Grove Documentation

Release 0.0.0

Rigetti Quantum Computing

Jun 18, 2017

Contents

Grove is a open source Python library containing quantum algorithms that uses the quantum programming library pyQuil and the Rigetti Forest toolkit.

CHAPTER 1

Structure

Grove is organized into modules for the various quantum algorithms, each of which has its own self-contained documentation.

Installation and Getting Started

Prerequisites

Before you can start writing using Grove, you will need Python 2.7 (version 2.7.10 or greater) and the Python package manager pip. We recommend installing Anaconda for an all-in-one installation of Python 2.7. If you don't have pip, it can be installed with `easy_install pip`.

Installation

You can install Grove directly from the Python package manager pip using:

```
pip install quantum-grove
```

To instead install the bleeding-edge version from source, clone the Grove GitHub repository, `cd` into it, and run:

```
pip install -e .
```

This will install Grove's dependencies if you do not already have them. The dependencies are:

- NumPy
- SciPy
- NetworkX
- Matplotlib
- pytest *(optional, for testing)*

- mock *(optional, for testing)*

Forest and pyQuil

Grove also requires the Python library for Quil, called pyQuil.

After obtaining the library from the pyQuil GitHub repository or from a source distribution, navigate into its directory in a terminal and run:

```
pip install -e .
```

You will need to make sure that your pyQuil installation is properly configured to run with a QVM or quantum processor (QPU) hosted on the Rigetti Forest, which requires an API key. See the pyQuil docs for instructions on how to do this.

Quantum Teleportation

Quantum teleportation is a method for transmitting the information of a qubit from one location to another. The process relies on a previously shared entangled state between the two locations, and classical communication.

Overview

In the canonical description of quantum teleportation[1][2] two parties (Alice and Bob) are trying to transmit a state from one to another. They start with an Alice having half of an entangled bell pair (Bob having the other half) and Alice with a third qubit that she would like to transmit to Bob. Alice then entangles the third qubit with her Bell pair qubit, measures both her qubits, and sends the measurement result $\{0, 1\}^2$ to Bob. Bob uses the classical information from Alice to fix up his state and now has his qubit in the state of Alice's original qubit she wanted to transfer.

Given that Alice's data qubit is labeled 0 and the Bell pair qubits are labeled 1 and 2 (Bob having the qubit labeled 2), a Quil program performing the transfer is as follows:

```
CNOT 0 1
H 0
MEASURE 0 [0]
MEASURE 1 [1]
JUMP-UNLESS @ NOX [1]
X 2
LABEL @NOX
JUMP-UNLESS @NOZ [0]
Z 2
LABEL @NOZ
```

The Bell state preparation can be prepended to the above Quil with:

```
H 1
CNOT 1 2
```

[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.
[2] Wikipedia contributors. "Quantum teleportation." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 4 Jan. 2017. Web. 4 Jan. 2017.

Teleportation in pyQuil

We have included pyQuil code that programatically generates a teleportation program between any two qubits using one ancilla qubit. The ancilla qubit is the piece of the Bell pair that Alice holds. Check the teleportation.py source code for examples.

Source Code Docs

Here you can find documentation for the different submodules in teleport.

grove.teleport.teleportation

References

Variational-Quantum-Eigensolver (VQE)

Overview

The Variational-Quantum-Eigensolver (VQE) [1, 2] is a quantum/classical hybrid algorithm that can be used to find eigenvalues of a (often large) matrix H. When this algorithm is used in quantum simulations, H is typically the Hamiltonian of some system [3, 4, 5]. In this hybrid algorithm a quantum subroutine is run inside of a classical optimization loop.

The quantum subroutine has two fundamental steps:

1. Prepare the quantum state $|\Psi(\vec{\theta})\rangle$, often called the *ansatz*.
2. Measure the expectation value $\langle \Psi(\vec{\theta}) \mid H \mid \Psi(\vec{\theta}) \rangle$.

The variational principle ensures that this expectation value is always greater than the smallest eigenvalue of H.

This bound allows us to use classical computation to run an optimization loop to find this eigenvalue:

1. Use a classical non-linear optimizer to minimize the expectation value by varying ansatz parameters $\vec{\theta}$.
2. Iterate until convergence.

Practically, the quantum subroutine of VQE amounts to preparing a state based off of a set of parameters $\vec{\theta}$ and performing a series of measurements in the appropriate basis. The paramaterized state (or ansatz) preparation can be tricky in these algorithms and can dramatically affect performance. Our `VQE` module allows any Python function that returns a pyQuil program to be used as an ansatz generator. This function is passed into `vqe_run` as the `variational_state_evolve` argument. More details are in the source documentation.

Measurements are then performed on these states based on a Pauli operator decomposition of H. Using Quil, these measurements will end up in classical memory. Doing this iteratively followed by a small amount of postprocessing, one may compute a real expectation value for the classical optimizer to use.

Below there is a very small first example of VQE and Grove's implementation of the Quantum Approximate Optimization Algorithm QAOA also makes use of the VQE module.

Basic Usage

Here we will take you through an example of a very small variational quantum eigensolver problem. In this example we will use a quantum circuit that consists of a single parametrized gate to calculate an eigenvalue of the Pauli Z matrix.

First we import the necessary pyQuil modules to construct our ansatz pyQuil program.

```
from pyquil.quil import Program
import pyquil.api as api
from pyquil.gates import *
qvm = api.SyncConnection()
```

Any Python function that takes a list of numeric parameters and outputs a pyQuil program can be used as an ansatz function. We will see some more examples of this later. For now, we just take a parameter list with a single parameter.

```
def small_ansatz(params):
    return Program(RX(params[0], 0))

print small_ansatz([1.0])
```

```
RX(1.0) 0
```

This `small_ansatz` function is our $\Psi(\vec{\theta})$. To construct the Hamiltonian that we wish to simulate, we use the `pyquil.paulis` module.

```
from pyquil.paulis import sZ
initial_angle = [0.0]
# Our Hamiltonian is just \sigma_z on the zeroth qubit
hamiltonian = sZ(0)
```

We now use the *vqe* module in Grove to construct a `VQE` object to perform our algorithm. In this example, we use `scipy.optimize.minimize()` with Nelder-Mead as our classical minimizer, but you can choose other parameters or write your own minimizer.

```
from grove import VQE
from scipy.optimize import minimize
import numpy as np

vqe_inst = VQE(minimizer=minimize,
               minimizer_kwargs={'method': 'nelder-mead'})
```

Before we run the minimizer, let us look manually at what expectation values $\langle \Psi(\vec{\theta}) \mid H \mid \Psi(\vec{\theta}) \rangle$ we calculate for fixed parameters of $\vec{\theta}$.

```
angle = 2.0
vqe_inst.expectation(small_ansatz([angle]), hamiltonian, None, qvm)
```

```
-0.4161468365471423
```

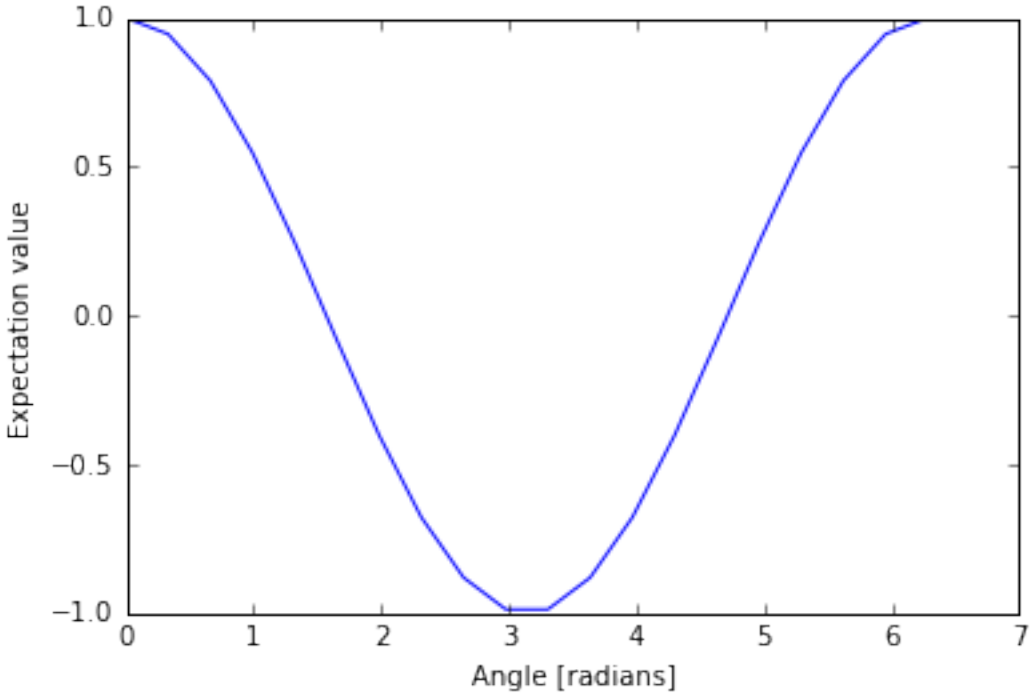
The expectation value was calculated by running the pyQuil program output from `small_ansatz`, saving the wavefunction, and using that vector to calculate the expectation value. We can sample the wavefunction as you would on a quantum computer by passing an integer, instead of *None*, as the samples argument of the `expectation()` method.

```
angle = 2.0
vqe_inst.expectation(small_ansatz([angle]), hamiltonian, 10000, qvm)
```

```
-0.42900000000000005
```

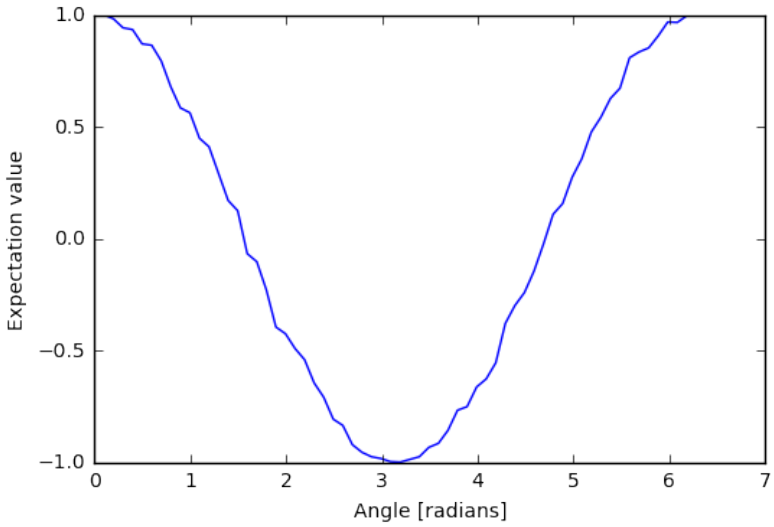
We can loop over a range of these angles and plot the expectation value.

```
angle_range = np.linspace(0.0, 2 * np.pi, 20)
data = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian, None, qvm)
        for angle in angle_range]

import matplotlib.pyplot as plt
plt.xlabel('Angle [radians]')
plt.ylabel('Expectation value')
plt.plot(angle_range, data)
plt.show()
```



Now with sampling...

```
angle_range = np.linspace(0.0, 2 * np.pi, 20)
data = [vqe_inst.expectation(small_ansatz([angle]), hamiltonian, 1000, qvm)
        for angle in angle_range]

import matplotlib.pyplot as plt
plt.xlabel('Angle [radians]')
plt.ylabel('Expectation value')
plt.plot(angle_range, data)
plt.show()
```



We can compare this plot against the value we obtain when we run the our variational quantum eigensolver.

```
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, None, qvm=qvm)
print result
```

```
{'fun': -0.99999999954538055, 'x': array([ 3.1415625])}
```

Running Noisy VQE

A great thing about VQE is that it is somewhat insensitive to noise. We can test this out by running the previous algorithm on a noisy qvm.

Remember that Pauli channels are defined as a list of three probabilities that correspond to the probability of a random X, Y, or Z gate respectively. First we'll study the impact of a channel that has the same probability of each random Pauli.

```
pauli_channel = [0.1, 0.1, 0.1] #10% chance of each gate at each timestep
noisy_qvm = api.SyncConnection(gate_noise=pauli_channel)
```

Let us check that this QVM has noise:

```
p = Program(X(0), X(1)).measure(0, [0]).measure(1, [1])
noisy_qvm.run(p, [0, 1], 10)
```

```
[[1, 1],
 [0, 1],
 [1, 0],
 [0, 1],
 [0, 0],
 [1, 1],
 [0, 1],
 [1, 0],
 [1, 0],
 [0, 1]]
```

We can run the VQE under noise. Let's modify the classical optimizer to start with a larger simplex so we don't get stuck at an initial minimum.

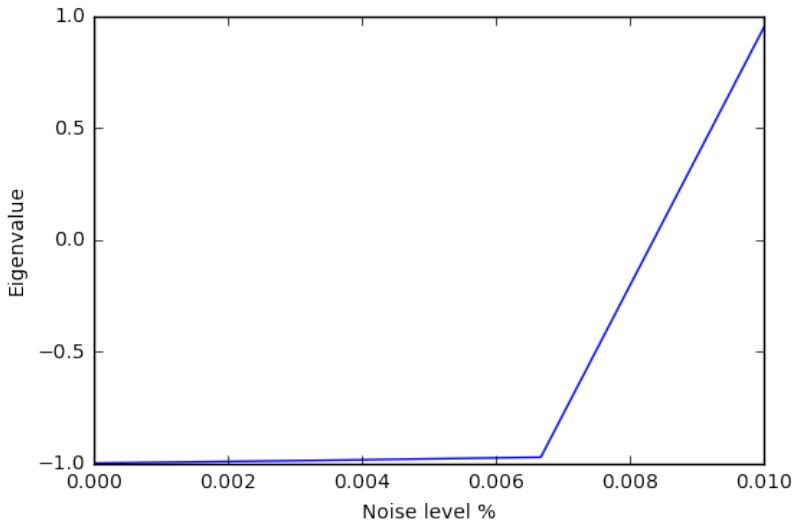
```
vqe_inst.minimizer_kwargs = {'method': 'Nelder-mead', 'options': {'initial_simplex':␣
↪np.array([[0.0], [0.05]]), 'xatol': 1.0e-2}}
result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, samples=10000,␣
↪qvm=noisy_qvm)
print result
```

```
{'fun': 0.5875999999999999, 'x': array([ 0.01874886])}
```

10% error is a huge amount of error! We can plot the effect of increasing noise on the result of this algorithm:

```
data = []
noises = np.linspace(0.0, 0.01, 4)
for noise in noises:
    pauli_channel = [noise] * 3
    noisy_qvm = api.SyncConnection(gate_noise=pauli_channel)
    # We can pass the noise params directly into the vqe_run instead of passing the␣
↪noisy connection
    result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle,
                              gate_noise=pauli_channel)
    data.append(result['fun'])
```

```
plt.xlabel('Noise level %')
plt.ylabel('Eigenvalue')
plt.plot(noises, data)
plt.show()
```



It looks like this algorithm is pretty robust to noise up until 0.6% error. However measurement noise might be a different story.

```
meas_channel = [0.1, 0.1, 0.1] #10% chance of each gate at each measurement
noisy_meas_qvm = api.SyncConnection(measurement_noise=meas_channel)
```

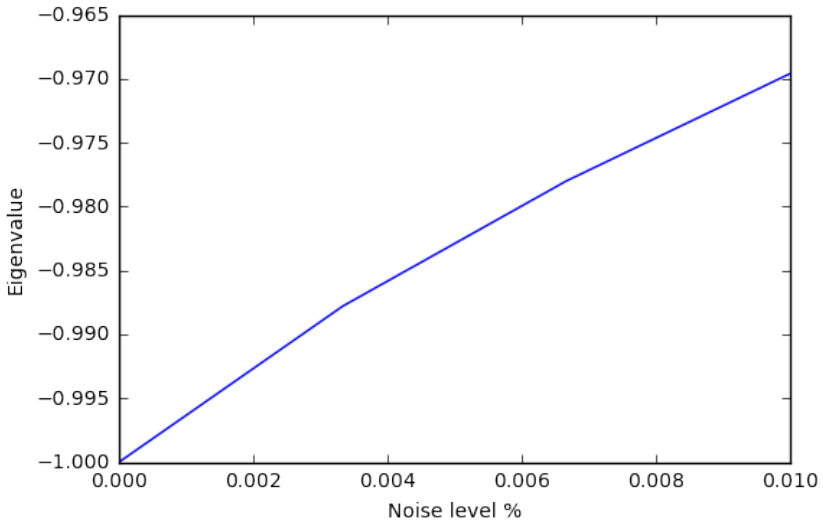
Measurement noise has a different effect:

```
p = Program(X(0), X(1)).measure(0, [0]).measure(1, [1])
noisy_meas_qvm.run(p, [0, 1], 10)
```

```
[[1, 1],
 [1, 1],
 [1, 1],
 [1, 1],
 [1, 1],
 [1, 1],
 [0, 1],
 [1, 0],
 [1, 1],
 [1, 0]]
```

```
data = []
noises = np.linspace(0.0, 0.01, 4)
for noise in noises:
    meas_channel = [noise] * 3
    noisy_qvm = api.SyncConnection(measurement_noise=meas_channel)
    result = vqe_inst.vqe_run(small_ansatz, hamiltonian, initial_angle, samples=10000,
↪ qvm=noisy_qvm)
    data.append(result['fun'])
```

```
plt.xlabel('Noise level %')
plt.ylabel('Eigenvalue')
plt.plot(noises, data)
plt.show()
```



We see this particular VQE algorithm is generally more sensitive to measurement noise than gate noise.

More Sophisticated Ansatzes

Because we are working with Python, we can leverage the full language to make much more sophisticated ansatzes for VQE. As an example we can easily change the number of gates.

```
def smallish_ansatz(params):
    return Program(RX(params[0], 0), RX(params[1], 0))

print smallish_ansatz([1.0, 2.0])
```

```
RX(1.0) 0
RX(2.0) 0
```

```
vqe_inst = VQE(minimizer=minimize,
               minimizer_kwargs={'method': 'nelder-mead'})
initial_angles = [1.0, 1.0]
result = vqe_inst.vqe_run(smallish_ansatz, hamiltonian, initial_angles, None, qvm=qvm)
print result
```

```
{'fun': -1.0000000000000004, 'x': array([ 1.61767133,  1.52392133])}
```

We can even dynamically change the gates in the circuit based on a parameterization:

```
def variable_gate_ansatz(params):
    gate_num = int(np.round(params[1])) # for scipy.minimize params must be floats
    p = Program(RX(params[0], 0))
    for gate in range(gate_num):
        p.inst(X(0))
    return p

print variable_gate_ansatz([0.5, 3])
```

```
RX(0.5) 0
X 0
X 0
X 0
```

```
initial_params = [1.0, 3]
result = vqe_inst.vqe_run(variable_gate_ansatz, hamiltonian, initial_params, None,␣
↪qvm=qvm)
print result
```

```
{'fun': -1.0, 'x': array([  2.65393312e-09,   3.42891875e+00])}
```

Note that the restriction that the ansatz function take a single list of floats as parameters only comes from our choice of minimizer (this is what `scipy.optimize.minimize` takes). One could easily imagine writing a custom minimizer that takes more sophisticated forms of arguments.

Links and Further Reading

This concludes our brief tour of VQE. There is a lot of fascinating literature about this algorithm out there and we encourage you to both explore those topics as well as come up with new ideas using this library. Let us know if you have ideas about anything that you would like to see added!

Here are some links to get you started:

- A Variational Eigenvalue Solver on a Quantum Processor
- The Theory of Variational Hybrid Quantum-Classical Algorithms
- Hybrid Quantum-Classical Approach to Correlated Materials
- A Hybrid Classical/Quantum Approach for Large-Scale Studies of Quantum Systems with Density Matrix Embedding Theory
- Hybrid Quantum-Classical Hierarchy for Mitigation of Decoherence and Determination of Excited States

Source Code Docs

Here you can find documentation for the different submodules in pyVQE.

grove.pyvqe.vqe

Quantum Approximate Optimization Algorithm (QAOA)

Overview

pyQAOA is a Python module for running the Quantum Approximate Optimization Algorithm on an instance of a quantum abstract machine.

The pyQAOA package contains separate modules for each type of problem instance: MAX-CUT, graph partitioning, etc. For each problem instance the user specifies the driver Hamiltonian, cost Hamiltonian, and the approximation order of the algorithm.

`qaoa.py` contains the base QAOA class and routines for finding optimal rotation angles via Grove's variational-quantum-eigensolver method.

Cost Functions

- `maxcut_qaoa.py` implements the cost function for MAX-CUT problems.
- `numpartition_qaoa.py` implements the cost function for bipartitioning a list of numbers.

Quickstart Examples

To test your installation and get going we can run QAOA to solve MAX-CUT on a square ring with 4 nodes at the corners. In your python interpreter import the packages and connect to your QVM:

```
import numpy as np
from grove.pyqaoa.maxcut_qaoa import maxcut_qaoa
import pyquil.api as qvm
qvm_connection = qvm.SyncConnection()
```

Next define the graph on which to run MAX-CUT

```
square_ring = [(0,1),(1,2),(2,3),(3,0)]
```

The optional configuration parameter for the algorithm is given by the number of steps to use (which loosely corresponds to the accuracy of the optimization computation). We instantiate the algorithm and run the optimization routine on our QVM:

```
steps = 2
inst = maxcut_qaoa(graph=square_ring, steps=steps)
betas, gammas = inst.get_angles()
```

to see the final \(\mid \beta, \gamma \rangle \) state we can rebuild the quil program that gives us \(\mid \beta, \gamma \rangle \) and evaluate the wave function using the QVM

```
t = np.hstack((betas, gammas))
param_prog = inst.get_parameterized_program()
prog = param_prog(t)
wf, _ = qvm_connection.wavefunction(prog)
wf = wf.amplitudes
```

`wf` is now a numpy array of complex-valued amplitudes for each computational basis state. To visualize the distribution iterate over the states and calculate the probability.

```
for state_index in range(2**inst.n_qubits):
    print inst.states[state_index], np.conj(wf[state_index])*wf[state_index]
```

You should then see that the algorithm converges on the expected solutions of 0101 and 1010!

```
0000 (4.38395094039e-26+0j)
0001 (5.26193287055e-15+0j)
0010 (5.2619328789e-15+0j)
0011 (1.52416449345e-13+0j)
0100 (5.26193285935e-15+0j)
0101 (0.5+0j)
0110 (1.52416449362e-13+0j)
0111 (5.26193286607e-15+0j)
1000 (5.26193286607e-15+0j)
1001 (1.52416449362e-13+0j)
1010 (0.5+0j)
1011 (5.26193285935e-15+0j)
1100 (1.52416449345e-13+0j)
1101 (5.2619328789e-15+0j)
1110 (5.26193287055e-15+0j)
1111 (4.38395094039e-26+0j)
```

Algorithm and Details

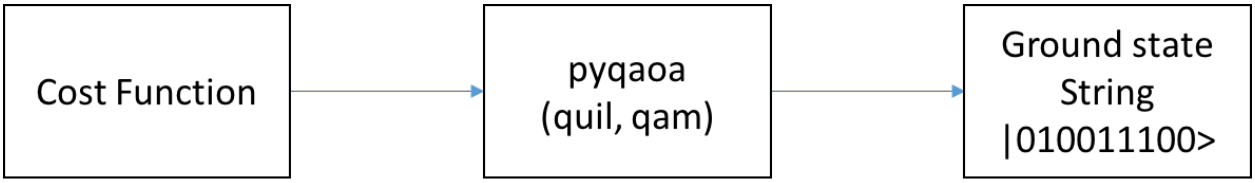
Introduction

The quantum-approximate-optimization-algorithm (QAOA, pronouced quah-wah), developed by Farhi, Goldstone, and Gutmann, is a polynomial time algorithm for finding "a 'good' solution to an optimization problem" [1, 2].

What's with the name? For a given NP-Hard problem an approximate algorithm is a polynomial-time algorithm that solves every instance of the problem with some guaranteed quality in expectation. The value of merit is the ratio between the quality of the polynomial time solution and the quality of the true solution.

One reason QAOA is interesting is its potential to exhibit quantum supremacy [1].

This package, which is an implementation of QAOA that runs on a simulated quantum computer, can be used as a stand alone optimizer or a plugin optimization routine in a larger environment. The usage pipeline is as follows: 1) encoding the cost function into a set of Pauli operators, 2) instantiating the problem with pyQAOA and pyQuil, and 3) retrieving ground state solution by sampling.

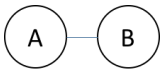


The following section of the pyQAOA documentation describes the algorithm and the NP-hard problem instance used in the original paper.

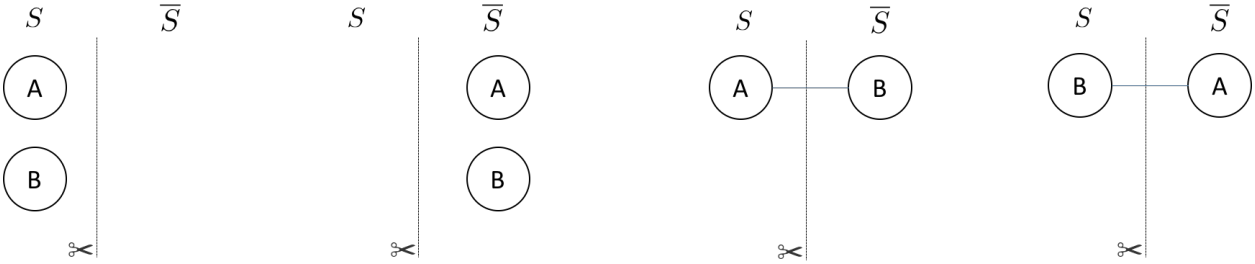
Our First NP-Hard Problem

The maximum-cut problem (MAX-CUT) was the first application described in the original quantum-approximate-optimization-algorithm paper [2]. This problem is similar to graph coloring. Given a graph of nodes and edges, color each node black or white, then score a point for each node that is next to a node of a different color. The aim is to find a coloring that scores the most points.

Stated a bit more formally, the problem is to partition the nodes of a graph into two sets such that the number of edges connecting nodes in opposite sets is maximized. For example, consider the barbell graph

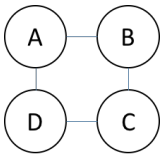


there are 4 ways of partitioning nodes into two sets:



We have drawn the edge only when it connects nodes in different sets. The line with the scissor symbol indicates that we count the edge in our cut. For the barbell graph there are two equal weight partitionings that correspond to a maximum cut (the right two partitonings)–i.e. cutting the barbell in half. One can denote which set \(S \) or \(\overline{S} \) a node is in with either a \(0\) or a \(1\), respectively, in a bit string of length \(N \). The four partitionings of the barbell graph listed above are, \(\{ 00, 11, 01, 10 \} \)—where the left most bit is node \(A\) and the right most bit is node \(B\). The bit string representation makes it easy to represent a particular partition of the graph. Each bit string has an associated cut weight.

For any graph, the bit string representations of the node partitionings are always length \(N\). The total number of partitionings grows as \(2^{N}\). For example, a square ring graph



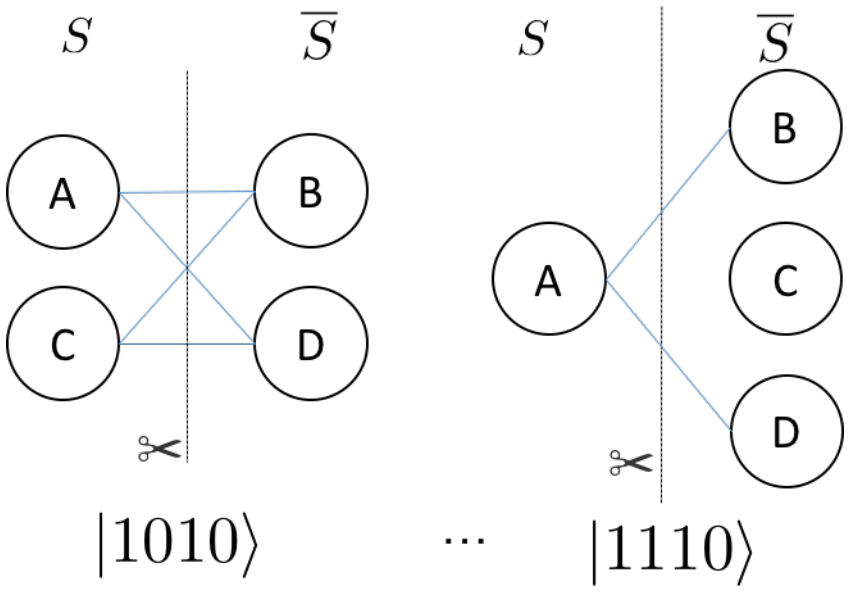
has 16 possible partitions (\(2^{4}\)). Below are two possible ways of parititioning of the nodes.

The bit strings associated with each parititioning are indicated in the figure. The right most bit corresponds with the node labeled \(A\) and the left most bit corresponds with the node labeled \(D\).

Classical Solutions

In order to find the best cut on a classical computer the obvious approach is to enumerate all partitions of the graph and check the weight of the cut associated with the partition.

Faced with an exponential cost for finding the optimal cut (or set of optimal cuts) one can devise a polynomial algorithm that is guaranteed to be of a particular quality. For example, a famous polynomial time algorithm is the



randomized partitioning approach. One simply iterates over the nodes of the graph and flips a coin. If the coin is heads the node is in \(S \), if tails the node is in \(\overline{S} \). The quality of the random assignment algorithm is at least 50 percent of the maximum cut. For a coin-flip process the probability of an edge being in the cut is 50%. Therefore, the expectation value of a cut produced by random assignment can be written as follows: $$\sum_{e \in E} w_{e} \cdot \mathrm{Pr}(e \in \mathrm{cut}) = \frac{1}{2} \sum_{e \in E}w_{e}$$ Since the sum of all the edges is necessarily an upper bound to the maximum cut the randomized approach produces a cut of expected value of at least 0.5 times the best cut on the graph.

Other polynomial approaches exist that involve semi-definite programming which give cuts of expected value at least 0.87856 times the maximum cut [3].

Quantum Approximate Optimization

One can think of the bit strings (or set of bit strings) that correspond to the maximum cut on a graph as the ground state of a Hamiltonian encoding the cost function. The form of this Hamiltonian can be determined by constructing the classical function that returns a 1 (or the weight of the edge) if the edge spans two-nodes in different sets, or 0 if the nodes are in the same set. \begin{align} C_{ij} = \frac{1}{2}(1 - z_{i}z_{j}) \end{align} \(z_{i}\) or \(z_{j}\) is \(+1\) if node \(i\) or node \(j\) is in \(S\) or \(-1\) if node \(i\) or node \(j\) is in \(\overline{S}\). The total cost is the sum of all \((i ,j) \) node pairs that form the edge set of the graph. This suggests that for MAX-CUT the Hamiltonian that encodes the problem is $$\sum_{ij}\frac{1}{2}(\mathbf{I} - \sigma_{i}^{z}\sigma_{j}^{z})$$ where the sum is over \((i,j) \) node pairs that form the edges of the graph. The quantum-approximate-optimization-algorithm relies on the fact that we can prepare something approximating the ground state of this Hamiltonian and perform a measurement on that state. Performing a measurement on the \(N\)-body quantum state returns the bit string corresponding to the maximum cut with high probability.

To make this concrete let us return to the barbell graph. The graph requires two qubits in order to represent the nodes. The Hamiltonian has the form \begin{align} \hat{H} = \frac{1}{2}(\mathbf{I} - \sigma_{z}^{1}\otimes \sigma_{z}^{0}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{align} where the basis ordering corresponds to increasing integer values in binary format (the left most bit being the most significant). This corresponds to a basis ordering for the \(\hat{H}\) operator above as \begin{align} (| 00\rangle, | 01\rangle, | 10\rangle, | 11\rangle). \end{align} Here the Hamiltonian is diagonal with integer eigenvalues. Clearly each bit string is an eigenstate of the Hamiltonian because \(\hat{H}\) is diagonal.

QAOA identifies the ground state of the MAXCUT Hamiltonian by evolving from a reference state. This reference state is the ground state of a Hamiltonian that couples all \(2^{N} \) states that form the basis of the cost Hamiltonian—

i.e. the diagonal basis for cost function. For MAX-CUT this is the \(Z\) computational basis.

The evolution between the ground state of the reference Hamiltonian and the ground state of the MAXCUT Hamiltonian can be generated by an interpolation between the two operators \begin{align} \hat{H}_{\tau} = \tau\hat{H}_{\mathrm{ref}} + (1 - \tau)\hat{H}_{\mathrm{MAXCUT}} \end{align} where \(\tau\) changes between 1 and 0. If the ground state of the reference Hamiltonian is prepared and \(\tau = 1\) the state is a stationary state of \(\hat{H}_{\tau}\). As \(\hat{H}_{\tau}\) transforms into the MAXCUT Hamiltonian the ground state will evolve as it is no longer stationary with respect to \(\hat{H}_{\tau \neq 1 }\). This can be thought of as a continuous version of the of the evolution in QAOA.

The appproximate portion of the algorithm comes from how many values of \(\tau\) are used for approximating the continuous evolution. We will call this number of slices \(\alpha\). The original paper [2] demonstrated that for \(\alpha = 1\) the optimal circuit produced a distribution of states with a Hamiltonian expectation value of 0.6924 of the true maximum cut for 3-regular graphs. Furthermore, the ratio between the true maximum cut and the expectation value from QAOA could be improved by increasing the number of slices approximating the evolution.

Details

For MAXCUT, the reference Hamiltonian is the sum of \(\sigma_{x}\) operators on each qubit. \begin{align} \hat{H}_{\mathrm{ref}} = \sum_{i=0}^{N-1} \sigma_{i}^{X} \end{align} This Hamiltonian has a ground state which is the tensor product of the lowest eigenvectors of the \(\sigma_{x}\) operator (\(\mid + \rangle\)). \begin{align} \mid \psi_{\mathrm{ref}}\rangle = \mid + \rangle_{N-1}\otimes\mid + \rangle_{N-2}\otimes...\otimes\mid + \rangle_{0} \end{align}

The reference state is easily generated by performing a Hadamard gate on each qubit–assuming the initial state of the system is all zeros. The Quil code generating this state is

```
H 0
H 1
...
H N-1
```

pyQAOA requires the user to input how many slices (approximate steps) for the evolution between the reference and MAXCUT Hamiltonian. The algorithm then variationally determines the parameters for the rotations (denoted \(\beta\) and \(\gamma\)) using the quantum-variational-eigensolver method [4][5] that maximizes the cost function.

For example, if (\(\alpha = 2\)) is selected two unitary operators approximating the continuous evolution are generated. \begin{align} U = U(\hat{H}_{\alpha_{1}})U(\hat{H}_{\alpha_{0}}) \label{eq:evolve} \end{align} Each \(U(\hat{H}_{\alpha_{i}})\) is approximated by a first order Trotter-Suzuki decomposition with the number of Trotter steps equal to one \begin{align} U(\hat{H}_{s_{i}}) = U(\hat{H}_{\mathrm{ref}}, \beta_{i})U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i}) \end{align} where \begin{align} U(\hat{H}_{\mathrm{ref}}, \beta_{i}) = e^{-i \hat{H}_{\mathrm{ref}} \beta_{i}} \end{align} and \begin{align} U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i}) = e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{i}} \end{align} \(U(\hat{H}_{\mathrm{ref}}, \beta_{i}) \) and \(U(\hat{H}_{\mathrm{MAXCUT}}, \gamma_{i})\) can be expressed as a short quantum circuit.

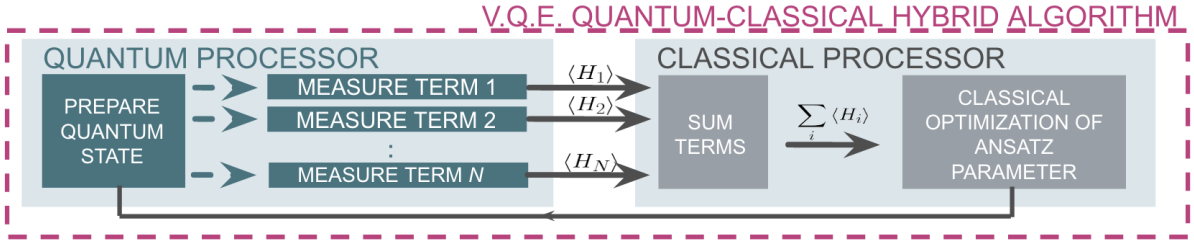
For the \(U(\hat{H}_{\mathrm{ref}}, \beta_{i})\) term (or mixing term) all operators in the sum commute and thus can be split into a product of exponentiated \(\sigma_{x}\) operators. \begin{align} e^{-i\hat{H}_{\mathrm{ref}} \beta_{i}} = \prod_{n = 0}^{1}e^{-i\sigma_{n}^{x}\beta_{i}} \end{align}

```
H 0
RZ(beta_i) 0
H 0
H 1
RZ(beta_i) 1
H 1
```

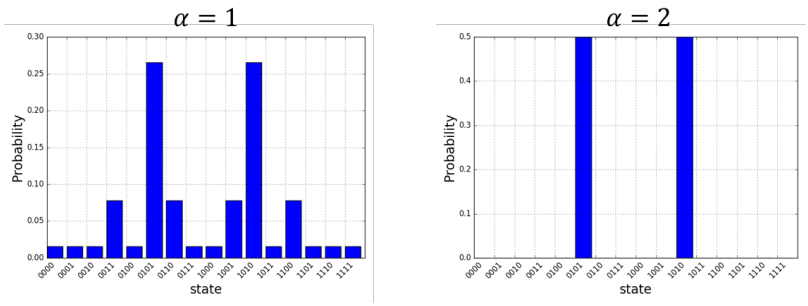
Of course, if RX is in the natural gate set for the quantum-processor this Quil is compiled into a set of RX rotations. The Quil code for the cost function \begin{align} e^{-i \frac{\gamma_{i}}{2}(\mathbf{I} - \sigma_{1}^{z} \otimes \sigma_{0}^{z}) } \end{align} looks like this:

```
X 0
PHASE(gamma{i}/2) 0
X 0
PHASE(gamma{i}/2) 0
CNOT 0 1
RZ(gamma{i}/2) 1
CNOT 0 1
```

Executing the Quil code will generate the \(\mid + \rangle_{1}\otimes\mid + \rangle_{0}\) state and perform the evolution with selected \(\beta\) and \(\gamma\) angles. \begin{align} \mid \beta, \gamma \rangle = e^{-i \hat{H}_{\mathrm{ref}} \beta_{1}}e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{1}}e^{-i \hat{H}_{\mathrm{ref}} \beta_{0}}e^{-i \hat{H}_{\mathrm{MAXCUT}} \gamma_{0}} \mid + \rangle_{N-1,...,0} \end{align} In order to indentify the set of \(\beta\) and \(\gamma\) angles that maximize the objective function \begin{align} \mathrm{Cost} = \langle \beta, \gamma \mid \hat{H}_{\mathrm{MAXCUT}} \mid \beta, \gamma \rangle \label{expect} \end{align} pyQAOA leverages the classical-quantum hybrid approach known as the quantum-variational-eigensolver[4][5]. The quantum processor is used to prepare a state through a polynomial number of operations which is then used to evaluate the cost. Evaluating the cost (\(\langle \beta, \gamma \mid \hat{H}_{\mathrm{MAXCUT}} \mid \beta, \gamma \rangle\)) requires many preparations and measurements to generate enough samples to accurately construct the distribution. The classical computer then generates a new set of parameters (\(\beta, \gamma\)) for maximizing the cost function.



By allowing variational freedom in the \(\beta \) and \(\gamma \) angles QAOA finds the optimal path for a fixed number of steps. Once optimal angles are determined by the classical optimization loop one can read off the distribution by many preparations of the state with \(\beta, \gamma\) and sampling.



The probability distributions above are for the four ring graph discussed earlier. As expected the approximate evolution becomes more accurate as the number of steps (\(\alpha\)) is increased. For this simple model \(\alpha = 2\) is sufficient to find the two degenerate cuts of the four ring graph.

Source Code Docs

Here you can find documentation for the different submodules in pyQAOA.

grove.pyqaoa.qaoa

grove.pyqaoa.maxcut_qaoa

grove.pyqaoa.numpartition_qaoa

Quantum Fourier Transform (QFT)

Overview

The quantum Fourier transform is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction. Detailed explanations can be found in references[1] and[2]. The QFT forms the basis of many quantum algorithms such as Shor's factoring algorithm, discrete logarithm, and others to be found in the quantum algorithms zoo[3].

Source Code Docs

Here you can find documentation for the different submodules in qft.

grove.qft.fourier

References

Phase Estimation Algorithm

Overview

The phase estimation algorithm is a quantum subroutine useful for finding the eigenvalue corresponding to an eigenvector \(u\) of some unitary operator. It is the starting point for many other algorithms and relies on the inverse quantum Fourier transform. More details can be found in references[1].

Source Code Docs

Here you can find documentation for the different submodules in phaseestimation.

[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.
[2] Rieffel, E. G., and W. Polak. "A Gentle Introduction to Quantum Computing." (2011).
[3] http://math.nist.gov/quantum/zoo/
[1] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. Cambridge University Press, 2010.

grove.phaseestimation.phase_estimation

References

Grover's Search Algorithm

Overview

Quantum search algorithm via amplitude amplification.

Source Code Docs

Here you can find documentation for the different submodules in phaseestimation.

grove.grover.grover

CHAPTER 2

Indices and Tables

- genindex
- modindex
- search