Knapsack VQE

August 10, 2021

0.1 Solving the Knapsack problem with Variational Quantum Eigensolver Algorithm

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

[3]:
```
# Importing libaries
from qiskit_optimization.applications import Knapsack
from qiskit.algorithms import NumPyMinimumEigensolver
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit.utils import QuantumInstance
from qiskit import Aer
from qiskit_optimization.converters import QuadraticProgramToQubo
from qiskit.algorithms import VQE
from qiskit.algorithms.optimizers import SPSA
from qiskit.circuit.library import EfficientSU2, TwoLocal
import numpy as np
from qiskit import QuantumCircuit, execute, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit_textbook.tools import array_to_latex

# Loading IBM Quantum account
provider = IBMQ.load_account()

print("All libraries imported successfully!")
```

```
All libraries imported successfully!
```

0.2 Setting up the problem

[4]:
```
# Defining the knapsack problem. We define a list of values, a list of weights,␣
↪the maximum weight, and then
# put it all together into a variable called `problem' using the Knapsack␣
↪function.

# list of the values of items
values = [4, 2, 5, 10]

# list of the weights of items
weights = [3, 1, 2, 6]

# maximum knapsack capacity
max_weight = 10

knapsack_problem = Knapsack(values = values, weights = weights, max_weight =␣
↪max_weight)
```

0.3 Using a classical method to solve the problem

[5]:
```
# Since this is a small problem,we can check the answer we will get later with␣
↪VQE using a classical solver.
# Later, we will use VQE instead of the classical NumPyMinimumEigensolver.

method = NumPyMinimumEigensolver()
```

[6]:
```
# Using the classical solver NumPyMinimumEigensolver

calc = MinimumEigenOptimizer(method)
result = calc.solve(knapsack_problem.to_quadratic_program())
print('result:\n', result)
print('\nsolution:\n', knapsack_problem.interpret(result))
```

```
result:
 optimal function value: 17.0
optimal value: [0. 1. 1. 1.]
status: SUCCESS

solution:
 [1, 2, 3]
```

0.4 Using VQE to solve the problem

[8]:
```
# Converting the knapsack problem to a quantum circuit,

operator, offset = QuadraticProgramToQubo().convert(knapsack_problem.
 ↪to_quadratic_program()).to_ising()
print("number of qubits =", operator.num_qubits)
```

```
number of qubits = 8
```

[9]:
```
# Setting up VQE
# We tell our code to use VQE with the tunable circuit, the optimizer, and the␣
 ↪quantum instance

qinstance = QuantumInstance(backend=Aer.get_backend('qasm_simulator'),␣
 ↪shots=1000)

tunable_circuit = EfficientSU2(operator.num_qubits, reps=2, entanglement='full')
tunable_circuit = TwoLocal(operator.num_qubits, rotation_blocks = ['h', 'rx'],␣
 ↪entanglement_blocks = 'cz', reps=3, entanglement='full')
optimizer = SPSA(maxiter=15)

method = VQE(ansatz = tunable_circuit, optimizer = optimizer, quantum_instance␣
 ↪= qinstance)
```

[10]:
```
# Running VQE and printing results

calc = MinimumEigenOptimizer(method)
result = calc.solve(knapsack_problem.to_quadratic_program())
print('result:\n', result)
print('\nsolution:\n', knapsack_problem.interpret(result))
print('\ntime:', result.min_eigen_solver_result.optimizer_time)
```

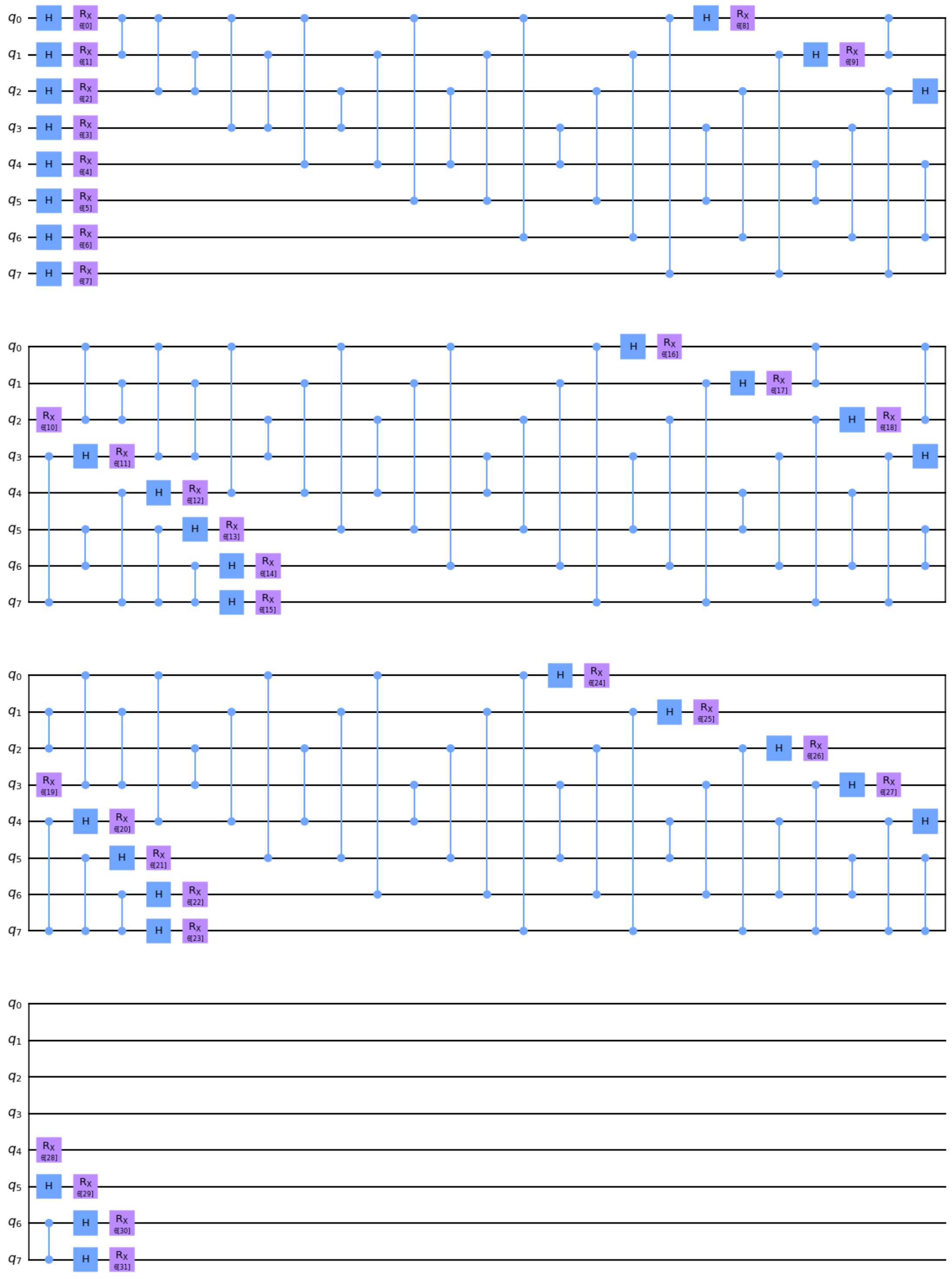
```
result:
 optimal function value: 17.0
optimal value: [0. 1. 1. 1.]
status: SUCCESS

solution:
 [1, 2, 3]

time: 8.660624027252197
```

[11]:
```
# Drawing the tunable quantum circuit to see how exactly the algorithm was run
tunable_circuit.decompose().draw()
```

[11]:


q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
H
R_X
θ[0] … θ[31]