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].

If we are given 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 bitstring(s) that returns the lowest value.

QAOA is able to solve this problem by taking a variational approach. We first map the cost function to a diagonal cost Hamiltonian. We then use a parametrized quantum circuit to repeatedly prepare ansatz states, measuring the expectation value of the cost Hamiltonian with respect to this state. This process is repeated with new parameters, chosen by a classical optimizer, until the algorithm converges on the state that yields the lowest energy expectation value. This yields a set of optimal parameters, which are used to execute the circuit many times to find the optimal bitstring(s), thus solving the optimization problem.

In QAOA, the ansatz state is prepared in a unique way. 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 the alternating unitary evolution, similarly to a Trotterized version of adiabatic quantum evolution:

$$|\psi(\gamma,\alpha)\rangle \ = \ e^{-i\alpha_P H_M} e^{-i\gamma_P H_C} \ ... \ e^{-i\alpha_2 H_M} e^{-i\gamma_2 H_C} e^{-i\alpha_1 H_M} e^{-i\gamma_1 H_C} |\psi_0\rangle$$

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

The cost function is then defined to be $F(\gamma,\alpha) \ = \ \langle\psi(\gamma,\alpha)|H_C|\psi(\gamma,\alpha)\rangle$, which we minimize, as was described above.

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. Extensive research has been done on the possibility of using it as a robust variational algorithm that can achieve quantum supremacy on near-term quantum computers and as a result, researchers are interested in performing calculations, simulations, and benchmarking of different QAOA experiments on classical computers.

1.3 Analysis

There are four main components that are necessary to defining a QAOA workflow:

1. Specifying the optimization problem
2. Allowing the user to define a mixer layer
3. Creating a cost layer and combining it with the mixer and pre-processing to create a variational circuit
4. Post-processing of results

This ADR will disucss each in detail.

The main goal of the QAOA module is to make it as **problem-centric** as possible, where the user can 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 pennylane.qaoa module where all of the functionality for creating QAOA workflows would be located.

1.4.1 The Cost and Mixer Layers

The main component of any QAOA procedure is the cost function that the user wants to optimize. In the PennyLane QAOA module, we propose two ways that a user can go about choosing an optimization problem to solve:

1. The user passes a custom cost Hamiltonian matrix into the workflow.
2. The user chooses from one of the built-in cost Hamiltonians, which correspond to well-known optimization problems. To begin, we propose that the following optimization problems be implemented:
    - MaxCut (for a user-defined graph)
    - 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 optimization problems will be instances of this "Problem" class with the following attributes:

- The matrix form of the cost Hamiltonian
- The cost layer. With knowledge of the cost Hamiltonian, the cost layer can be constructed. In theory, it is possible to simply take the cost Hamiltonian and decompose it into a linear combination of Z gates, however, this process would lengthen the runtime of the simulation. Thus, when defining new, built-in cost function that have an already well-know quantum circuit implementation (for instance, MaxCut), there is an option to specify the circuit directly.
- A *recommended* mixer layer, which is a function containing PennyLane operations. The default is built-in RX layer.
- 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 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.

There are also cases to consider where the circuit decomposition of the cost layer corresponding to some cost Hamiltonian is unknown (for example, when the user passes in their own custom Hamiltonian). Thus, we must write a subroutine that performs the decomposition for us. We begin by expressing the cost Hamiltonian as a linear combination 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:

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

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:

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

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

1.4.2 Further Notes on Mixer Layers

There are a wide variety of mixers that are suited to different instances of 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 optimization procedures.

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 function, 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 mixers **unitaries** are implemented:
  - The RX mixer
  - The XY mixer
  - The Hamming weight-preserving generalized SWAP mixer, defined as:

$$\exp\Big[-\frac{\beta}{2}(X_iX_j \ + \ Y_iY_j)\Big]$$

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 being the RX mixer.

1.4.3 Defining the QAOA Circuit

With all of the necessary peices of the simulation defined, it must be combined to create a variational QAOA circuit. We see 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 one line of code, but without as much customizability as the former option.

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.

1.4.4 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 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 circuit. 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 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

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))
```