QUBO_VQE

November 18, 2022

1 Variational Quantum Optimization using CVaR

[1]:
```
import gc
import time
```

[2]:
```
#disable garbage collector
gc.disable()
```

[3]:
```
start_counter_ns = time.perf_counter_ns()
```

1.1 Introduction

This notebook shows how to use the Conditional Value at Risk (CVaR) objective function introduced in [1] within the variational quantum optimization algorithms provided by Qiskit. Particularly, it is shown how to setup the `MinimumEigenOptimizer` using `VQE` accordingly. For a given set of shots with corresponding objective values of the considered optimization problem, the CVaR with confidence level $\alpha \in [0, 1]$ is defined as the average of the α best shots. Thus, $\alpha = 1$ corresponds to the standard expected value, while $\alpha = 0$ corresponds to the minimum of the given shots, and $\alpha \in (0, 1)$ is a tradeoff between focusing on better shots, but still applying some averaging to smoothen the optimization landscape.

1.2 References

[1] P. Barkoutsos et al., *Improving Variational Quantum Optimization using CVaR,* Quantum 4, 256 (2020).

[4]:
```
from qiskit.circuit.library import RealAmplitudes
from qiskit.algorithms.optimizers import COBYLA
from qiskit.algorithms import NumPyMinimumEigensolver, VQE
from qiskit.opflow import PauliExpectation, CVaRExpectation
from qiskit_optimization import QuadraticProgram
from qiskit_optimization.converters import LinearEqualityToPenalty
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit import execute, Aer, BasicAer
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit.algorithms import QAOA, NumPyMinimumEigensolver
from qiskit_optimization.algorithms import (
    MinimumEigenOptimizer,
```

```
    RecursiveMinimumEigenOptimizer,
    SolutionSample,
    OptimizationResultStatus,
)
from qiskit.visualization import plot_histogram
from typing import List, Tuple
import numpy as np
import matplotlib.pyplot as plt
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning:
scipy._lib.messagestream.MessageStream size changed, may indicate binary
incompatibility. Expected 56 from C header, got 64 from PyObject
```

[5]:
```
algorithm_globals.random_seed = 123456
n = 4
```

1.3 Converting a QUBO to an Operator

2 prepare problem instance

n = 4 # number of assets

[6]:
```
# create a QUBO
qubo = QuadraticProgram()
qubo.binary_var("Z1")
qubo.binary_var("Z2")
qubo.binary_var("Z3")
qubo.binary_var("Z4")
qubo.minimize(linear=[-5, -3, -8, -6], quadratic={("Z1", "Z2"): 4, ("Z1", "Z3"):
 ↪ 8, ("Z2", "Z3"): 2, ("Z3", "Z4"): 10})
print(qubo.prettyprint())
```

```
Problem name:

Minimize
  4*Z1*Z2 + 8*Z1*Z3 + 2*Z2*Z3 + 10*Z3*Z4 - 5*Z1 - 3*Z2 - 8*Z3 - 6*Z4

Subject to
  No constraints

  Binary variables (4)
    Z1 Z2 Z3 Z4
```

Next we translate this QUBO into an Ising operator. This results not only in an `Operator` but also in a constant offset to be taken into account to shift the resulting value.

[7]:
```
op, offset = qubo.to_ising()
print("offset: {}".format(offset))
print("operator:")
print(op)
```

```
offset: -5.0
operator:
-0.5 * IIIZ
- 1.0 * IZII
+ 0.5 * ZIII
+ 1.0 * IIZZ
+ 2.0 * IZIZ
+ 0.5 * IZZI
+ 2.5 * ZZII
```

Sometimes a `QuadraticProgram` might also directly be given in the form of an `Operator`. For such cases, Qiskit also provides a translator from an `Operator` back to a `QuadraticProgram`, which we illustrate in the following.

[8]:
```
qp = QuadraticProgram()
qp.from_ising(op, offset, linear=True)
print(qp.prettyprint())
```

```
Problem name:

Minimize
  4*x0*x1 + 8*x0*x2 + 2*x1*x2 + 10*x2*x3 - 5*x0 - 3*x1 - 8*x2 - 6*x3

Subject to
  No constraints

  Binary variables (4)
    x0 x1 x2 x3
```

[9]:
```
# solve classically as reference
opt_result = MinimumEigenOptimizer(NumPyMinimumEigensolver()).solve(qp)
print(opt_result.prettyprint())
```

```
objective function value: -11.0
variable values: x0=1.0, x1=0.0, x2=0.0, x3=1.0
status: SUCCESS
```

This translator allows, for instance, one to translate an `Operator` to a `QuadraticProgram` and then solve the problem with other algorithms that are not based on the Ising Hamiltonian representation, such as the `GroverOptimizer`.

2.1 Minimum Eigen Optimizer using VQE

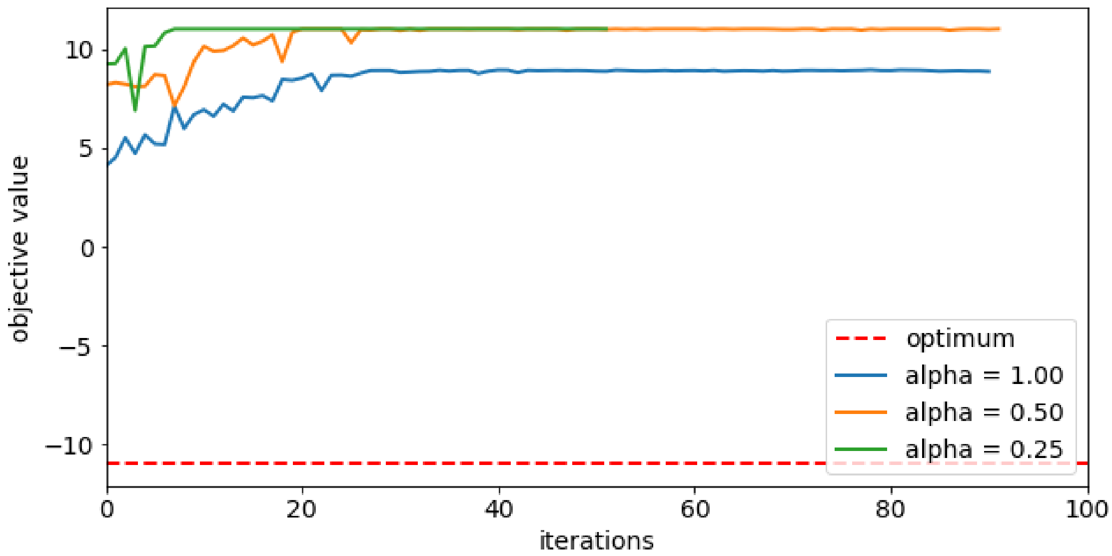
[10]:
```
# set classical optimizer
maxiter = 100
optimizer = COBYLA(maxiter=maxiter)

# set variational ansatz
ansatz = RealAmplitudes(n, reps=1)
m = ansatz.num_parameters

# set backend
backend_name = "qasm_simulator"  # use this for QASM simulator
# backend_name = 'aer_simulator_statevector'  # use this for statevector␣
↪simlator
backend = Aer.get_backend(backend_name)

# run variational optimization for different values of alpha
alphas = [1.0, 0.50, 0.25]  # confidence levels to be evaluated
```

[11]:
```
# dictionaries to store optimization progress and results
objectives = {alpha: [] for alpha in alphas}  # set of tested objective␣
↪functions w.r.t. alpha
results = {}  # results of minimum eigensolver w.r.t alpha

# callback to store intermediate results
def callback(i, params, obj, stddev, alpha):
    # we translate the objective from the internal Ising representation
    # to the original optimization problem
    objectives[alpha] += [-(obj + offset)]


# loop over all given alpha values
for alpha in alphas:

    # initialize CVaR_alpha objective
    cvar_exp = CVaRExpectation(alpha, PauliExpectation())
    cvar_exp.compute_variance = lambda x: [0]  # to be fixed in PR #1373

    # initialize VQE using CVaR
    vqe = VQE(
        expectation=cvar_exp,
        optimizer=optimizer,
        ansatz=ansatz,
        quantum_instance=backend,
        callback=lambda i, params, obj, stddev: callback(i, params, obj,␣
↪stddev, alpha),
    )
```

```python
    # initialize optimization algorithm based on CVaR-VQE
    opt_alg = MinimumEigenOptimizer(vqe)

    # solve problem
    results[alpha] = opt_alg.solve(qp)

    # print results
    print("alpha = {}:".format(alpha))
    print(results[alpha].prettyprint())
    print()
```

```
alpha = 1.0:
objective function value: -9.0
variable values: x0=0.0, x1=1.0, x2=1.0, x3=0.0
status: SUCCESS

alpha = 0.5:
objective function value: -11.0
variable values: x0=1.0, x1=0.0, x2=0.0, x3=1.0
status: SUCCESS

alpha = 0.25:
objective function value: -11.0
variable values: x0=1.0, x1=0.0, x2=0.0, x3=1.0
status: SUCCESS
```

[12]:
```python
# plot resulting history of objective values
plt.figure(figsize=(10, 5))
plt.plot([0, maxiter], [opt_result.fval, opt_result.fval], "r--", linewidth=2, label="optimum")
for alpha in alphas:
    plt.plot(objectives[alpha], label="alpha = %.2f" % alpha, linewidth=2)
plt.legend(loc="lower right", fontsize=14)
plt.xlim(0, maxiter)
plt.xticks(fontsize=14)
plt.xlabel("iterations", fontsize=14)
plt.yticks(fontsize=14)
plt.ylabel("objective value", fontsize=14)
plt.show()
```



[13]:
```
# evaluate and sort all objective values
objective_values = np.zeros(2**n)
for i in range(2**n):
    x_bin = ("{0:0%sb}" % n).format(i)
    x = [0 if x_ == "0" else 1 for x_ in reversed(x_bin)]
    objective_values[i] = qp.objective.evaluate(x)
ind = np.argsort(objective_values)

# evaluate final optimal probability for each alpha
probabilities = np.zeros(len(objective_values))
for alpha in alphas:
    if backend_name == "qasm_simulator":
        counts = results[alpha].min_eigen_solver_result.eigenstate
        shots = sum(counts.values())
        for key, val in counts.items():
            i = int(key, 2)
            probabilities[i] = val / shots
    else:
        probabilities = np.abs(results[alpha].min_eigen_solver_result.
 ↪eigenstate) ** 2
    print("optimal probabilitiy (alpha = %.2f):  %.4f" % (alpha,␣
 ↪probabilities[ind][-1:]))
```

```
optimal probabilitiy (alpha = 1.00):  0.0000
optimal probabilitiy (alpha = 0.50):  0.0817
optimal probabilitiy (alpha = 0.25):  0.0817
```

[14]:
```
exact_mes = NumPyMinimumEigensolver()
```

Then, we use the `MinimumEigensolver` to create `MinimumEigenOptimizer`.

[15]:
```
exact_alg = MinimumEigenOptimizer(exact_mes)  # using the exact classical numpy␣
↪minimum eigen solver
```

We first use the `MinimumEigenOptimizer` based on the classical exact `NumPyMinimumEigensolver` to get the optimal benchmark solution for this small example.

[16]:
```
exact_result = exact_alg.solve(qubo)
print(exact_result.prettyprint())
```

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```

Next we apply the `MinimumEigenOptimizer` based on `QAOA` to the same problem.

[17]:
```
vqe_result = opt_alg.solve(qubo)
print(vqe_result.prettyprint())
```

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```

2.1.1 Analysis of Samples

`OptimizationResult` provides useful information in the form of `SolutionSamples` (here denoted as *samples*). Each `SolutionSample` contains information about the input values (`x`), the corresponding objective function value (`fval`), the fraction of samples corresponding to that input (`probability`), and the solution `status` (`SUCCESS`, `FAILURE`, `INFEASIBLE`). Multiple samples corresponding to the same input are consolidated into a single `SolutionSample` (with its `probability` attribute being the aggregate fraction of samples represented by that `SolutionSample`).

[18]:
```
print("variable order:", [var.name for var in vqe_result.variables])
for s in vqe_result.samples:
    print(s)
```

```
variable order: ['Z1', 'Z2', 'Z3', 'Z4']
SolutionSample(x=array([1., 0., 0., 1.]), fval=-11.0,
probability=0.29687500000000006, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.10253906249999999, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 0.]), fval=-9.0,
probability=0.0019531250000000004, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.37207031250000006, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0, probability=0.0458984375,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0, probability=0.0263671875,
status=<OptimizationResultStatus.SUCCESS: 0>)
```

```
SolutionSample(x=array([1., 0., 1., 0.]), fval=-5.0,
probability=0.0019531250000000004, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.04980468750000001, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.014648437500000002, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 1.]), fval=-4.0, probability=0.00390625,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0, probability=0.0400390625,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.027343750000000003, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 0.]), fval=0.0, probability=0.00390625,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.012695312499999998, status=<OptimizationResultStatus.SUCCESS: 0>)
```

We may also want to filter samples according to their status or probabilities.

[19]:
```python
def get_filtered_samples(
    samples: List[SolutionSample],
    threshold: float = 0,
    allowed_status: Tuple[OptimizationResultStatus] = (OptimizationResultStatus.
 ↪SUCCESS,),
):
    res = []
    for s in samples:
        if s.status in allowed_status and s.probability > threshold:
            res.append(s)

    return res
```

[20]:
```python
filtered_samples = get_filtered_samples(
    vqe_result.samples, threshold=0.005,␣
 ↪allowed_status=(OptimizationResultStatus.SUCCESS,)
)
for s in filtered_samples:
    print(s)
```

```
SolutionSample(x=array([1., 0., 0., 1.]), fval=-11.0,
probability=0.29687500000000006, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.10253906249999999, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.37207031250000006, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0, probability=0.0458984375,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0, probability=0.0263671875,
```

```
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.04980468750000001, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.014648437500000002, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0, probability=0.0400390625,
status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.027343750000000003, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.012695312499999998, status=<OptimizationResultStatus.SUCCESS: 0>)
```

If we want to obtain a better perspective of the results, statistics is very helpful, both with respect to the objective function values and their respective probabilities. Thus, mean and standard deviation are the very basics for understanding the results.

[21]:
```
fvals = [s.fval for s in vqe_result.samples]
probabilities = [s.probability for s in vqe_result.samples]
```

[22]:
```
np.mean(fvals)
```

[22]:
```
-5.0
```

[23]:
```
np.std(fvals)
```

[23]:
```
3.6839419880650364
```

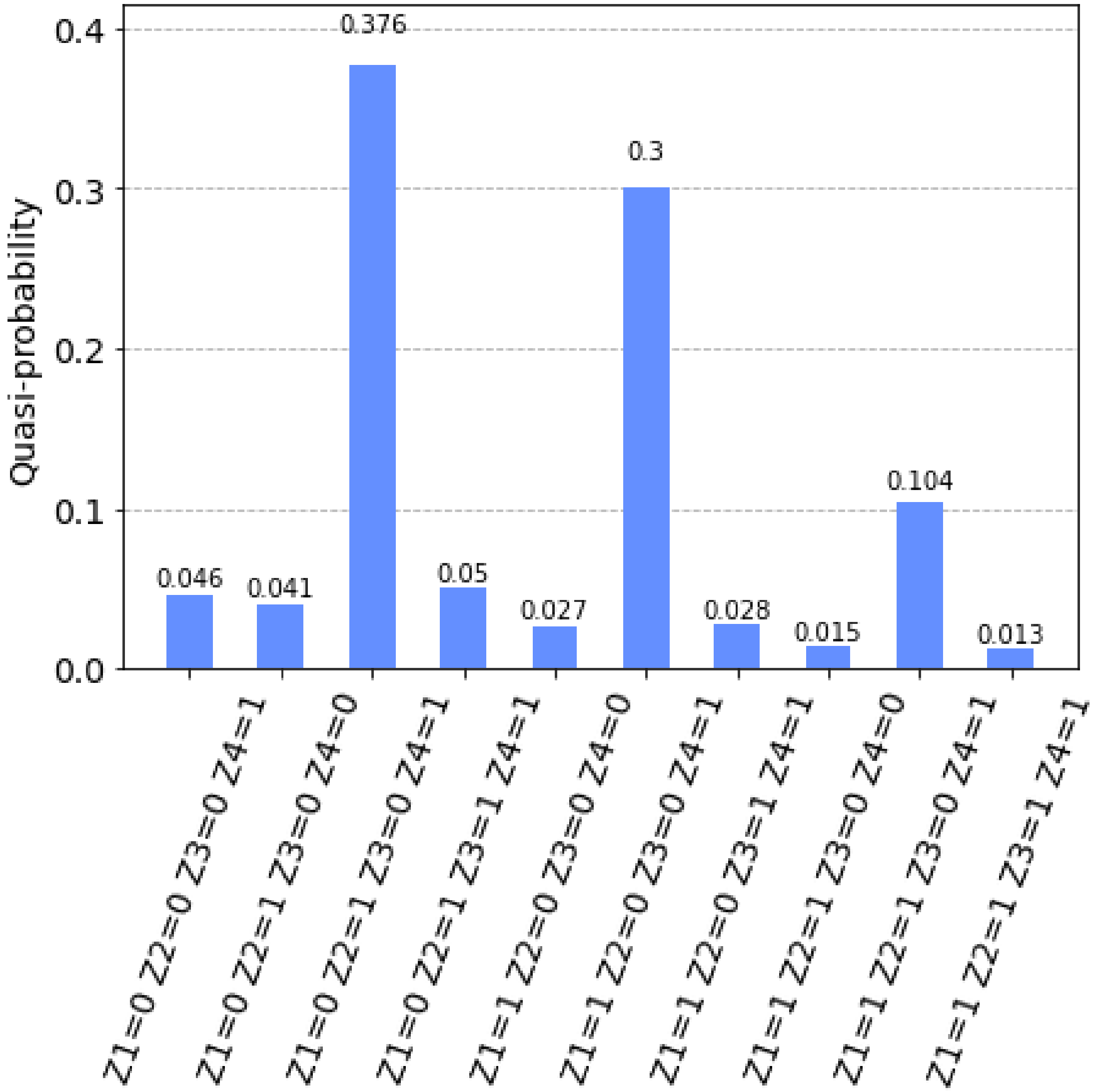
Finally, despite all the number-crunching, visualization is usually the best early-analysis approach.

[24]:
```
samples_for_plot = {
    " ".join(f"{vqe_result.variables[i].name}={int(v)}" for i, v in enumerate(s.
 ↪x)): s.probability
    for s in filtered_samples
}
samples_for_plot
```

[24]:
```
{'Z1=1 Z2=0 Z3=0 Z4=1': 0.29687500000000006,
 'Z1=1 Z2=1 Z3=0 Z4=1': 0.10253906249999999,
 'Z1=0 Z2=1 Z3=0 Z4=1': 0.37207031250000006,
 'Z1=0 Z2=0 Z3=0 Z4=1': 0.0458984375,
 'Z1=1 Z2=0 Z3=0 Z4=0': 0.0263671875,
 'Z1=0 Z2=1 Z3=1 Z4=1': 0.04980468750000001,
 'Z1=1 Z2=1 Z3=0 Z4=0': 0.014648437500000002,
 'Z1=0 Z2=1 Z3=0 Z4=0': 0.0400390625,
 'Z1=1 Z2=0 Z3=1 Z4=1': 0.027343750000000003,
 'Z1=1 Z2=1 Z3=1 Z4=1': 0.012695312499999998}
```

[25]:
```
plot_histogram(samples_for_plot)
```

[25]:



RecursiveMinimumEigenOptimizer

The `RecursiveMinimumEigenOptimizer` takes a `MinimumEigenOptimizer` as input and applies the recursive optimization scheme to reduce the size of the problem one variable at a time. Once the size of the generated intermediate problem is below a given threshold (`min_num_vars`), the `RecursiveMinimumEigenOptimizer` uses another solver (`min_num_vars_optimizer`), e.g., an exact classical solver such as CPLEX or the `MinimumEigenOptimizer` based on the `NumPyMinimumEigensolver`.

In the following, we show how to use the `RecursiveMinimumEigenOptimizer` using the two `MinimumEigenOptimizer`s introduced before.

First, we construct the `RecursiveMinimumEigenOptimizer` such that it reduces the problem size from 3 variables to 1 variable and then uses the exact solver for the last variable. Then we call solve to optimize the considered problem.

[26]:
```
rqaoa = RecursiveMinimumEigenOptimizer(opt_alg, min_num_vars=1,␣
↪min_num_vars_optimizer=exact_alg)
```

[27]:
```
rqaoa_result = rqaoa.solve(qubo)
print(rqaoa_result.prettyprint())
```

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```

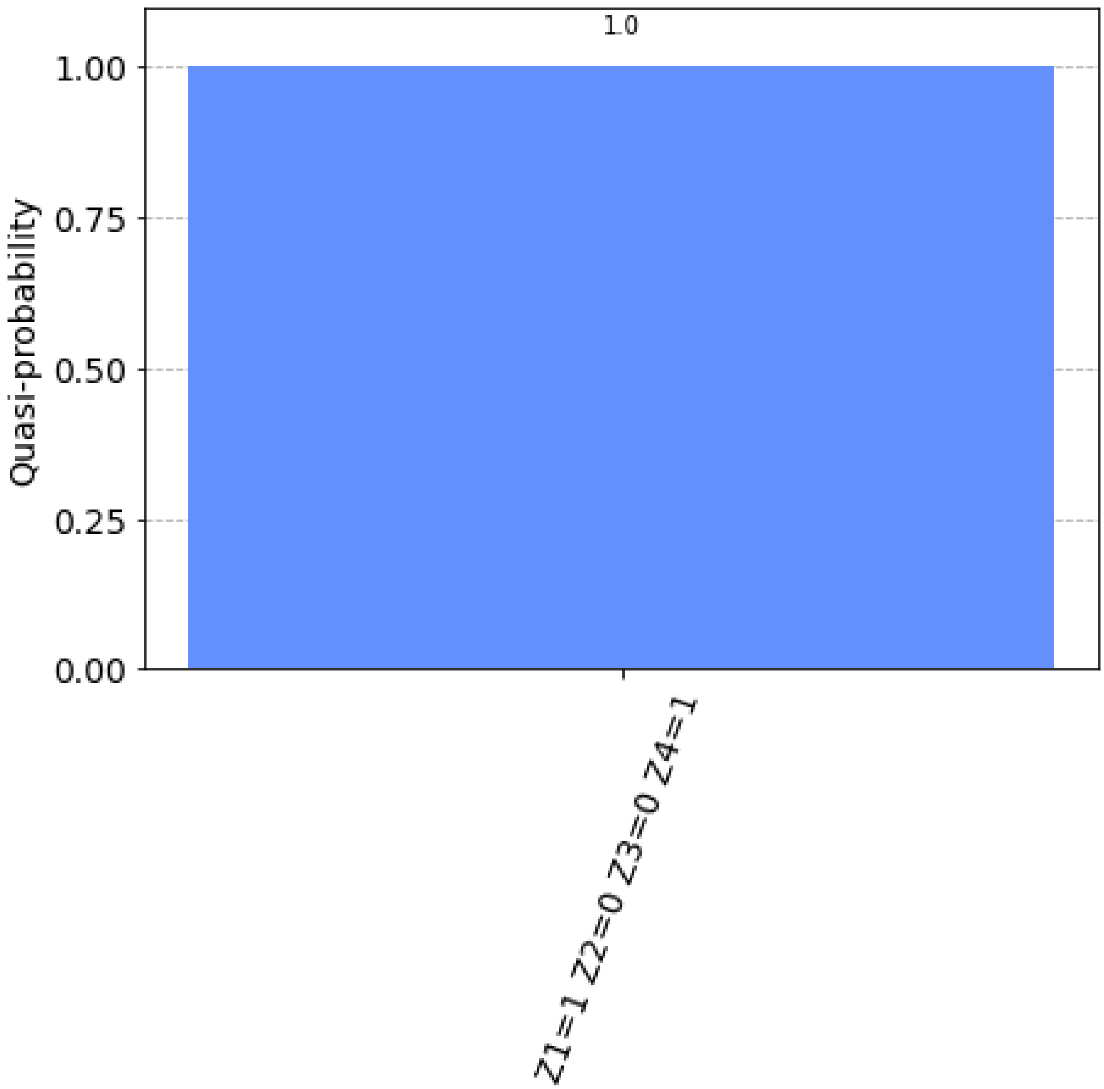
[28]:
```
filtered_samples = get_filtered_samples(
    rqaoa_result.samples, threshold=0.005,␣
↪allowed_status=(OptimizationResultStatus.SUCCESS,)
)
```

[29]:
```
samples_for_plot = {
    " ".join(f"{rqaoa_result.variables[i].name}={int(v)}" for i, v in␣
↪enumerate(s.x)): s.probability
    for s in filtered_samples
}
samples_for_plot
```

[29]:
```
{'Z1=1 Z2=0 Z3=0 Z4=1': 1.0}
```

[30]:
```
plot_histogram(samples_for_plot)
```

[30]:



[31]:
```
end_counter_ns = time.perf_counter_ns()
```

[32]:
```
# re-enable garbage collector
gc.enable()
```

[33]:
```
timer_ns = end_counter_ns - start_counter_ns
print("Average Execution time:",timer_ns)
```

```
Average Execution time: 11089415282
```

[34]:
```
import qiskit.tools.jupyter

%qiskit_version_table
```

```
%qiskit_copyright
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

[ ]:

```

```