QUBO_QAOA

November 18, 2022

1 Minimum Eigen Optimizer

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

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

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

1.1 Introduction

An interesting class of optimization problems to be addressed by quantum computing are Quadratic Unconstrained Binary Optimization (QUBO) problems. Finding the solution to a QUBO is equivalent to finding the ground state of a corresponding Ising Hamiltonian, which is an important problem not only in optimization, but also in quantum chemistry and physics. For this translation, the binary variables taking values in $\{0, 1\}$ are replaced by spin variables taking values in $\{-1, +1\}$, which allows one to replace the resulting spin variables by Pauli Z matrices, and thus, an Ising Hamiltonian. For more details on this mapping we refer to [1].

Qiskit provides automatic conversion from a suitable `QuadraticProgram` to an Ising Hamiltonian, which then allows leveraging all the `MinimumEigenSolver` implementations, such as

- `VQE`,
- `QAOA`, or
- `NumpyMinimumEigensolver` (classical exact method).

Qiskit Optimization provides a the `MinimumEigenOptimizer` class, which wraps the translation to an Ising Hamiltonian (in Qiskit Terra also called `Operator`), the call to a `MinimumEigensolver`, and the translation of the results back to an `OptimizationResult`.

In the following we first illustrate the conversion from a `QuadraticProgram` to an `Operator` and then show how to use the `MinimumEigenOptimizer` with different `MinimumEigensolvers` to solve a given `QuadraticProgram`. The algorithms in Qiskit automatically try to convert a given problem to the supported problem class if possible, for instance, the `MinimumEigenOptimizer` will automatically translate integer variables to binary variables or add linear equality constraints as a quadratic penalty term to the objective. It should be mentioned that a `QiskitOptimizationError` will be thrown if conversion of a quadratic program with integer variables is attempted.

The circuit depth of `QAOA` potentially has to be increased with the problem size, which might be prohibitive for near-term quantum devices. A possible workaround is Recursive QAOA, as introduced in [2]. Qiskit generalizes this concept to the `RecursiveMinimumEigenOptimizer`, which is introduced at the end of this tutorial.

1.1.1 References

[1] A. Lucas, *Ising formulations of many NP problems,* Front. Phys., 12 (2014).

[2] S. Bravyi, A. Kliesch, R. Koenig, E. Tang, *Obstacles to State Preparation and Variational Optimization from Symmetry Protection,* arXiv preprint arXiv:1910.08980 (2019).

1.2 Converting a QUBO to an Operator

[4]:
```
from qiskit import 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_optimization import QuadraticProgram
from qiskit.visualization import plot_histogram
from typing import List, Tuple
import numpy as np
```

```
<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]:
```
# 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.

[6]:
```
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.

[7]:
```
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
```

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`.

1.3 Solving a QUBO with the MinimumEigenOptimizer

We start by initializing the `MinimumEigensolver` we want to use.

[8]:
```
algorithm_globals.random_seed = 10598
quantum_instance = QuantumInstance(
    BasicAer.get_backend("statevector_simulator"),
    seed_simulator=algorithm_globals.random_seed,
    seed_transpiler=algorithm_globals.random_seed,
)
qaoa_mes = QAOA(quantum_instance=quantum_instance, initial_point=[0.0, 0.0])
exact_mes = NumPyMinimumEigensolver()
```

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

[9]:
```
qaoa = MinimumEigenOptimizer(qaoa_mes)  # using QAOA
exact = 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.

[10]:
```
exact_result = exact.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.

[11]:
```
qaoa_result = qaoa.solve(qubo)
print(qaoa_result.prettyprint())
```

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

1.3.1 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`).

[12]:
```
print("variable order:", [var.name for var in qaoa_result.variables])
for s in qaoa_result.samples:
    print(s)
```

```
variable order: ['Z1', 'Z2', 'Z3', 'Z4']
SolutionSample(x=array([1., 0., 0., 1.]), fval=-11.0,
```

```
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 0.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 0.]), fval=-8.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 1.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 0.]), fval=-2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 0.]), fval=0.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
```

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

[13]:
```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
```

[14]:
```
filtered_samples = get_filtered_samples(
    qaoa_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=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 0.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 0.]), fval=-8.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 1.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 0.]), fval=-2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 0.]), fval=0.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.06249999999999996, 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.

[15]:
```
fvals = [s.fval for s in qaoa_result.samples]
probabilities = [s.probability for s in qaoa_result.samples]
```

[16]:
```
np.mean(fvals)
```

[16]: -5.0

[17]:
```
np.std(fvals)
```

[17]: 3.605551275463989

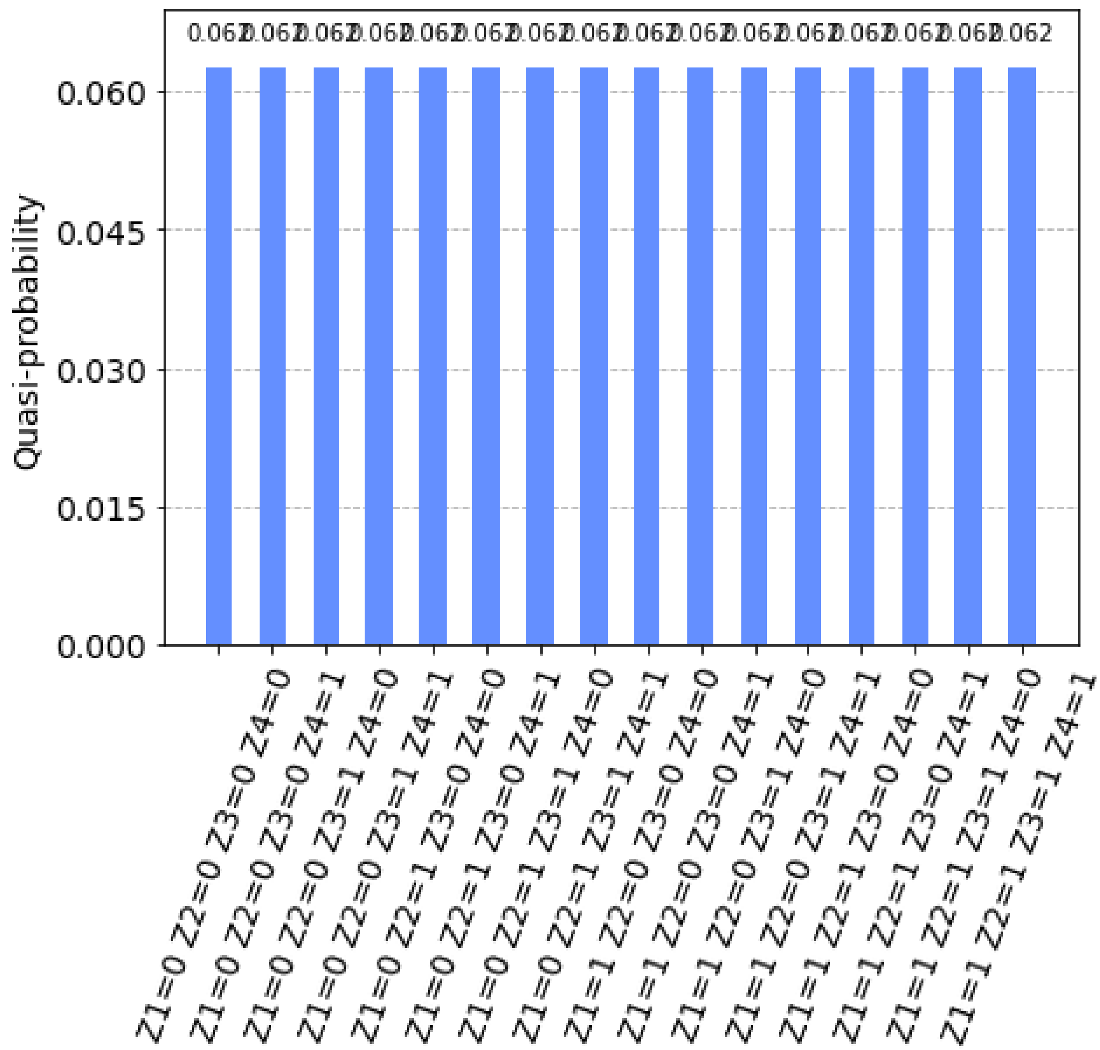
Finally, despite all the number-crunching, visualization is usually the best early-analysis approach.

[18]:
```
samples_for_plot = {
    " ".join(f"{qaoa_result.variables[i].name}={int(v)}" for i, v in␣
 ↪enumerate(s.x)): s.probability
    for s in filtered_samples
}
samples_for_plot
```

[18]:
```
{'Z1=1 Z2=0 Z3=0 Z4=1': 0.06249999999999996,
 'Z1=1 Z2=1 Z3=0 Z4=1': 0.06249999999999996,
 'Z1=0 Z2=1 Z3=1 Z4=0': 0.06249999999999996,
 'Z1=0 Z2=1 Z3=0 Z4=1': 0.06249999999999996,
 'Z1=0 Z2=0 Z3=1 Z4=0': 0.06249999999999996,
 'Z1=0 Z2=0 Z3=0 Z4=1': 0.06249999999999996,
 'Z1=1 Z2=0 Z3=0 Z4=0': 0.06249999999999996,
 'Z1=1 Z2=0 Z3=1 Z4=0': 0.06249999999999996,
 'Z1=0 Z2=1 Z3=1 Z4=1': 0.06249999999999996,
 'Z1=1 Z2=1 Z3=0 Z4=0': 0.06249999999999996,
 'Z1=0 Z2=0 Z3=1 Z4=1': 0.06249999999999996,
 'Z1=0 Z2=1 Z3=0 Z4=0': 0.06249999999999996,
 'Z1=1 Z2=1 Z3=1 Z4=0': 0.06249999999999996,
 'Z1=1 Z2=0 Z3=1 Z4=1': 0.06249999999999996,
 'Z1=0 Z2=0 Z3=0 Z4=0': 0.06249999999999996,
 'Z1=1 Z2=1 Z3=1 Z4=1': 0.06249999999999996}
```

[19]:
```
plot_histogram(samples_for_plot)
```

[19]:



1.4 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.

[20]:
```
rqaoa = RecursiveMinimumEigenOptimizer(qaoa, min_num_vars=1,␣
↪min_num_vars_optimizer=exact)
```

[21]:
```
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
```

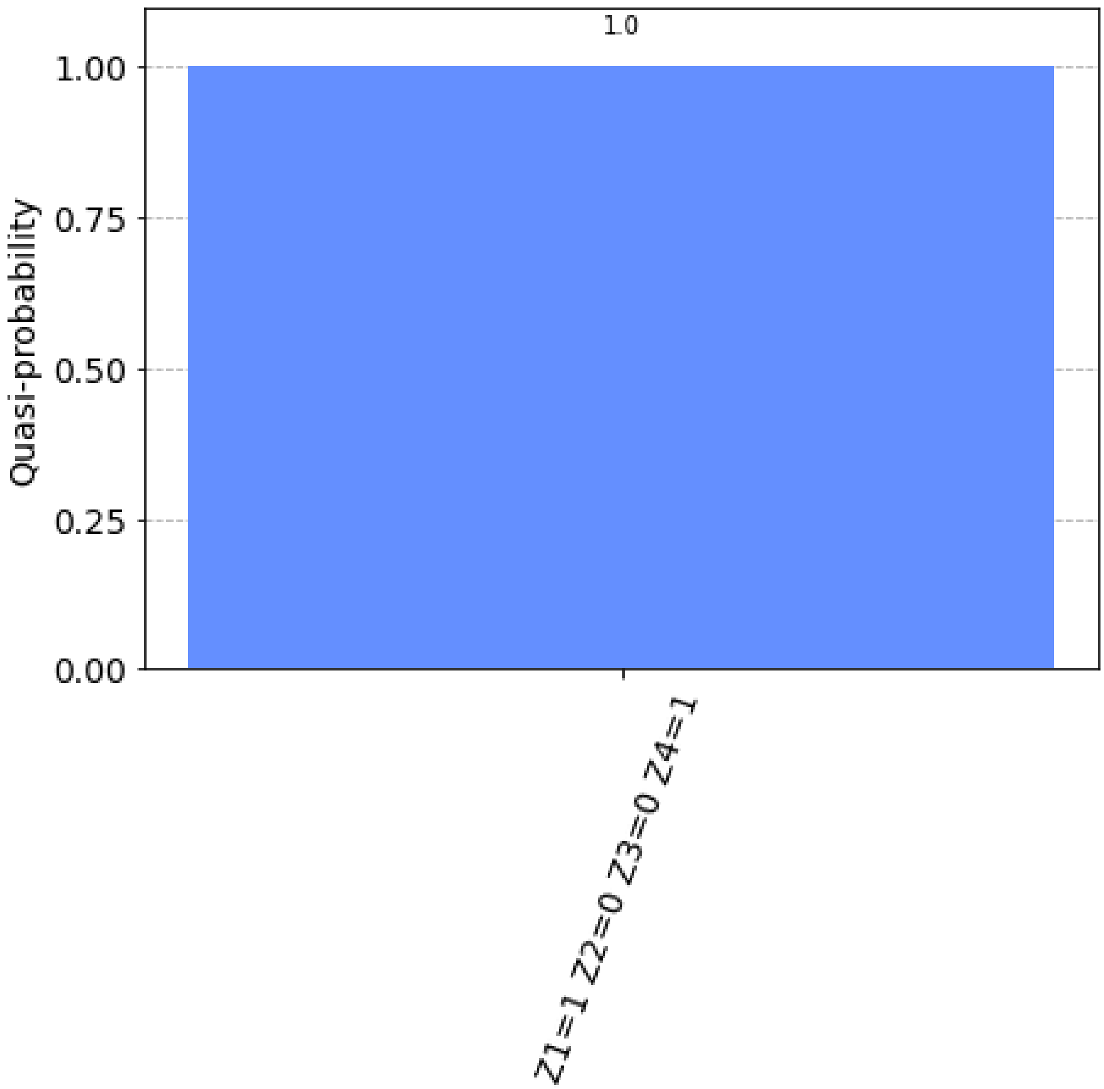
[22]:
```
filtered_samples = get_filtered_samples(
    rqaoa_result.samples, threshold=0.005,␣
↪allowed_status=(OptimizationResultStatus.SUCCESS,)
)
```

[23]:
```
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
```

[23]:
```
{'Z1=1 Z2=0 Z3=0 Z4=1': 1.0}
```

[24]:
```
plot_histogram(samples_for_plot)
```

[24]:



[25]:
```
end_counter_ns = time.perf_counter_ns()
```

[26]:
```
# re-enable garbage collector
gc.enable()
```

[27]:
```
timer_ns = end_counter_ns - start_counter_ns
print("Average Execution time:",timer_ns)
```

```
Average Execution time: 6776381187
```

[28]:
```
import qiskit.tools.jupyter

%qiskit_version_table
```

```
%qiskit_copyright
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

```

```