

Grover's Algorithm - An attemptation

Yuanhao Ma

Before Everything

In this notebook, I aim to reproduce some key contents from the paper *Discrete Optimization: A Quantum Revolution?* by Prof. Stefan Creemers and Dr. Luis Fernando Pérez using a numpy-based simulation. This is my first exposure to the idea that optimization methods can be fundamentally accelerated through quantum computing, a paradigm shift that offers a completely different perspective. If quantum computing becomes practically viable in the future, not only will hardware undergo a revolution, but there will also be a significant demand for new software. Moreover, the experience accumulated in developing classical algorithms could play a crucial role in the quantum computing era. This presents a vast research gap that needs to be explored in but not only in the Operations Research Research community.

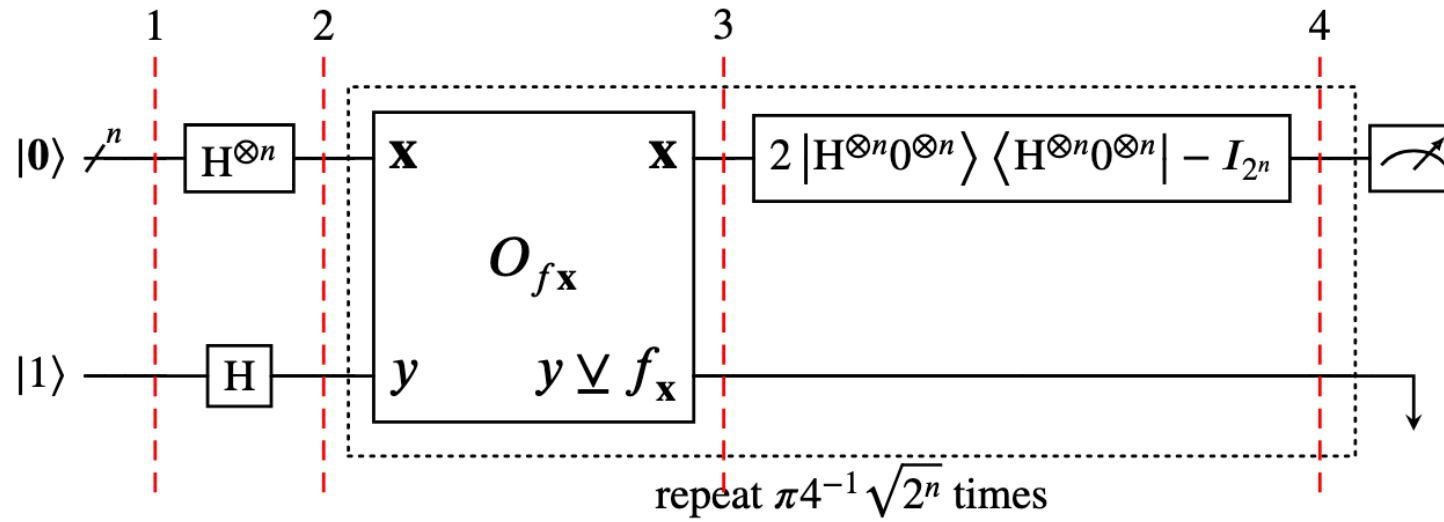
For now, we will primarily focus on reproducing the results from **Part 4: Grover's Algorithm**

Grover's algorithm is one of the most well-known quantum algorithms, making it an excellent starting point for understanding the fundamentals of quantum computing. By first exploring its mechanics, we can build a solid foundation.

Grover's Algorithm

Grover's Algorithm is designed to find a target entry in an unsorted database. The quantum circuit for a two-qubit system is shown below (From *Discrete Optimization: A Quantum Revolution?*). The fundamental steps of the algorithm are as follows:

- **Initialization** (region1-2 in the circuit): Apply Hadamard gates to all qubits to create an equal superposition state.
- **Oracle Operator** (region2-3 in the circuit): Pass qubits through Oracle Operator to mark the target state by flipping its phase
- **Diffusion Operator** (region3-4 in the circuit): Amplify the probability amplitude of the target state through an inversion-about-the-mean operation. This step is repeated for n iterations to maximize the probability of measuring the correct result.
- **Measurement** (region4- in the circuit): Perform a quantum measurement to obtain the final result with high probability.



```
In [18]: # import essential libraries
import numpy as np
import matplotlib.pyplot as plt
```

First, lets define how to use martix to represent n-qubit

For a single qubit system ($n = 1$), assume the basis state are $|0\rangle$ and $|1\rangle$, a general state representation is

$$|\phi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$$

and the column vector could be

$$|\phi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} \quad (1)$$

And for a 2-qubit system ($n = 2$) there could be 2^2 possible state which are $|\phi\rangle = \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$

And similarly a column vector representation could be

$$|\phi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \quad (2)$$

The initialization of this column vector is very flexible; we only need to ensure that the sum of the squares of each entry is equal to 1 to meet the amplitude definition. We simply initialize the first state with an amplitude of 1, and all other states with 0 in the remaining part

Then, lets dive into the main algorithm.

The first step is to define the Hadamard Gate function. This is crucial because it initializes the qubits into an equal superposition state, distributing their probability amplitudes evenly across all possible states.

For example, a 1-qubit system column vector should meet $\alpha_0^2 + \alpha_1^2 = 1$ and $\alpha_0^2 = \alpha_1^2 = 0.5$ after Hadmard gate;

a 2-qubit system column vector should meet $\alpha_0^2 + \alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1$ and $\alpha_0^2 = \alpha_1^2 = \alpha_2^2 = \alpha_3^2 = 0.25$ after Hadmard gate.

Since we want our function to be generalized for an n-qubit system, we will construct the n-dimensional Hadamard gate using the Kronecker product (`np.kron`). The reason for using the Kronecker product is that the Hadamard transform on multiple qubits is simply the tensor product of single-qubit Hadamard gates.

```
In [19]: # Define the Hadmard Gate
def create_H_gate(n):
    base_H = np.array([[1, 1],
                       [1, -1]]) * (1/np.sqrt(2))

    H_n = base_H
    for _ in range(n-1):
        H_n = np.kron(H_n, base_H)

    return H_n
```

Next lets define the Oracle Operator.

It serves for flipping the phase of the target state. Mathematically, an oracle for a search problem can be represented as a unitary matrix $U_{f(x)}$ that:

$$O|x\rangle = \begin{cases} -|x\rangle, & x \text{ is the target state} \\ |x\rangle, & \text{else} \end{cases} \quad (3)$$

In quantum computing literature, the oracle is often expressed in terms of a quantum Boolean function

$$O_{f(x)}|xy\rangle = |x, y \underline{\vee} f(x)\rangle \quad (4)$$

where $\underline{\vee}$ represent the XOR operation and y is typically an auxiliary qubit. In some literatues, XOR also use the notation of \oplus . However, for simplicity, in our simulation, we adop a simplified diagonal representation without explicitly using an auxiliary qubit.

For example, considering a 2-qubit system with the basis state order as $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. We can create a $4 * 4$ identity matrix as our Oracle Operator and each column corresponds exactly to a computational basis state in the above order. To mark the target state, we simply change the diagonal entry corresponding to that state from 1 to -1 . For example, an Oracle operator for a 2-qubit system with target state $|10\rangle$ should be:

$$O = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (5)$$

```
In [20]: # Define the Oracle Gate
def oracle_fx(n, f):
    """
    n: Number of qubits
    f: A Boolean function identifying the target state

    """
    # possible number of states
    N = 2**n
    O = np.eye(N)
    for i in range(N):
        # Flip the phase for the target state
        # e.g. if state |00> is the target state, then O[0,0] will become -1 in order to flip the phase of the first quantum state
        # This is just an easy implementation of the Oracle Gate
        bin_x = format(i, f'0{n}b')
        if f(bin_x):
            O[i, i] = -1
    return O
```

Then, let's go into the diffusion Operator:

Diffusion operator serves to inverse the amplitude of each basis state by the mean level of all phases. It is defined as

$$D = 2|s\rangle\langle s| - I$$

where $|s\rangle$ is defined as $H^{\otimes n}|0\rangle^{\otimes n}$ (\otimes means the tensor product) and I is an identity matrix.

To see why this operator works let's first see a simple example. Given a vector

$$a = [1, 1, -1, 1]$$

, we want to inverse every item by the mean. So we first calculate the mean of it, is $\bar{a} = (1 + 1 - 1 + 1)/4 = 0.5$. For the first item 1, we first calculate the distance between the mean and it as $1 - 0.5 = 0.5$, then use mean to minus this get $0.5 - 0.5 = 0$. So generally for the vector, the flip is

$$\hat{a} = \bar{a} - (a - \bar{a}) = 2 \times \bar{a} - a$$

We want the same effect happened on $|\phi\rangle$ which is the column vector record the amplitudes of all basis states.

The average amplitude can be calculate as:

$$\bar{a} = \frac{1}{N} \sum_{x=0}^{N-1} a_x \quad (6)$$

where $N = 2^n$.

And

$$|s\rangle = H^{\otimes n} |0\rangle^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

is the superposition with equal amplitude ($|s\rangle$ is a $N \times 1$ matrix, $\langle s|$ is a $1 \times N$ matrix). $|\phi\rangle$ can be generally represented by

$$|\phi\rangle = \sum_{x=0}^{N-1} a_x |x\rangle$$

Then $\langle s|\phi\rangle$ is the inner product as

$$\langle s|\phi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} a_x = \sqrt{N} \bar{a}$$

Then

$$\frac{1}{\sqrt{N}} \bar{a} |s\rangle = \sqrt{N} \bar{a} \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle = \bar{a} \sum_{x=0}^{N-1} |x\rangle$$

Then

$$2|s\rangle\langle s||\psi\rangle - |\psi\rangle = 2 \times \bar{a} \sum_{x=0}^{N-1} |x\rangle - \sum_{x=0}^{N-1} a_x |x\rangle$$

is just the same as the simple example we illustrated. By using the diffusion operator we inverse the amplitude by mean.

```
In [21]: # Strengthen the signal
def diffusion_strengthen(n):
    N = 2**n
    H = create_H_gate(n)
    s_ket = np.ones(N)/np.sqrt(N) # equal amplitude |x>
    D = 2*np.outer(s_ket, s_ket) - np.eye(N) # |x><x| - I
    return D
```

```
In [53]: def Gover_Algo(n,f, corr_pos = -3, iterations = None):
        """
        n: Number of qubits
```

```

f: A Boolean function identifying the target state
corr_pos: the correct pos of the target state, defaultly is -3 for target '101' in a 3-qubit system
iterations: how many iterations of diffusion operator you want, defaultly best iterations = int(np.round(np.pi / 4 * np.sqrt(2*n)))
"""
# Start the simulation
N = 2 ** n
H = create_H_gate(n)
O = oracle_fx(n, f)
D = diffusion_strengthen(n)

res = []

# initial state after a H gate for equal amplitude
# ATTENTION!! : Here the initialization is very flexible, what we need to ensure is the sum of the square is equal to 1
psi = H @ np.array([1]+[0]*(N-1))

# save the amplitude ** 2 at the iteration 0
res.append((np.abs(psi)**2)[corr_pos])

# Best Grover iteration
if iterations is None:
    iterations = int(np.round(np.pi / 4 * np.sqrt(2*n)))

print("Initial probability distribution: ", np.abs(psi)**2)

# Grover Algorithm
for i in range(iterations):
    psi = O @ psi
    psi = D @ psi
    print(f"The {i+1}th iteration 'probability distribution': {np.round(np.abs(psi)**2, 4)}")
    res.append((np.abs(psi)**2)[corr_pos])

# Final result
probabilities = np.abs(psi)**2
result_state = np.argmax(probabilities)

print(f"\n Final 'probability distribution': {np.round(probabilities,4)}")
print(f"Founded target state is: |{format(result_state, f'0{n}b')}>")

return res

```

Test a 3-qubit system

```

In [54]: def get_corr_pos(n, target:str):
        """
        target: your target state in str format
        """
        basis_states = []

```

```

for i in range(8):
    bin_x = format(i, f'0{n}b')
    basis_states.append(bin_x)

return basis_states.index(target)

```

```

In [72]: TARGET = '111'
# Here we define the f(x) that we want to evaluate
def f(x):
    if x == TARGET:
        return 1
    else:
        return 0

```

```

In [62]: # Test with best iterations
n = 3
corr_pos = get_corr_pos(n, TARGET)
res = Gover_Algo(n, f, corr_pos = corr_pos, iterations = None)

```

Initial probability distribution: [0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125]

The 1th iteration 'probability distribution': [0.0312 0.0312 0.0312 0.0312 0.0312 0.0312 0.0312 0.7812]

The 2th iteration 'probability distribution': [0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.9453]

Final 'probability distribution': [0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.9453]

Founded target state is: |111)

```

In [63]: # Test with 30 iterations
n = 3
corr_pos = get_corr_pos(n, TARGET)
res = Gover_Algo(n, f, corr_pos = corr_pos, iterations = 30)

```

```

Inital probability distribution: [0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125]
The 1th iteration 'probability distribution': [0.0312 0.0312 0.0312 0.0312 0.0312 0.0312 0.0312 0.0312 0.7812]
The 2th iteration 'probability distribution': [0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.0078 0.9453]
The 3th iteration 'probability distribution': [0.0957 0.0957 0.0957 0.0957 0.0957 0.0957 0.0957 0.0957 0.3301]
The 4th iteration 'probability distribution': [0.1411 0.1411 0.1411 0.1411 0.1411 0.1411 0.1411 0.1411 0.0122]
The 5th iteration 'probability distribution': [0.0646 0.0646 0.0646 0.0646 0.0646 0.0646 0.0646 0.0646 0.548 ]
The 6th iteration 'probability distribution': [0.    0.    0.    0.    0.    0.    0.    0.9998]
The 7th iteration 'probability distribution': [0.0604 0.0604 0.0604 0.0604 0.0604 0.0604 0.0604 0.0604 0.577 ]
The 8th iteration 'probability distribution': [0.1401 0.1401 0.1401 0.1401 0.1401 0.1401 0.1401 0.1401 0.0195]
The 9th iteration 'probability distribution': [0.0996 0.0996 0.0996 0.0996 0.0996 0.0996 0.0996 0.0996 0.3029]
The 10th iteration 'probability distribution': [0.0098 0.0098 0.0098 0.0098 0.0098 0.0098 0.0098 0.0098 0.9313]
The 11th iteration 'probability distribution': [0.0279 0.0279 0.0279 0.0279 0.0279 0.0279 0.0279 0.0279 0.8049]
The 12th iteration 'probability distribution': [0.1221 0.1221 0.1221 0.1221 0.1221 0.1221 0.1221 0.1221 0.145 ]
The 13th iteration 'probability distribution': [0.1277 0.1277 0.1277 0.1277 0.1277 0.1277 0.1277 0.1277 0.1063]
The 14th iteration 'probability distribution': [0.0348 0.0348 0.0348 0.0348 0.0348 0.0348 0.0348 0.0348 0.7566]
The 15th iteration 'probability distribution': [0.006 0.006 0.006 0.006 0.006 0.006 0.006 0.006 0.9578]
The 16th iteration 'probability distribution': [0.0917 0.0917 0.0917 0.0917 0.0917 0.0917 0.0917 0.0917 0.3578]
The 17th iteration 'probability distribution': [0.1419 0.1419 0.1419 0.1419 0.1419 0.1419 0.1419 0.1419 0.0066]
The 18th iteration 'probability distribution': [0.0687 0.0687 0.0687 0.0687 0.0687 0.0687 0.0687 0.0687 0.5188]
The 19th iteration 'probability distribution': [3.000e-04 3.000e-04 3.000e-04 3.000e-04 3.000e-04 3.000e-04 3.000e-04 3.000e-04 9.981e-01]
The 20th iteration 'probability distribution': [0.0563 0.0563 0.0563 0.0563 0.0563 0.0563 0.0563 0.0563 0.6057]
The 21th iteration 'probability distribution': [0.1388 0.1388 0.1388 0.1388 0.1388 0.1388 0.1388 0.1388 0.0283]
The 22th iteration 'probability distribution': [0.1034 0.1034 0.1034 0.1034 0.1034 0.1034 0.1034 0.1034 0.2764]
The 23th iteration 'probability distribution': [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.9157]
The 24th iteration 'probability distribution': [0.0246 0.0246 0.0246 0.0246 0.0246 0.0246 0.0246 0.0246 0.8276]
The 25th iteration 'probability distribution': [0.1191 0.1191 0.1191 0.1191 0.1191 0.1191 0.1191 0.1191 0.1661]
The 26th iteration 'probability distribution': [0.1301 0.1301 0.1301 0.1301 0.1301 0.1301 0.1301 0.1301 0.089 ]
The 27th iteration 'probability distribution': [0.0384 0.0384 0.0384 0.0384 0.0384 0.0384 0.0384 0.0384 0.7311]
The 28th iteration 'probability distribution': [0.0045 0.0045 0.0045 0.0045 0.0045 0.0045 0.0045 0.0045 0.9688]
The 29th iteration 'probability distribution': [0.0877 0.0877 0.0877 0.0877 0.0877 0.0877 0.0877 0.0877 0.3861]
The 30th iteration 'probability distribution': [0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.0027]

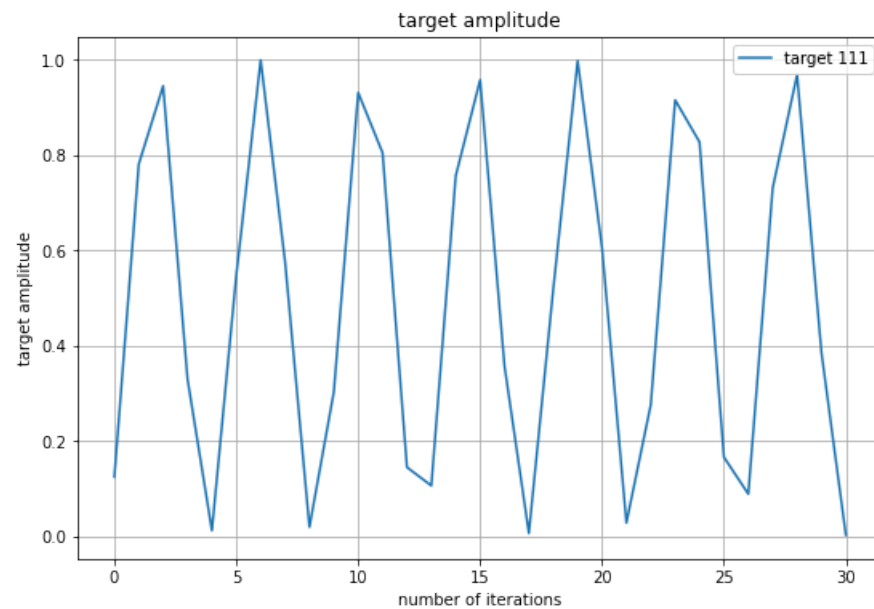
Final 'probability distribution': [0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.1425 0.0027]
Founded target state is: |000>

```

```

In [64]: # Visulization
plt.figure(figsize=(9, 6))
plt.plot(res, label = f'target {TARGET}')
plt.legend()
plt.grid(True)
plt.xlabel('number of iterations')
plt.ylabel('target amplitude')
plt.title('target amplitude')
plt.show()

```

Test a 2-qubit system

```
In [78]: TARGET = '01'
# Here we define the f(x) that we want to evaluate
def f(x):
    if x == TARGET:
        return 1
    else:
        return 0
```

```
In [79]: # Test with best iterations
n = 2
corr_pos = get_corr_pos(n, TARGET)
res = Gover_Algo(n, f, corr_pos = corr_pos, iterations = 1)
```

Initial probability distribution: [0.25 0.25 0.25 0.25]
 The 1th iteration 'probability distribution': [0. 1. 0. 0.]

Final 'probability distribution': [0. 1. 0. 0.]
 Founded target state is: |01>

```
In [80]: # Test with 30 iterations
n = 2
corr_pos = get_corr_pos(n, TARGET)
res = Gover_Algo(n, f, corr_pos = corr_pos, iterations = 30)
```

```

Inital probability distribution: [0.25 0.25 0.25 0.25]
The 1th iteration 'probability distribution': [0. 1. 0. 0.]
The 2th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 3th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 4th iteration 'probability distribution': [0. 1. 0. 0.]
The 5th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 6th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 7th iteration 'probability distribution': [0. 1. 0. 0.]
The 8th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 9th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 10th iteration 'probability distribution': [0. 1. 0. 0.]
The 11th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 12th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 13th iteration 'probability distribution': [0. 1. 0. 0.]
The 14th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 15th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 16th iteration 'probability distribution': [0. 1. 0. 0.]
The 17th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 18th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 19th iteration 'probability distribution': [0. 1. 0. 0.]
The 20th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 21th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 22th iteration 'probability distribution': [0. 1. 0. 0.]
The 23th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 24th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 25th iteration 'probability distribution': [0. 1. 0. 0.]
The 26th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 27th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 28th iteration 'probability distribution': [0. 1. 0. 0.]
The 29th iteration 'probability distribution': [0.25 0.25 0.25 0.25]
The 30th iteration 'probability distribution': [0.25 0.25 0.25 0.25]

Final 'probability distribution': [0.25 0.25 0.25 0.25]
Founded target state is: |00>

```

```

In [81]: # Visulization
plt.figure(figsize=(9, 6))
plt.plot(res, label = f'target {TARGET}')
plt.legend()
plt.grid(True)
plt.xlabel('number of iterations')
plt.ylabel('target amplitude')
plt.title('target amplitude')
plt.show()

```

