Grover's Algorithm

In this section, we introduce Grover's algorithm and how it can be used to solve unstructured search problems. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

## **Contents**

1. [Introduction](https://qiskit.org/textbook/ch-algorithms/grover.html#introduction)
2. [Example: 2 Qubits](https://qiskit.org/textbook/ch-algorithms/grover.html#2qubits)2.1 [Simulation](https://qiskit.org/textbook/ch-algorithms/grover.html#2qubits-simulation)2.2 [Device](https://qiskit.org/textbook/ch-algorithms/grover.html#2qubits-device)
3. [Example: 3 Qubits](https://qiskit.org/textbook/ch-algorithms/grover.html#3qubits)3.1 [Simulation](https://qiskit.org/textbook/ch-algorithms/grover.html#3qubits-simulation)3.2 [Device](https://qiskit.org/textbook/ch-algorithms/grover.html#3qubits-device)
4. [Problems](https://qiskit.org/textbook/ch-algorithms/grover.html#problems)
5. [Solving Sudoku using Grover's Algorithm](https://qiskit.org/textbook/ch-algorithms/grover.html#sudoku)
6. [References](https://qiskit.org/textbook/ch-algorithms/grover.html#references)

## **1. Introduction**

You have likely heard that one of the many advantages a quantum computer has over a classical computer is its superior speed searching databases. Grover's algorithm demonstrates this capability. This algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms. This is called the amplitude amplification trick.

### **Unstructured Search**

Suppose you are given a large list of

N

N

items. Among these items there is one item with a unique property that we wish to locate; we will call this one the winner

w

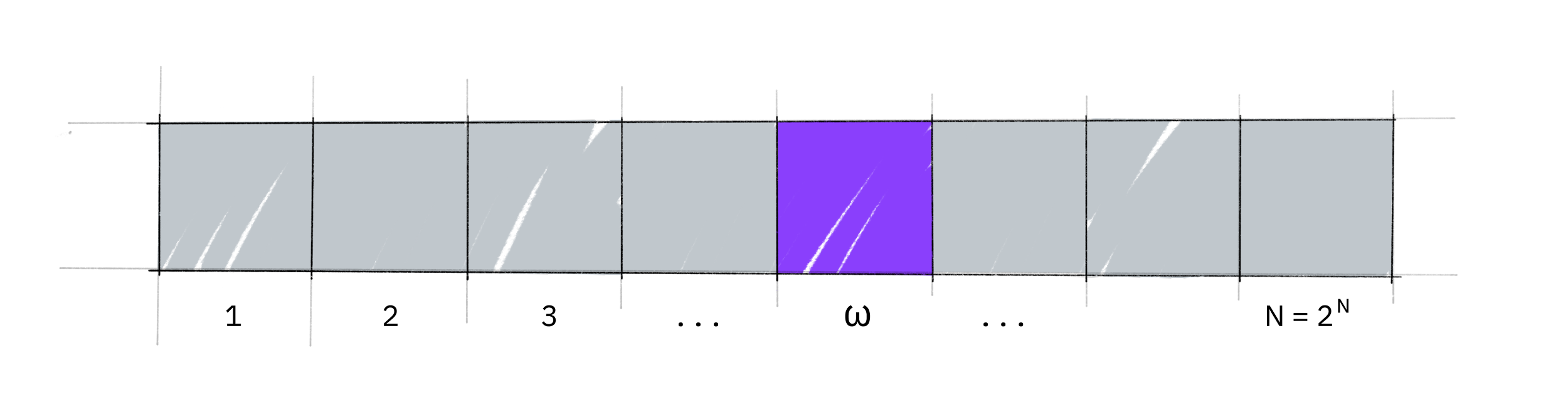
w

. Think of each item in the list as a box of a particular color. Say all items in the list are gray except the winner

w

w

, which is purple.



To find the purple box -- the *marked item* -- using classical computation, one would have to check on average

N

/

2

N/2

of these boxes, and in the worst case, all

N

N

of them. On a quantum computer, however, we can find the marked item in roughly

√

N

N

steps with Grover's amplitude amplification trick. A quadratic speedup is indeed a substantial time-saver for finding marked items in long lists. Additionally, the algorithm does not use the list's internal structure, which makes it *generic;* this is why it immediately provides a quadratic quantum speed-up for many classical problems.

### **Creating an Oracle**

For the examples in this textbook, our 'database' is comprised of all the possible computational basis states our qubits can be in. For example, if we have 3 qubits, our list is the states

|

000

⟩

,

|

001

⟩

,

…

|

111

⟩

|000⟩,|001⟩,…|111⟩

(i.e the states

|

0

⟩

→

|

7

⟩

|0⟩→|7⟩

).

Grover’s algorithm solves oracles that add a negative phase to the solution states. I.e. for any state

|

x

⟩

|x⟩

in the computational basis:

U

ω

|

x

⟩

=

{

|

x

⟩

if

x

≠

ω

−

|

x

⟩

if

x

=

ω

Uω|x⟩={−|x⟩ifx≠ω−|x⟩ifx=ω

This oracle will be a diagonal matrix, where the entry that correspond to the marked item will have a negative phase. For example, if we have three qubits and

ω

=

101

ω=101

, our oracle will have the matrix:

U

ω

=

⎡

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎢

⎣

1

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

0

−

1

0

0

0

0

0

0

0

0

1

0

0

0

0

0

0

0

0

1

⎤

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎥

⎦

←

ω

=

101

Uω=[100000000100000000100000000100000000100000000−1000000001000000001]←ω=101

What makes Grover’s algorithm so powerful is how easy it is to convert a problem to an oracle of this form. There are many computational problems in which it’s difficult to *find* a solution, but relatively easy to *verify* a solution. For example, we can easily verify a solution to a [sudoku](https://en.wikipedia.org/wiki/Sudoku) by checking all the rules are satisfied. For these problems, we can create a function

f

f

that takes a proposed solution

x

x

, and returns

f

(

x

)

=

0

f(x)=0

if

x

x

is not a solution (

x

≠

ω

x≠ω

) and

f

(

x

)

=

1

f(x)=1

for a valid solution (

x

=

ω

x=ω

). Our oracle can then be described as:

U

ω

|

x

⟩

=

(

−

1

)

f

(

x

)

|

x

⟩

Uω|x⟩=(−1)f(x)|x⟩

and the oracle's matrix will be a diagonal matrix of the form:

U

ω

=

⎡

⎢

⎢

⎢

⎢

⎢

⎢

⎣

(

−

1

)

f

(

0

)

0

⋯

0

0

(

−

1

)

f

(

1

)

⋯

0

⋮

0

⋱

⋮

0

0

⋯

(

−

1

)

f

(

2

n

−

1

)

⎤

⎥

⎥

⎥

⎥

⎥

⎥

⎦

Uω=[(−1)f(0)0⋯00(−1)f(1)⋯0⋮0⋱⋮00⋯(−1)f(2n−1)]

Circuit Construction of a Grover Oracle (click to expand)

For the next part of this chapter, we aim to teach the core concepts of the algorithm. We will create example oracles where we know

ω

ω

beforehand, and not worry ourselves with whether these oracles are useful or not. At the end of the chapter, we will cover a short example where we create an oracle to solve a problem (sudoku).

### **Amplitude Amplification**

So how does the algorithm work? Before looking at the list of items, we have no idea where the marked item is. Therefore, any guess of its location is as good as any other, which can be expressed in terms of a uniform superposition:

|

s

⟩

=

1

√

N

∑

N

−

1

x

=

0

|

x

⟩

.

|s⟩=1N∑x=0N−1|x⟩.

If at this point we were to measure in the standard basis

{

|

x

⟩

}

{|x⟩}

, this superposition would collapse, according to the fifth quantum law, to any one of the basis states with the same probability of

1

N

=

1

2

n

1N=12n

. Our chances of guessing the right value

w

w

is therefore

1

1

in

2

n

2n

, as could be expected. Hence, on average we would need to try about

N

/

2

=

2

n

−

1

N/2=2n−1

times to guess the correct item.

Enter the procedure called amplitude amplification, which is how a quantum computer significantly enhances this probability. This procedure stretches out (amplifies) the amplitude of the marked item, which shrinks the other items' amplitude, so that measuring the final state will return the right item with near-certainty.

This algorithm has a nice geometrical interpretation in terms of two reflections, which generate a rotation in a two-dimensional plane. The only two special states we need to consider are the winner

|

w

⟩

|w⟩

and the uniform superposition

|

s

⟩

|s⟩

. These two vectors span a two-dimensional plane in the vector space

C

N

.

CN.

They are not quite perpendicular because

|

w

⟩

|w⟩

occurs in the superposition with amplitude

N

−

1

/

2

N−1/2

as well. We can, however, introduce an additional state

|

s

′

⟩

|s′⟩

that is in the span of these two vectors, which is perpendicular to

|

w

⟩

|w⟩

and is obtained from

|

s

⟩

|s⟩

by removing

|

w

⟩

|w⟩

and rescaling.

Step 1: The amplitude amplification procedure starts out in the uniform superposition

|

s

⟩

|s⟩

, which is easily constructed from

|

s

⟩

=

H

⊗

n

|

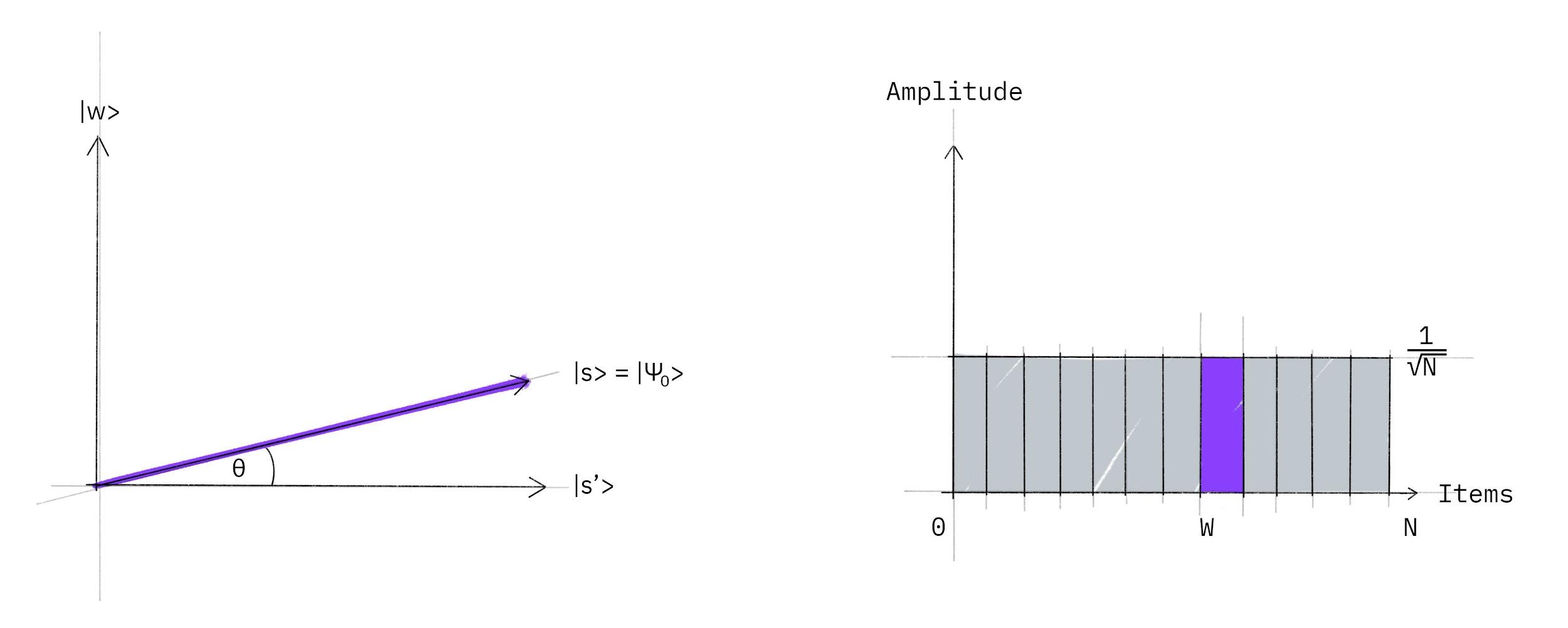
0

⟩

n

|s⟩=H⊗n|0⟩n

.



The left graphic corresponds to the two-dimensional plane spanned by perpendicular vectors

|

w

⟩

|w⟩

and

|

s

′

⟩

|s′⟩

which allows to express the initial state as

|

s

⟩

=

sin

θ

|

w

⟩

+

cos

θ

|

s

′

⟩

,

|s⟩=sin⁡θ|w⟩+cos⁡θ|s′⟩,

where

θ

=

arcsin

⟨

s

|

w

⟩

=

arcsin

1

√

N

θ=arcsin⁡⟨s|w⟩=arcsin⁡1N

. The right graphic is a bar graph of the amplitudes of the state

|

s

⟩

|s⟩

.

Step 2: We apply the oracle reflection

U

f

Uf

to the state

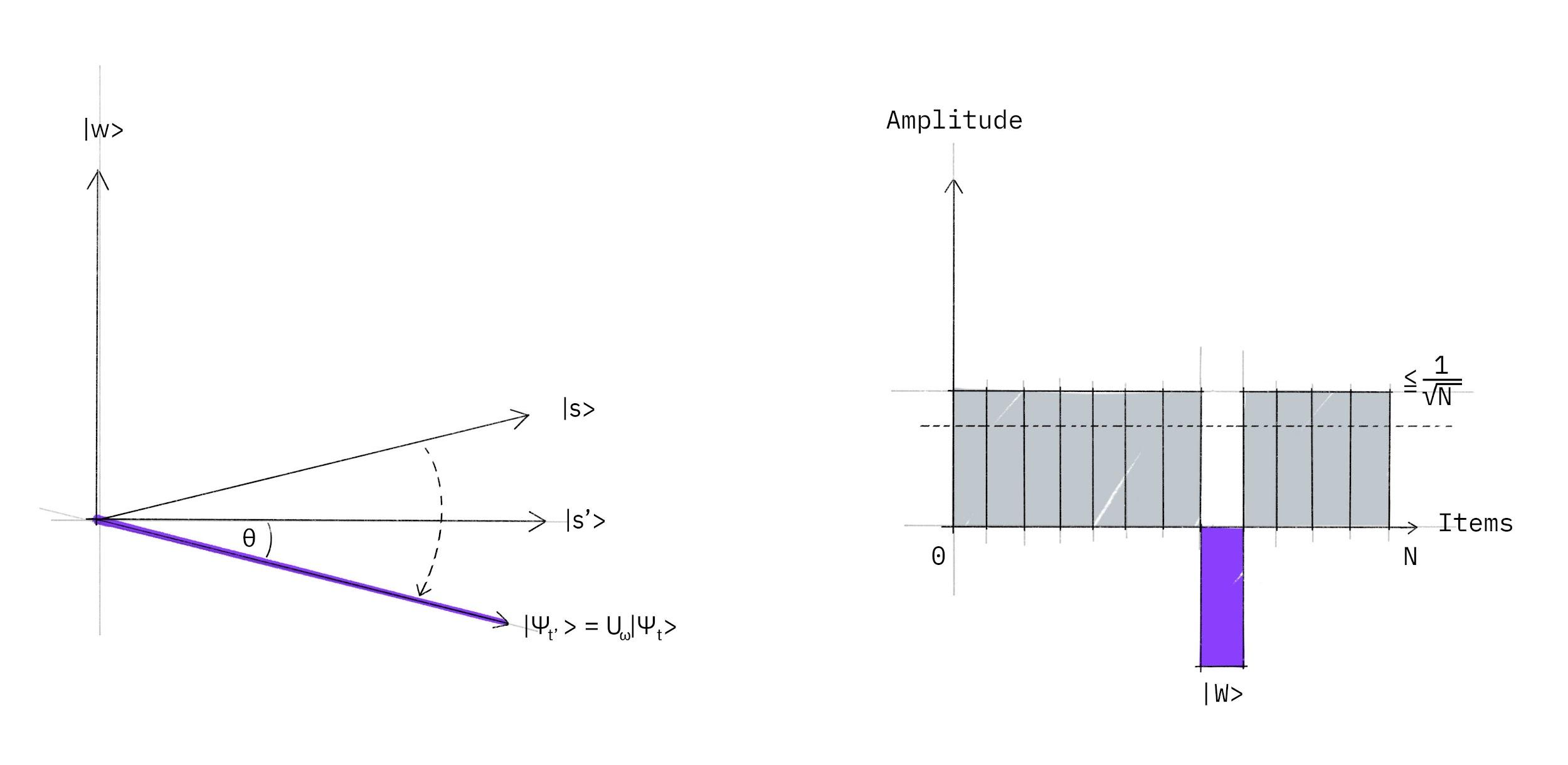
|

s

⟩

|s⟩

.



Geometrically this corresponds to a reflection of the state

|

s

⟩

|s⟩

about

|

s

′

⟩

|s′⟩

. This transformation means that the amplitude in front of the

|

w

⟩

|w⟩

state becomes negative, which in turn means that the average amplitude (indicated by a dashed line) has been lowered.

Step 3: We now apply an additional reflection (

U

s

Us

) about the state

|

s

⟩

|s⟩

:

U

s

=

2

|

s

⟩

⟨

s

|

−

1

Us=2|s⟩⟨s|−1

. This transformation maps the state to

U

s

U

f

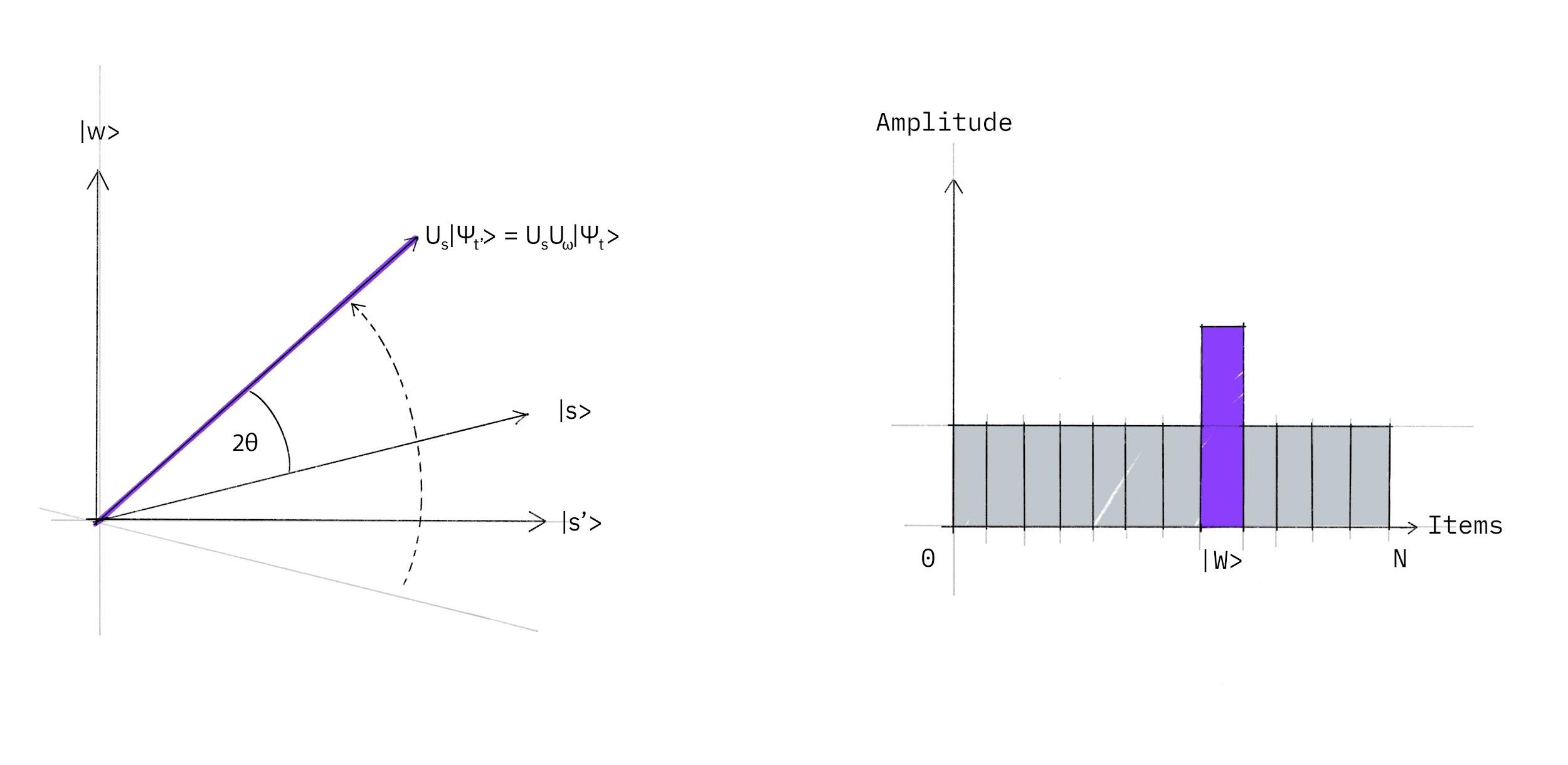
|

s

⟩

UsUf|s⟩

and completes the transformation.



Two reflections always correspond to a rotation. The transformation

U

s

U

f

UsUf

rotates the initial state

|

s

⟩

|s⟩

closer towards the winner

|

w

⟩

|w⟩

. The action of the reflection

U

s

Us

in the amplitude bar diagram can be understood as a reflection about the average amplitude. Since the average amplitude has been lowered by the first reflection, this transformation boosts the negative amplitude of

|

w

⟩

|w⟩

to roughly three times its original value, while it decreases the other amplitudes. We then go to step 2 to repeat the application. This procedure will be repeated several times to zero in on the winner.

After

t

t

steps we will be in the state

|

ψ

t

⟩

|ψt⟩

where:

|

ψ

t

⟩

=

(

U

s

U

f

)

t

|

s

⟩

.

|ψt⟩=(UsUf)t|s⟩.

How many times do we need to apply the rotation? It turns out that roughly

√

N

N

rotations suffice. This becomes clear when looking at the amplitudes of the state

|

ψ

⟩

|ψ⟩

. We can see that the amplitude of

|

w

⟩

|w⟩

grows linearly with the number of applications

∼

t

N

−

1

/

2

∼tN−1/2

. However, since we are dealing with amplitudes and not probabilities, the vector space's dimension enters as a square root. Therefore it is the amplitude, and not just the probability, that is being amplified in this procedure.

In the case that there are multiple solutions,

M

M

, it can be shown that roughly

√

(

N

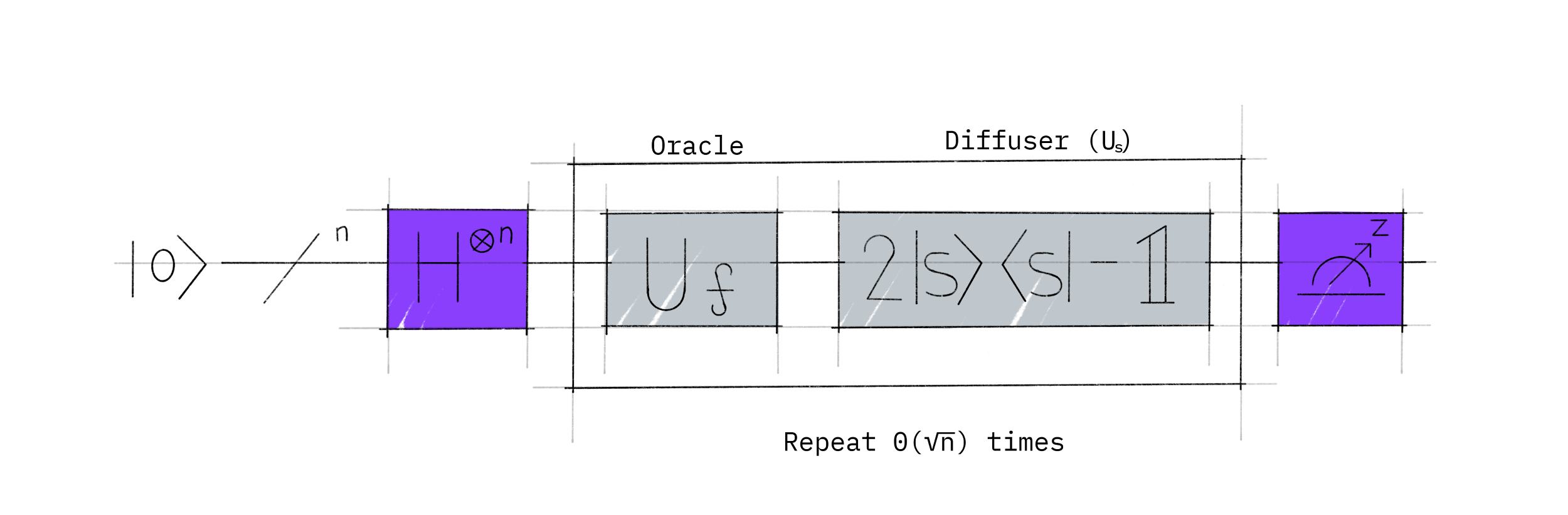
/

M

)

(N/M)

rotations will suffice.



## **2. Example: 2 Qubits**

Let's first have a look at the case of Grover's algorithm for

N

=

4

N=4

which is realized with 2 qubits. In this particular case, only one rotation is required to rotate the initial state

|

s

⟩

|s⟩

to the winner

|

w

⟩

|w⟩

[3]:

1. Following the above introduction, in the case
2. N
3. =
4. 4
5. N=4
6. we have
7. θ
8. =
9. arcsin
10. 1
11. 2
12. =
13. π
14. 6
15. .
16. θ=arcsin⁡12=π6.
17. After
18. t
19. t
20. steps, we have
21. (
22. U
23. s
24. U
25. ω
26. )
27. t
28. |
29. s
30. ⟩
31. =
32. sin
33. θ
34. t
35. |
36. ω
37. ⟩
38. +
39. cos
40. θ
41. t
42. |
43. s
44. ′
45. ⟩
46. ,
47. (UsUω)t|s⟩=sin⁡θt|ω⟩+cos⁡θt|s′⟩,
48. where
49. θ
50. t
51. =
52. (
53. 2
54. t
55. +
56. 1
57. )
58. θ
59. .
60. θt=(2t+1)θ.
61. In order to obtain
62. |
63. ω
64. ⟩
65. |ω⟩
66. we need
67. θ
68. t
69. =
70. π
71. 2
72. θt=π2
73. , which with
74. θ
75. =
76. π
77. 6
78. θ=π6
79. inserted above results to
80. t
81. =
82. 1
83. t=1
84. . This implies that after
85. t
86. =
87. 1
88. t=1
89. rotation the searched element is found.

We will now follow through an example using a specific oracle.

#### **Oracle for**

#### |

#### ω

#### ⟩

#### =

#### |

#### 11

#### ⟩

#### |ω⟩=|11⟩

#### 

Let's look at the case

|

w

⟩

=

|

11

⟩

|w⟩=|11⟩

. The oracle

U

ω

Uω

in this case acts as follows:

U

ω

|

s

⟩

=

U

ω

1

2

(

|

00

⟩

+

|

01

⟩

+

|

10

⟩

+

|

11

⟩

)

=

1

2

(

|

00

⟩

+

|

01

⟩

+

|

10

⟩

−

|

11

⟩

)

.

Uω|s⟩=Uω12(|00⟩+|01⟩+|10⟩+|11⟩)=12(|00⟩+|01⟩+|10⟩−|11⟩).

or:

U

ω

=

⎡

⎢

⎢

⎢

⎣

1

0

0

0

0

1

0

0

0

0

1

0

0

0

0

−

1

⎤

⎥

⎥

⎥

⎦

Uω=[100001000010000−1]

which you may recognise as the controlled-Z gate. Thus, for this example, our oracle is simply the controlled-Z gate:

image6

#### **Reflection**

#### U

#### s

#### Us

#### 

In order to complete the circuit we need to implement the additional reflection

U

s

=

2

|

s

⟩

⟨

s

|

−

1

Us=2|s⟩⟨s|−1

. Since this is a reflection about

|

s

⟩

|s⟩

, we want to add a negative phase to every state orthogonal to

|

s

⟩

|s⟩

.

One way we can do this is to use the operation that transforms the state

|

s

⟩

→

|

0

⟩

|s⟩→|0⟩

, which we already know is the Hadamard gate applied to each qubit:

H

⊗

n

|

s

⟩

=

|

0

⟩

H⊗n|s⟩=|0⟩

Then we apply a circuit that adds a negative phase to the states orthogonal to

|

0

⟩

|0⟩

:

U

0

1

2

(

|

00

⟩

+

|

01

⟩

+

|

10

⟩

+

|

11

⟩

)

=

1

2

(

|

00

⟩

−

|

01

⟩

−

|

10

⟩

−

|

11

⟩

)

U012(|00⟩+|01⟩+|10⟩+|11⟩)=12(|00⟩−|01⟩−|10⟩−|11⟩)

i.e. the signs of each state are flipped except for

|

00

⟩

|00⟩

. As can easily be verified, one way of implementing

U

0

U0

is the following circuit:

Circuit for reflection around |0>

Finally, we do the operation that transforms the state

|

0

⟩

→

|

s

⟩

|0⟩→|s⟩

(the H-gate again):

H

⊗

n

U

0

H

⊗

n

=

U

s

H⊗nU0H⊗n=Us

The complete circuit for

U

s

Us

looks like this:

Circuit for reflection around |s>

#### **Full Circuit for**

#### |

#### w

#### ⟩

#### =

#### |

#### 11

#### ⟩

#### |w⟩=|11⟩

#### 

Since in the particular case of

N

=

4

N=4

only one rotation is required we can combine the above components to build the full circuit for Grover's algorithm for the case

|

w

⟩

=

|

11

⟩

|w⟩=|11⟩

:

image10

### **2.1 Qiskit Implementation**

We now implement Grover's algorithm for the above case of 2 qubits for

|

w

⟩

=

|

11

⟩

|w⟩=|11⟩

.

*#initialization*

import matplotlib.pyplot **as** plt

import numpy **as** np

*# importing Qiskit*

from qiskit import IBMQ, Aer, assemble, transpile

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

from qiskit.providers.ibmq import least\_busy

*# import basic plot tools*

from qiskit.visualization import plot\_histogram



try

We start by preparing a quantum circuit with two qubits:

n **=** 2

grover\_circuit **=** QuantumCircuit(n)



try

Then we simply need to write out the commands for the circuit depicted above. First, we need to initialize the state

|

s

⟩

|s⟩

. Let's create a general function (for any number of qubits) so we can use it again later:

**def** **initialize\_s**(qc, qubits):

"""Apply a H-gate to 'qubits' in qc"""

**for** q **in** qubits:

qc**.**h(q)

**return** qc



try

grover\_circuit **=** initialize\_s(grover\_circuit, [0,1])

grover\_circuit**.**draw()



try

Apply the Oracle for

|

w

⟩

=

|

11

⟩

|w⟩=|11⟩

. This oracle is specific to 2 qubits:

grover\_circuit**.**cz(0,1) *# Oracle*

grover\_circuit**.**draw()



try

We now want to apply the diffuser (

U

s

Us

). As with the circuit that initializes

|

s

⟩

|s⟩

, we'll create a general diffuser (for any number of qubits) so we can use it later in other problems.

*# Diffusion operator (U\_s)*

grover\_circuit**.**h([0,1])

grover\_circuit**.**z([0,1])

grover\_circuit**.**cz(0,1)

grover\_circuit**.**h([0,1])

grover\_circuit**.**draw()



try

This is our finished circuit.

### **2.1.1 Experiment with Simulators**

Let's run the circuit in simulation. First, we can verify that we have the correct statevector:

sim **=** Aer**.**get\_backend('aer\_simulator')

*# we need to make a copy of the circuit with the 'save\_statevector'*

*# instruction to run on the Aer simulator*

grover\_circuit\_sim **=** grover\_circuit**.**copy()

grover\_circuit\_sim**.**save\_statevector()

qobj **=** assemble(grover\_circuit\_sim)

result **=** sim**.**run(qobj)**.**result()

statevec **=** result**.**get\_statevector()

from qiskit\_textbook.tools import vector2latex

vector2latex(statevec, pretext**=**"|\\psi\\rangle =")



try

|

ψ

⟩

=

⎡

⎢

⎢

⎢

⎣

0

0

0

1

⎤

⎥

⎥

⎥

⎦

|ψ⟩=[0001]

As expected, the amplitude of every state that is not

|

11

⟩

|11⟩

is 0, this means we have a 100% chance of measuring

|

11

⟩

|11⟩

:

grover\_circuit**.**measure\_all()

aer\_sim **=** Aer**.**get\_backend('aer\_simulator')

qobj **=** assemble(grover\_circuit)

result **=** aer\_sim**.**run(qobj)**.**result()

counts **=** result**.**get\_counts()

plot\_histogram(counts)



try

### **2.1.2 Experiment with Real Devices**

We can run the circuit a real device as below.

*# Load IBM Q account and get the least busy backend device*

provider **=** IBMQ**.**load\_account()

provider **=** IBMQ**.**get\_provider("ibm-q")

device **=** least\_busy(provider**.**backends(filters**=lambda** x: x**.**configuration()**.**n\_qubits **>=** 3 **and**

**not** x**.**configuration()**.**simulator **and** x**.**status()**.**operational**==True**))

print("Running on current least busy device: ", device)



try

Running on current least busy device: ibmq\_mumbai

*# Run our circuit on the least busy backend. Monitor the execution of the job in the queue*

from qiskit.tools.monitor import job\_monitor

transpiled\_grover\_circuit **=** transpile(grover\_circuit, device, optimization\_level**=**3)

job **=** device**.**run(transpiled\_grover\_circuit)

job\_monitor(job, interval**=**2)



try

Job Status: job has successfully run

*# Get the results from the computation*

results **=** job**.**result()

answer **=** results**.**get\_counts(grover\_circuit)

plot\_histogram(answer)



try

We confirm that in the majority of the cases the state

|

11

⟩

|11⟩

is measured. The other results are due to errors in the quantum computation.

## **3. Example: 3 Qubits**

We now go through the example of Grover's algorithm for 3 qubits with two marked states

|

101

⟩

|101⟩

and

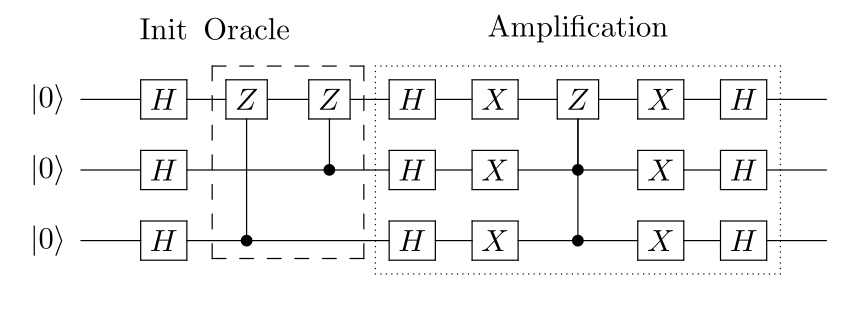
|

110

⟩

|110⟩

, following the implementation found in Reference [2]. The quantum circuit to solve the problem using a phase oracle is:



1. Apply Hadamard gates to
2. 3
3. 3
4. qubits initialized to
5. |
6. 000
7. ⟩
8. |000⟩
9. to create a uniform superposition:
10. |
11. ψ
12. 1
13. ⟩
14. =
15. 1
16. √
17. 8
18. (
19. |
20. 000
21. ⟩
22. +
23. |
24. 001
25. ⟩
26. +
27. |
28. 010
29. ⟩
30. +
31. |
32. 011
33. ⟩
34. +
35. |
36. 100
37. ⟩
38. +
39. |
40. 101
41. ⟩
42. +
43. |
44. 110
45. ⟩
46. +
47. |
48. 111
49. ⟩
50. )
51. |ψ1⟩=18(|000⟩+|001⟩+|010⟩+|011⟩+|100⟩+|101⟩+|110⟩+|111⟩)
52. Mark states
53. |
54. 101
55. ⟩
56. |101⟩
57. and
58. |
59. 110
60. ⟩
61. |110⟩
62. using a phase oracle:
63. |
64. ψ
65. 2
66. ⟩
67. =
68. 1
69. √
70. 8
71. (
72. |
73. 000
74. ⟩
75. +
76. |
77. 001
78. ⟩
79. +
80. |
81. 010
82. ⟩
83. +
84. |
85. 011
86. ⟩
87. +
88. |
89. 100
90. ⟩
91. −
92. |
93. 101
94. ⟩
95. −
96. |
97. 110
98. ⟩
99. +
100. |
101. 111
102. ⟩
103. )
104. |ψ2⟩=18(|000⟩+|001⟩+|010⟩+|011⟩+|100⟩−|101⟩−|110⟩+|111⟩)
105. Perform the reflection around the average amplitude:
     1. Apply Hadamard gates to the qubits
     2. |
     3. ψ
     4. 3
     5. a
     6. ⟩
     7. =
     8. 1
     9. 2
     10. (
     11. |
     12. 000
     13. ⟩
     14. +
     15. |
     16. 011
     17. ⟩
     18. +
     19. |
     20. 100
     21. ⟩
     22. −
     23. |
     24. 111
     25. ⟩
     26. )
     27. |ψ3a⟩=12(|000⟩+|011⟩+|100⟩−|111⟩)
     28. Apply X gates to the qubits
     29. |
     30. ψ
     31. 3
     32. b
     33. ⟩
     34. =
     35. 1
     36. 2
     37. (
     38. −
     39. |
     40. 000
     41. ⟩
     42. +
     43. |
     44. 011
     45. ⟩
     46. +
     47. |
     48. 100
     49. ⟩
     50. +
     51. |
     52. 111
     53. ⟩
     54. )
     55. |ψ3b⟩=12(−|000⟩+|011⟩+|100⟩+|111⟩)
     56. Apply a doubly controlled Z gate between the 1, 2 (controls) and 3 (target) qubits
     57. |
     58. ψ
     59. 3
     60. c
     61. ⟩
     62. =
     63. 1
     64. 2
     65. (
     66. −
     67. |
     68. 000
     69. ⟩
     70. +
     71. |
     72. 011
     73. ⟩
     74. +
     75. |
     76. 100
     77. ⟩
     78. −
     79. |
     80. 111
     81. ⟩
     82. )
     83. |ψ3c⟩=12(−|000⟩+|011⟩+|100⟩−|111⟩)
     84. Apply X gates to the qubits
     85. |
     86. ψ
     87. 3
     88. d
     89. ⟩
     90. =
     91. 1
     92. 2
     93. (
     94. −
     95. |
     96. 000
     97. ⟩
     98. +
     99. |
     100. 011
     101. ⟩
     102. +
     103. |
     104. 100
     105. ⟩
     106. −
     107. |
     108. 111
     109. ⟩
     110. )
     111. |ψ3d⟩=12(−|000⟩+|011⟩+|100⟩−|111⟩)
     112. Apply Hadamard gates to the qubits
     113. |
     114. ψ
     115. 3
     116. e
     117. ⟩
     118. =
     119. 1
     120. √
     121. 2
     122. (
     123. −
     124. |
     125. 101
     126. ⟩
     127. −
     128. |
     129. 110
     130. ⟩
     131. )
     132. |ψ3e⟩=12(−|101⟩−|110⟩)
106. Measure the
107. 3
108. 3
109. qubits to retrieve states
110. |
111. 101
112. ⟩
113. |101⟩
114. and
115. |
116. 110
117. ⟩
118. |110⟩

Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration (steps 2 & 3).

### **3.1 Qiskit Implementation**

We now implement Grover's algorithm for the above [example](https://qiskit.org/textbook/ch-algorithms/grover.html#3qubits) for

3

3

-qubits and searching for two marked states

|

101

⟩

|101⟩

and

|

110

⟩

|110⟩

. Note: Remember that Qiskit orders it's qubits the opposite way round to this resource, so the circuit drawn will appear flipped about the horizontal.

We create a phase oracle that will mark states

|

101

⟩

|101⟩

and

|

110

⟩

|110⟩

as the results (step 1).

qc **=** QuantumCircuit(3)

qc**.**cz(0, 2)

qc**.**cz(1, 2)

oracle\_ex3 **=** qc**.**to\_gate()

oracle\_ex3**.**name **=** "U$\_\omega$"



try

In the last section, we used a diffuser specific to 2 qubits, in the cell below we will create a general diffuser for any number of qubits.

Details: Creating a General Diffuser (click to expand)

**def** **diffuser**(nqubits):

qc **=** QuantumCircuit(nqubits)

*# Apply transformation |s> -> |00..0> (H-gates)*

**for** qubit **in** range(nqubits):

qc**.**h(qubit)

*# Apply transformation |00..0> -> |11..1> (X-gates)*

**for** qubit **in** range(nqubits):

qc**.**x(qubit)

*# Do multi-controlled-Z gate*

qc**.**h(nqubits**-**1)

qc**.**mct(list(range(nqubits**-**1)), nqubits**-**1) *# multi-controlled-toffoli*

qc**.**h(nqubits**-**1)

*# Apply transformation |11..1> -> |00..0>*

**for** qubit **in** range(nqubits):

qc**.**x(qubit)

*# Apply transformation |00..0> -> |s>*

**for** qubit **in** range(nqubits):

qc**.**h(qubit)

*# We will return the diffuser as a gate*

U\_s **=** qc**.**to\_gate()

U\_s**.**name **=** "U$\_s$"

**return** U\_s



try

We'll now put the pieces together, with the creation of a uniform superposition at the start of the circuit and a measurement at the end. Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration.

n **=** 3

grover\_circuit **=** QuantumCircuit(n)

grover\_circuit **=** initialize\_s(grover\_circuit, [0,1,2])

grover\_circuit**.**append(oracle\_ex3, [0,1,2])

grover\_circuit**.**append(diffuser(n), [0,1,2])

grover\_circuit**.**measure\_all()

grover\_circuit**.**draw()



try

### **3.1.1 Experiment with Simulators**

We can run the above circuit on the simulator.

aer\_sim **=** Aer**.**get\_backend('aer\_simulator')

transpiled\_grover\_circuit **=** transpile(grover\_circuit, aer\_sim)

qobj **=** assemble(transpiled\_grover\_circuit)

results **=** aer\_sim**.**run(qobj)**.**result()

counts **=** results**.**get\_counts()

plot\_histogram(counts)



try

As we can see, the algorithm discovers our marked states

|

101

⟩

|101⟩

and

|

110

⟩

|110⟩

.

### **3.1.2 Experiment with Real Devices**

We can run the circuit on the real device as below.

backend **=** least\_busy(provider**.**backends(filters**=lambda** x: x**.**configuration()**.**n\_qubits **>=** 3 **and**

**not** x**.**configuration()**.**simulator **and** x**.**status()**.**operational**==True**))

print("least busy backend: ", backend)



try

least busy backend: ibmq\_mumbai

*# Run our circuit on the least busy backend. Monitor the execution of the job in the queue*

from qiskit.tools.monitor import job\_monitor

transpiled\_grover\_circuit **=** transpile(grover\_circuit, device, optimization\_level**=**3)

job **=** device**.**run(transpiled\_grover\_circuit)

job\_monitor(job, interval**=**2)



try

Job Status: job has successfully run

*# Get the results from the computation*

results **=** job**.**result()

answer **=** results**.**get\_counts(grover\_circuit)

plot\_histogram(answer)



try

As we can (hopefully) see, there is a higher chance of measuring

|

101

⟩

|101⟩

and

|

110

⟩

|110⟩

. The other results are due to errors in the quantum computation.

## **4. Problems**

The function grover\_problem\_oracle below takes a number of qubits (n), and a variant and returns an n-qubit oracle. The function will always return the same oracle for the same n and variant. You can see the solutions to each oracle by setting print\_solutions = True when calling grover\_problem\_oracle.

from qiskit\_textbook.problems import grover\_problem\_oracle

*## Example Usage*

n **=** 4

oracle **=** grover\_problem\_oracle(n, variant**=**1) *# 0th variant of oracle, with n qubits*

qc **=** QuantumCircuit(n)

qc**.**append(oracle, [0,1,2,3])

qc**.**draw()



try

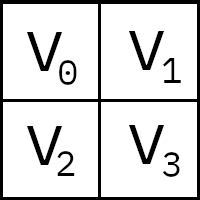
1. grover\_problem\_oracle(4, variant=2) uses 4 qubits and has 1 solution.  
   a. How many iterations do we need to have a > 90% chance of measuring this solution?  
   b. Use Grover's algorithm to find this solution state. c. What happens if we apply more iterations the number we calculated in problem 1a above? Why?
2. With 2 solutions and 4 qubits, how many iterations do we need for a >90% chance of measuring a solution? Test your answer using the oracle grover\_problem\_oracle(4, variant=1) (which has two solutions).
3. Create a function, grover\_solver(oracle, iterations) that takes as input:
   * A Grover oracle as a gate (oracle)
   * An integer number of iterations (iterations)
4. and returns a QuantumCircuit that performs Grover's algorithm on the 'oracle' gate, with 'iterations' iterations.

## **5. Solving Sudoku using Grover's Algorithm**

The oracles used throughout this chapter so far have been created with prior knowledge of their solutions. We will now solve a simple problem using Grover's algorithm, for which we do not necessarily know the solution beforehand. Our problem is a 2×2 binary sudoku, which in our case has two simple rules:

* No column may contain the same value twice
* No row may contain the same value twice

If we assign each square in our sudoku to a variable like so:



we want our circuit to output a solution to this sudoku.

Note that, while this approach of using Grover's algorithm to solve this problem is not practical (you can probably find the solution in your head!), the purpose of this example is to demonstrate the conversion of classical [decision problems](https://en.wikipedia.org/wiki/Decision_problem) into oracles for Grover's algorithm.

### **5.1 Turning the Problem into a Circuit**

We want to create an oracle that will help us solve this problem, and we will start by creating a circuit that identifies a correct solution. Similar to how we created a classical adder using quantum circuits in [*The Atoms of Computation*](https://qiskit.org/textbook/ch-states/atoms-computation.html), we simply need to create a *classical* function on a quantum circuit that checks whether the state of our variable bits is a valid solution.

Since we need to check down both columns and across both rows, there are 4 conditions we need to check:

v0 ≠ v1 # check along top row

v2 ≠ v3 # check along bottom row

v0 ≠ v2 # check down left column

v1 ≠ v3 # check down right column

Remember we are comparing classical (computational basis) states. For convenience, we can compile this set of comparisons into a list of clauses:

clause\_list **=** [[0,1],

[0,2],

[1,3],

[2,3]]



try

We will assign the value of each variable to a bit in our circuit. To check these clauses computationally, we will use the XOR gate (we came across this in the atoms of computation).

**def** **XOR**(qc, a, b, output):

qc**.**cx(a, output)

qc**.**cx(b, output)



try

Convince yourself that the output0 bit in the circuit below will only be flipped if input0 ≠ input1:

*# We will use separate registers to name the bits*

in\_qubits **=** QuantumRegister(2, name**=**'input')

out\_qubit **=** QuantumRegister(1, name**=**'output')

qc **=** QuantumCircuit(in\_qubits, out\_qubit)

XOR(qc, in\_qubits[0], in\_qubits[1], out\_qubit)

qc**.**draw()



try

This circuit checks whether input0 == input1 and stores the output to output0. To check each clause, we repeat this circuit for each pairing in clause\_list and store the output to a new bit:

*# Create separate registers to name bits*

var\_qubits **=** QuantumRegister(4, name**=**'v') *# variable bits*

clause\_qubits **=** QuantumRegister(4, name**=**'c') *# bits to store clause-checks*

*# Create quantum circuit*

qc **=** QuantumCircuit(var\_qubits, clause\_qubits)

*# Use XOR gate to check each clause*

i **=** 0

**for** clause **in** clause\_list:

XOR(qc, clause[0], clause[1], clause\_qubits[i])

i **+=** 1

qc**.**draw()



try

The final state of the bits c0, c1, c2, c3 will only all be 1 in the case that the assignments of v0, v1, v2, v3 are a solution to the sudoku. To complete our checking circuit, we want a single bit to be 1 if (and only if) all the clauses are satisfied, this way we can look at just one bit to see if our assignment is a solution. We can do this using a multi-controlled-Toffoli-gate:

*# Create separate registers to name bits*

var\_qubits **=** QuantumRegister(4, name**=**'v')

clause\_qubits **=** QuantumRegister(4, name**=**'c')

output\_qubit **=** QuantumRegister(1, name**=**'out')

qc **=** QuantumCircuit(var\_qubits, clause\_qubits, output\_qubit)

*# Compute clauses*

i **=** 0

**for** clause **in** clause\_list:

XOR(qc, clause[0], clause[1], clause\_qubits[i])

i **+=** 1

*# Flip 'output' bit if all clauses are satisfied*

qc**.**mct(clause\_qubits, output\_qubit)

qc**.**draw()



try

The circuit above takes as input an initial assignment of the bits v0, v1, v2 and v3, and all other bits should be initialized to 0. After running the circuit, the state of the out0 bit tells us if this assignment is a solution or not; out0 = 0 means the assignment *is not* a solution, and out0 = 1 means the assignment *is* a solution.

Important: Before you continue, it is important you fully understand this circuit and are convinced it works as stated in the paragraph above.

### **5.2 Uncomputing, and Completing the Oracle**

We can now turn this checking circuit into a Grover oracle using [phase kickback](https://qiskit.org/textbook/ch-gates/phase-kickback.html). To recap, we have 3 registers:

* One register which stores our sudoku variables (we'll say
* x
* =
* v
* 3
* ,
* v
* 2
* ,
* v
* 1
* ,
* v
* 0
* x=v3,v2,v1,v0
* )
* One register that stores our clauses (this starts in the state
* |
* 0000
* ⟩
* |0000⟩
* which we'll abbreviate to
* |
* 0
* ⟩
* |0⟩
* )
* And one qubit (
* |
* out
* 0
* ⟩
* |out0⟩
* ) that we've been using to store the output of our checking circuit.

To create an oracle, we need our circuit (

U

ω

Uω

) to perform the transformation:

U

ω

|

x

⟩

|

0

⟩

|

out

0

⟩

=

|

x

⟩

|

0

⟩

|

out

0

⊕

f

(

x

)

⟩

Uω|x⟩|0⟩|out0⟩=|x⟩|0⟩|out0⊕f(x)⟩

If we set the out0 qubit to the superposition state

|

−

⟩

|−⟩

we have:

U

ω

|

x

⟩

|

0

⟩

|

−

⟩

=

U

ω

|

x

⟩

|

0

⟩

⊗

1

√

2

(

|

0

⟩

−

|

1

⟩

)

=

|

x

⟩

|

0

⟩

⊗

1

√

2

(

|

0

⊕

f

(

x

)

⟩

−

|

1

⊕

f

(

x

)

⟩

)

Uω|x⟩|0⟩|−⟩=Uω|x⟩|0⟩⊗12(|0⟩−|1⟩)=|x⟩|0⟩⊗12(|0⊕f(x)⟩−|1⊕f(x)⟩)

If

f

(

x

)

=

0

f(x)=0

, then we have the state:

=

|

x

⟩

|

0

⟩

⊗

1

√

2

(

|

0

⟩

−

|

1

⟩

)

=

|

x

⟩

|

0

⟩

|

−

⟩

=|x⟩|0⟩⊗12(|0⟩−|1⟩)=|x⟩|0⟩|−⟩

(i.e. no change). But if

f

(

x

)

=

1

f(x)=1

(i.e.

x

=

ω

x=ω

), we introduce a negative phase to the

|

−

⟩

|−⟩

qubit:

=

|

x

⟩

|

0

⟩

⊗

1

√

2

(

|

1

⟩

−

|

0

⟩

)

=

|

x

⟩

|

0

⟩

⊗

−

1

√

2

(

|

0

⟩

−

|

1

⟩

)

=

−

|

x

⟩

|

0

⟩

|

−

⟩

=−|x⟩|0⟩⊗12(|1⟩−|0⟩)=−|x⟩|0⟩⊗−12(|0⟩−|1⟩)=−|x⟩|0⟩|−⟩

This is a functioning oracle that uses two auxiliary registers in the state

|

0

⟩

|

−

⟩

|0⟩|−⟩

:

U

ω

|

x

⟩

|

0

⟩

|

−

⟩

=

{

|

x

⟩

|

0

⟩

|

−

⟩

for

x

≠

ω

−

|

x

⟩

|

0

⟩

|

−

⟩

for

x

=

ω

Uω|x⟩|0⟩|−⟩={−|x⟩|0⟩|−⟩forx≠ω−|x⟩|0⟩|−⟩forx=ω

To adapt our checking circuit into a Grover oracle, we need to guarantee the bits in the second register (c) are always returned to the state

|

0000

⟩

|0000⟩

after the computation. To do this, we simply repeat the part of the circuit that computes the clauses which guarantees c0 = c1 = c2 = c3 = 0 after our circuit has run. We call this step *'uncomputation'*.

var\_qubits **=** QuantumRegister(4, name**=**'v')

clause\_qubits **=** QuantumRegister(4, name**=**'c')

output\_qubit **=** QuantumRegister(1, name**=**'out')

cbits **=** ClassicalRegister(4, name**=**'cbits')

qc **=** QuantumCircuit(var\_qubits, clause\_qubits, output\_qubit, cbits)

**def** **sudoku\_oracle**(qc, clause\_list, clause\_qubits):

*# Compute clauses*

i **=** 0

**for** clause **in** clause\_list:

XOR(qc, clause[0], clause[1], clause\_qubits[i])

i **+=** 1

*# Flip 'output' bit if all clauses are satisfied*

qc**.**mct(clause\_qubits, output\_qubit)

*# Uncompute clauses to reset clause-checking bits to 0*

i **=** 0

**for** clause **in** clause\_list:

XOR(qc, clause[0], clause[1], clause\_qubits[i])

i **+=** 1

sudoku\_oracle(qc, clause\_list, clause\_qubits)

qc**.**draw()



try

In summary, the circuit above performs:

U

ω

|

x

⟩

|

0

⟩

|

out

0

⟩

=

{

|

x

⟩

|

0

⟩

|

out

0

⟩

for

x

≠

ω

|

x

⟩

|

0

⟩

⊗

X

|

out

0

⟩

for

x

=

ω

Uω|x⟩|0⟩|out0⟩={|x⟩|0⟩|out0⟩forx≠ω|x⟩|0⟩⊗X|out0⟩forx=ω

and if the initial state of

|

out

0

⟩

=

|

−

⟩

|out0⟩=|−⟩

,:

U

ω

|

x

⟩

|

0

⟩

|

−

⟩

=

{

|

x

⟩

|

0

⟩

|

−

⟩

for

x

≠

ω

−

|

x

⟩

|

0

⟩

|

−

⟩

for

x

=

ω

Uω|x⟩|0⟩|−⟩={−|x⟩|0⟩|−⟩forx≠ω−|x⟩|0⟩|−⟩forx=ω

### **5.3 The Full Algorithm**

All that's left to do now is to put this oracle into Grover's algorithm!

var\_qubits **=** QuantumRegister(4, name**=**'v')

clause\_qubits **=** QuantumRegister(4, name**=**'c')

output\_qubit **=** QuantumRegister(1, name**=**'out')

cbits **=** ClassicalRegister(4, name**=**'cbits')

qc **=** QuantumCircuit(var\_qubits, clause\_qubits, output\_qubit, cbits)

*# Initialize 'out0' in state |->*

qc**.**initialize([1, **-**1]**/**np**.**sqrt(2), output\_qubit)

*# Initialize qubits in state |s>*

qc**.**h(var\_qubits)

qc**.**barrier() *# for visual separation*

*## First Iteration*

*# Apply our oracle*

sudoku\_oracle(qc, clause\_list, clause\_qubits)

qc**.**barrier() *# for visual separation*

*# Apply our diffuser*

qc**.**append(diffuser(4), [0,1,2,3])

*## Second Iteration*

sudoku\_oracle(qc, clause\_list, clause\_qubits)

qc**.**barrier() *# for visual separation*

*# Apply our diffuser*

qc**.**append(diffuser(4), [0,1,2,3])

*# Measure the variable qubits*

qc**.**measure(var\_qubits, cbits)

qc**.**draw(fold**=-**1)



try

*# Simulate and plot results*

aer\_simulator **=** Aer**.**get\_backend('aer\_simulator')

transpiled\_qc **=** transpile(qc, aer\_simulator)

qobj **=** assemble(transpiled\_qc)

result **=** aer\_sim**.**run(qobj)**.**result()

plot\_histogram(result**.**get\_counts())



try

There are two bit strings with a much higher probability of measurement than any of the others, 0110 and 1001. These correspond to the assignments:

v0 = 0

v1 = 1

v2 = 1

v3 = 0

and

v0 = 1

v1 = 0

v2 = 0

v3 = 1

which are the two solutions to our sudoku! The aim of this section is to show how we can create Grover oracles from real problems. While this specific problem is trivial, the process can be applied (allowing large enough circuits) to any decision problem. To recap, the steps are:

1. Create a reversible classical circuit that identifies a correct solution
2. Use phase kickback and uncomputation to turn this circuit into an oracle
3. Use Grover's algorithm to solve this oracle

## **6. References**

1. L. K. Grover (1996), "A fast quantum mechanical algorithm for database search", Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996), [doi:10.1145/237814.237866](http://doi.acm.org/10.1145/237814.237866), [arXiv:quant-ph/9605043](https://arxiv.org/abs/quant-ph/9605043)
2. C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath & C. Monroe (2017), "Complete 3-Qubit Grover search on a programmable quantum computer", Nature Communications, Vol 8, Art 1918, [doi:10.1038/s41467-017-01904-7](https://doi.org/10.1038/s41467-017-01904-7), [arXiv:1703.10535](https://arxiv.org/abs/1703.10535)
3. I. Chuang & M. Nielsen, "Quantum Computation and Quantum Information", Cambridge: Cambridge University Press, 2000.

import qiskit.tools.jupyter

**%qiskit\_version\_table**



try

### **Version Information**

| **Qiskit Software** | **Version** |
| --- | --- |
| Qiskit | 0.27.0 |
| Terra | 0.17.4 |
| Aer | 0.8.2 |
| Ignis | 0.6.0 |
| Aqua | 0.9.2 |
| IBM Q Provider | 0.14.0 |
| **System information** |  |
| Python | 3.7.7 (default, May 6 2020, 04:59:01) [Clang 4.0.1 (tags/RELEASE\_401/final)] |
| OS | Darwin |
| CPUs | 8 |
| Memory (Gb) | 32.0 |
| Wed Jun 16 13:45:01 2021 BST | |