1 Add QAOA Functionality to PennyLane

1.1 Status

2020-07-01: Proposed

1.2 Context

Ever since the introduction of the Quantum Approximate Optimization Algorithm (QAOA) by Farhi, Gutmann and Goldstone in 2014, extensive research has been done on the possibility of using it as a robust variational algorithm that can achieve quantum supremacy on NISQ devices. As a result, researchers are interested in performing calculations and simulations of different QAOA experiments on classical computers.

1.3 Background

The QAOA is a variational quantum algorithm for solving combinatorial problems. More specifically, if we are given a cost function of the form

1.4 Analysis

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

1. Defining a diagonal cost Hamiltonian
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. We want for the user to be able to enter some combinatorial problem into PennyLane, and in return, get the answer without worrying too much about the specifics of the quantum circuits that are being executed in the backend.

1.4.1 Cost Hamiltonians

The first step in performing QAOA involves the user defining a cost Hamiltonian that they wish to minimize. This can be done in two ways:

1. The user passes a custom 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 cost Hamiltonians be implemented:
    - MaxCut (for a user-defined graph)
    - QUBO (for a user-defined Ising model)

After the core functionality has been built, more built-in cost Hamiltonians corresponding to other common optimization problems can be added.

1.4.2 Mixer Layers

The user must also have the ability to specify a mixer Hamiltonian to be used in the QAOA workflow.

Mixers could be defined in a similar way to the QAOA problem instances. The user can either define their own function, containing PennyLane operations, which can be used as the mixer layer, or they can 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:

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

With these unitaries defined, **broadcasted** to the circuit using the PennyLane broadcast function.

1.4.3 Code Structure

In terms of code structure, the QAOA module will have a general class which can be used to define an optimization problem. Each of the cost Hamiltonians 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.
- 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

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. To define the cost layer, the cost Hamiltonian will be decomposed into 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:

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

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

1.4.4 Defining the QAOA Circuit

There are two possible ways the QAOA circuit can be defined, after defining the cost and mixer layers.

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, output="expval")

# Defines the cost function

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

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.

1.4.5 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.

1.5 Conclusion