubits, graph=[(0, 1), (1, 2), (2, 3), (3, 0)])
```

```
    qaoa.mixers.RXLayer(beta, qubits)

# Defines the initialization circuit
init_layer = qubo.init

# Defines the QAOA circuit
qaoa_circuit = qaoa.circuit(qubo, mixer_layer, init_layer, depth=3)

# Defines a sampling cost function
qaoa_cost = qaoa.sampling_cost(qubo, qaoa_circuit, samples=200)

# Runs the circuit
steps = 100
optimizer = qml.AdamOptimizer()
params = qaoa.uniform_params(qaoa_circuit)

for i in range(0, steps):
        params = optimizer.step(qaoa_cost, params)

# Finds the optimal bitstring
optimal_bitstring = None
optimal_cost = math.inf
samples = 100

for i in range(0, samples):
        sample = qaoa.sample(qaoa_circuit)
    val = np.vdot(sample, (qubo.hamiltonian @ sample))

    if val < optimal_cost:
        optimal_cost = val
        optimal_bitstring = sample

print("The Optimal Bitstring is: {}".format(optimal_bitstring))
```