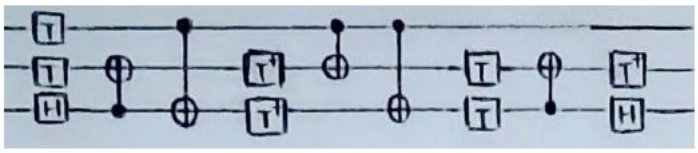
4108056052 資工三 徐幸妤

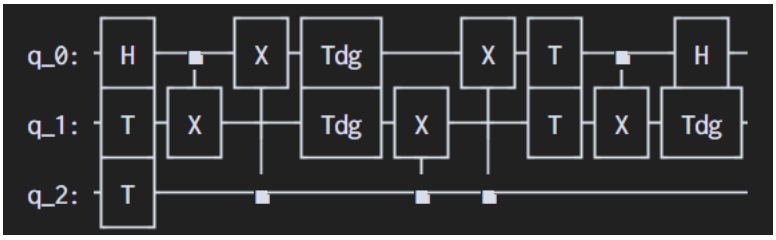
程式

- 此次作業的實作方式主要是在 python 裡 import qiskit 等相關套件來達成。
  做法是將本次題目的邏輯閘集合編碼進量子啟發式演算法 KNQTS 中，再透過它來搜索出正確的電路。
- 但由於從頭讓演算法自行去尋找的話，均無法找到正確的電路(會有兩個 output 一直換不過來)，因此一開始都會透過作業說明檔注意事項裡不能重複的那些電路來做導引。
- 而在 qiskit 套件中的輸入方式和我們平常使用的不同，所以一開始輸入電路時會剛好相反
  例如:說明檔中不能重複的(b)長這樣



(b)

那在程式中輸入的話就會長這樣



驗證電路正確與否的方法

本次作業中我有兩支程式(因為有些電路是我自行生成的有些不是)，一支是用來判斷電路是否正確的 compare_circuit，另一支則是可以利用演算法找出正確的程式，兩者判斷電路的方法都雷同。
首先，我們可以利用 Operator() 的方式，將電路輸入後取得它的 operator 並印出，再和作業的說明文件做比對便可判斷電路是否正確(例如下圖便是上面電路(b)印出的 operator)
註:因檔名規定，繳交時的 compare_circuit 程式為 hw1-1，合成電路程式為 hw1-2

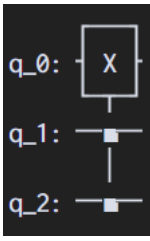
```
Operator([[1.00000000e+00+0.j, 1.11022302e-16+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j],
          [1.11022302e-16+0.j, 1.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 1.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           1.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 1.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           1.00000000e+00+0.j, 0.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 1.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j],
          [0.00000000e+00+0.j, 0.00000000e+00+0.j, 0.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j, 1.00000000e+00+0.j,
           0.00000000e+00+0.j, 0.00000000e+00+0.j]],
```

圖中的每一個[]中都是一個 row，一個個比對即可。

但因為這樣會對的很累，所以後來用了另一個方式。我發現如果使用 Variations of CN Gate 的 library 輸入電路的話，operator 就會容易看許多(如下圖)

```
Operator([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j]],
```

所以我先查找了作業中每一個矩陣代表的電路名稱，接著再找了他們在 Variations of CN Gate library 中的表示方式。例如: (a)為 Toffoli gate 的矩陣，所以輸入該形式後可以得到和整齊的 operator




```
Operator([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j]],
```

而(b)~(e)亦是藉由相同方式得到 operator。

接下來會將欲判斷正確與否的電路輸入，並和上面的 operator 的每一列做比對

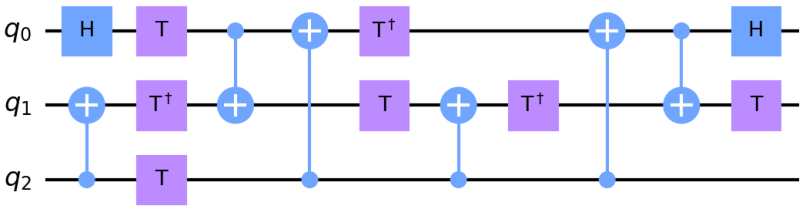
```
    # generate 000 ~ 111 input
    for a in range(pow(2, n)):
        possible_input.append(str(decToBin(a)).zfill(3))
        for b in range(pow(2, n)):
            circuit_op.data[a][b] = np.round(
                circuit_op.data[a][b], 2)  # 四捨五入目前矩陣
        # 比較每一列看是否相同
        if (target_op.data[a] == circuit_op.data[a]).all():
            COP += 1
```

參數 COP 是在計算輸入(000~111)相同的數目，如果兩者完全相同的話，COP = 8，驗證為正確的電路。

```
8
```

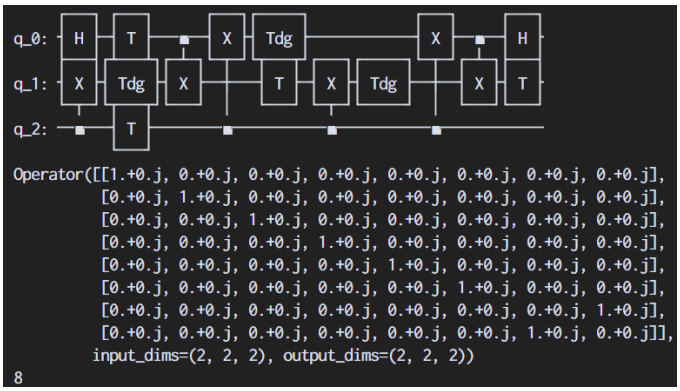
找到的電路

(a) 在做前面步驟中時，有發現(a)為 Toffoli 的矩陣而(c)為 Fredkin 的矩陣，接著從 https://www.researchgate.net/figure/Using-simplifying-transformations-to-prove-that-Fredkin-gate-is-its-own-inverse_fig15_220637922 可以知道 Fredkin 是由一個 Toffoli 前後夾 CNOT 所構成的，反之若是 Fredkin 前後拿掉 CNOT 便是 Toffoli，所以我將作業說明檔(c)中前後的 CNOT 拿掉後**得到(a)的電路**。

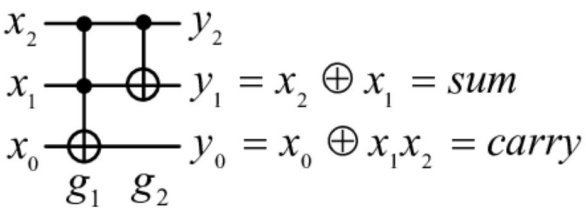


Gate Count	Depth	T-Count	T-Depth
15	10	7	4

確定正確

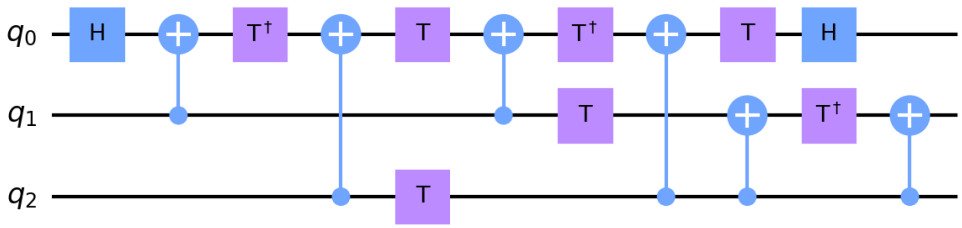


(b) 在做前面步驟中時，有發現(b)為 Peres 的矩陣，接著從 https://www.researchgate.net/figure/A-Peres-gate-or-a-reversible-half-adder-circuit_fig1_290266844 這個網站中可以得知(b)就是在(a)的電路後面多加一個 CNOT 邏輯閘。



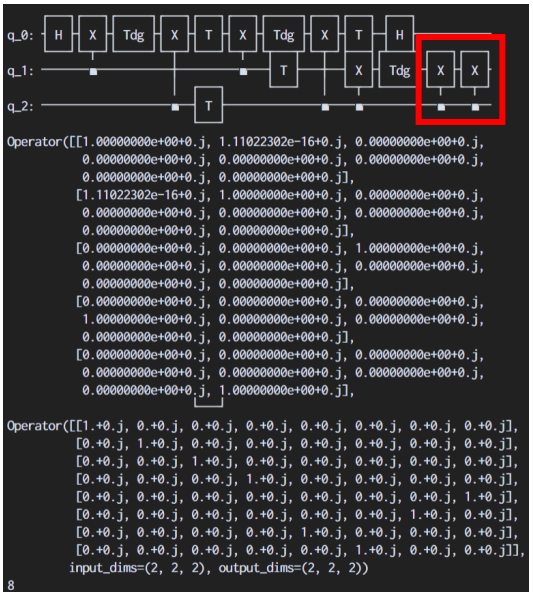
A Peres gate or a reversible half-adder circuit

所以我先透過演算法找到了這組(a)的電路

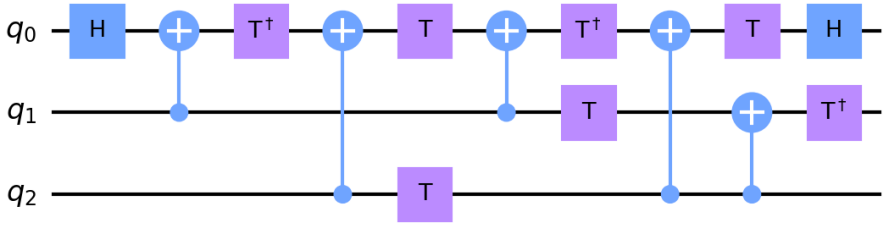


接著再電路的最後面加上 CNOT 邏輯閘，便可以找到(b)的電路

驗證電路正確

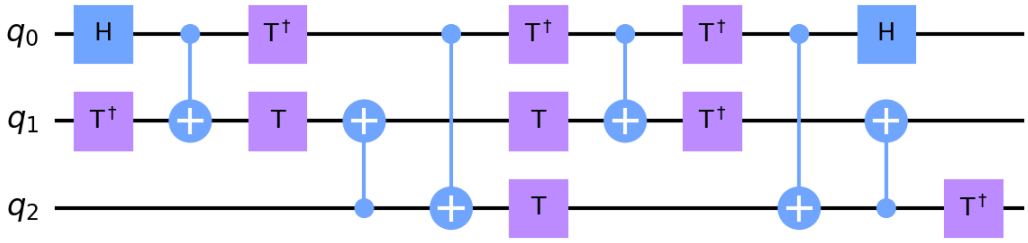


因為紅框中通過兩次一樣的邏輯閘所以可以消掉它，並得到**最終(b)的電路**



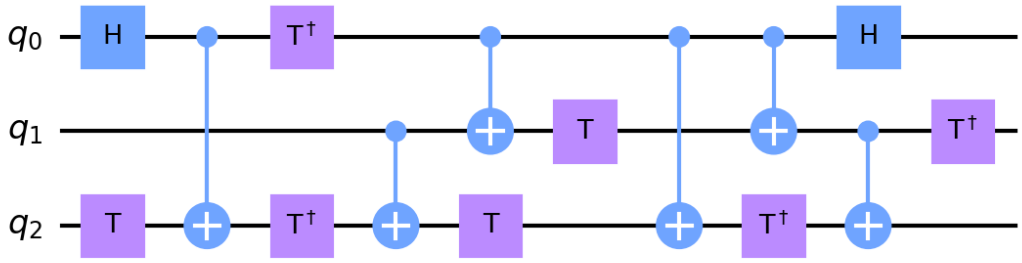
Gate Count	Depth	T-Count	T-Depth
14	10	7	5

(d) 使用演算法**合成出的電路(d)**，結果如下



Gate Count	Depth	T-Count	T-Depth
17	11	9	5

(e) 使用演算法**合成出的電路(e)**，結果如下



Gate Count	Depth	T-Count	T-Depth
15	10	7	6

註: 此處可以將 q2 的第三個 T gate 往右一格，則 T-Depth 變為 5

合成電路的方法

1. **Import 所需的函式庫、參數設定**: 需用的函式庫與參數相關設定如上圖所示。在這邊將 NOT、control、T、Tdg、H 和無作用的複製位元依序編碼為 0、1、2、3、4、5，而在紅框部分可以自行選擇要合成的電路，而編碼為

   (a) toffoli (b) peres (c) fredkin (d) or (e) toffoli with a 0-control (f) hhl

```
import random

import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Operator

# Parameter for KNQTS
population = 114
loop = 5000
DELTA = 0.00133
mu = 1.01
global last_HD
global Q  # prob.matrix
global gb, gw

# Parameter for exp
rand_seed = 100
test = 5

# Parameter for circuit
depth = 15
n = 3  # num of qubit → top qubit = rightmost
global target  # target gate
GATE_SET = {0: 'cx', 1: 'control', 2: 't', 3: 'tdg', 4: 'h', 5: 'copy_bit'}
TARGET = 'fredkin'
pi = 3.14
```

2. **class Circuit**: 這個 class 會用來產生與紀錄所有 circuit 相關的內容，例如多少位元、深度、fitness 等相關資訊。其中要特別注意的是，一開始在生成電路時，是利用亂數選擇值放在

measure 這個 list 中，等到修復完成確定電路合理了，才會透過 genCircuit()這個 function 生成真正的量子電路。

```
class Circuit:
    def __init__(self):
        self.n = n
        self.deep = depth
        self.gate_count = 0
        self.op = ''
        self.fit = 0
        self.correct = False
        self.diff = list()
        self.circuit = QuantumCircuit(n)
        self.measure = list()
        self.exp = ''
        self.i = ''
        for j in range(n):
            self.measure.append(list())

    # generate the circuit and cnt gate count
    def genCircuit(self):...

    # copy best ans
    def copy(self, a):...

    def print(self):...

    def __repr__(self):...
```

3. **主函式**: 這邊一開始會先初始化一些實驗的資訊。其中比較特別的是我會利用 target_op 來記錄目前我們想要的矩陣輸出為何，而 target_gate()的內容便是透過生成 Variations of CN Gate library 電路的方式來取得 Operator 並回傳(如同前面**驗證電路正確與否的方法**中所描述的)

```
if __name__ == '__main__':
    random.seed(rand_seed)
    target_op = target_gate()  # get target operator(matrix)
    target = Circuit()
    expBest = Circuit()
    expBest.fit = float('-inf')
    x = circuit_list()  # generate circuit (null)
```

註: x 為我在實驗中每一代產生的所有電路

再來迴圈中的內容便是合成電路。首先我們會先初始化實驗所需的相關資訊，Q 為 KNQTS 演算法中用以紀錄選中每個元件資訊的機率矩陣，因為總共有 6 種可能所以先全部初始為 1/6；而 delta 和 last_HD 均為在 KNQTS 中用以調整 Q 矩陣的參數值。gb、gw、lb、lw 分別代表歷史最佳解、歷史最差解、當代最佳解及當代最差解。

```
    for expTime in range(test):

        # ↓ initialize ↓ #
        delta = DELTA  # init delta value
        Q = np.full((n, depth, len(GATE_SET)), 1 / 6)  # init prob. matrix Q
        last_HD = float('inf')
        gb = Circuit()
        gw = Circuit()
        lb = Circuit()
        lw = Circuit()
        gb.fit = float('-inf')
        gw.fit = float('inf')
        # ↑ initialize ↑ #
```

接著對於每次實驗我們都會執行 5000 代。在每一代都會做

(1) initx: 初始化此代的電路 (因為每一代都會重新產生新的電路)

```
def initx(x, expTime, ecx):
    for p in range(population):
        x[p] = Circuit()
        x[p].exp = expTime
        x[p].i = ecx
```

(2) measure: 透過產生亂數的方式來決定每一個位置需要選中的 gate

```
# measure and build the circuit
def measure(x):
    for p in x:
        for a in range(n):  # through n bit
            for b in range(depth):  # through each line
                r = random.random()  # 0~1
                # ↓ measure ↓ #
                if r <= Q[a][b][0]:
                    p.measure[a].append(GATE_SET[0])
                elif Q[a][b][0] < r <= (Q[a][b][0] + Q[a][b][1]):
                    p.measure[a].append(GATE_SET[1])
                elif (Q[a][b][0] + Q[a][b][1]) < r <= (Q[a][b][0] + Q[a][b][1] + Q[a][b][2]):
                    p.measure[a].append(GATE_SET[2])
                elif (Q[a][b][0] + Q[a][b][1] + Q[a][b][2]) < r <= (Q[a][b][0] + Q[a][b][1] + Q[a][b][2] + Q[a][b][3]):
                    p.measure[a].append(GATE_SET[3])
                elif (Q[a][b][0] + Q[a][b][1] + Q[a][b][2] + Q[a][b][3]) < r <= (Q[a][b][0] + Q[a][b][1] + Q[a][b][2] +
                                                                                 Q[a][b][3] + Q[a][b][4]):
                    p.measure[a].append(GATE_SET[4])
                else:
                    p.measure[a].append(GATE_SET[5])
```

(3) repair: 由於(2)是隨機產生，所以可能會有一些不合理的 gate，在這部分會做修復

(4) fitness: 會先將(3)中修復後的電路轉為真正的量子電路，接著在判斷電路的好壞

(5) update: 根據(4)的資訊調整 Q 矩陣中的機率分布

```
            initx(x, expTime, ecx)
            measure(x)  # built circuit
            repair(x)

            if ecx == 0:
                presetAns(x[0])
                target.copy(x[0])

            fitness(x, target_op)
            last_HD = update(x, gb, gw, lb, lw, ecx, expTime, last_HD, delta)

        if gb.fit > expBest.fit:
            expBest.copy(gb)
```

紅框部分說明:由於從頭讓演算法自行去尋找的話，均無法找到正確的電路(會有兩個 output 一直換不過來)，因此一開始都會透過作業說明檔注意事項裡不能重複的那些電路來做導引。

(3) repair: 此處主要是在修復 CNOT gate。我將不合理的情形分為以下三種:

Case1: 有 control 也有 not

- 如果有兩個 control 的話，保留選中 control 機率最高者
- 如果有兩個 not 的話，保留選中 not 機率最高者

而被換掉的會換成該位置中，其他選中機率最高的可能

Case2: 有 control 沒有 not

- 若其他線路中機率最高的 gate 是 not 的話，就將它會成 not
- 如果沒有的話，就將 control 換成它機率最高的可能
- 如果 control 即為這個位置的最佳選擇，無法換掉的話，那就隨機將其他線路換成 not

Case3: 有 not 沒有 control

- 若其他線路中機率最高的元件是 control 的話，就將它會成 control
- 如果沒有的話，就將 not 換成它機率最高的可能
- 如果 not 即為這個位置的最佳選擇，無法換掉的話，那就隨機將其他線路換成 control

(4) fitness: 此處將判斷電路好壞的方式分為兩部分:

- fit1 表示的是正確程度，這邊會透過比較產生電路與目標電路的 Operator 來判斷好壞，相同的列數越多，表示此電路越正確
- fit2 表示此電路的成本，我是透過計算 gate count 來當成本，數目越低就表示此電路越好
- 最終適應值函數為兩項相加，當 fit1 確定電路正確時才會啟動 fit2

```
def fitness(x, target_op):
    for idx, p in enumerate(x):
        w = 0  # use fit 2 or not

        p.genCircuit()

        COP, p.diff = cntCOP(target_op, p.op)
        WG = (n * depth) - p.gate_count
        fit1 = COP / pow(2, n)
        fit2 = WG / (n * depth)

        if fit1 == 1:
            w = 1
            p.correct = True

        p.fit = fit1 + w * fit2
```

(5) update: 此處主要會做三件事:

- 找到當代最佳解與當代最差解，並更新歷史最佳解和最差解

```
    # find local best and local worst
    lb.copy(x[0])
    lw.copy(x[-1])
    for p in x:
        if p.fit >= lb.fit:
            lb.copy(p)
        if p.fit <= lw.fit:
            lw.copy(p)

    recordAndUpdate(gb, gw, lb, lw, ecx, expTime)
```

註:在 recordAndUpdate 的階段，我會將找到的正確電路印出

- 透過計算此代最佳解和最差解間的長相差異，並和上代比較來得知目前的收斂程度，藉此調整 delta 值

```
    # ↓ adjust delta ↓ #
    # cnt ham
    HD = 0
    for a in range(n):
        for b in range(depth):
            if lb.measure[a][b] != lw.measure[a][b]:
                HD += 1
    last_HD = HD if last_HD == float('inf') else last_HD

    if HD > last_HD:  # 差異變大
        delta *= mu
    elif HD < last_HD:  # 差異變小
        delta *= (2 - mu)
    else:
        pass
```

- 更新 Q 矩陣，會增加選中 gb 所選中值的機率，同時也減少選中 gw 所選中值的機率

```
if gb.measure[a][b] != gw.measure[a][b]:
    try:
        gb_index = list(GATE_SET.values()).index(gb.measure[a][b])
        gw_index = list(GATE_SET.values()).index(gw.measure[a][b])
        Q[a][b][gb_index] += delta
        Q[a][b][gw_index] -= delta

        # repair prob
        if Q[a][b][gw_index] <= 0:
            Q[a][b][gw_index] = 0

            Q[a][b][gb_index] = 1
            for c in range(len(GATE_SET)):
                if c != gb_index:
                    Q[a][b][gb_index] -= Q[a][b][c]

        # Quantum NOT
        if Q[a][b][gb_index] < Q[a][b][gw_index]:
            Q[a][b][gb_index], Q[a][b][gw_index] = Q[a][b][gw_index], Q[a][b][gb_index]
```

註 1: 在調整機率時有可能扣到剩下負的，會造成機率分布有問題，故若有小於 0 的情況便會做修復

註 2: 若今天選中最差解的機率比選中最佳解還高時，我就會啟動 Quantum NOT 的機制，將兩者的機率做互換，幫助導引至最佳解

最後，我會將最終實驗結果的最佳電路印出，然後從(5)中產生的電路及此處的最佳電路中挑選出最合適的電路。

```
print("__________Finish__________")
expBest.print()
expBest.circuit.draw('mpl', filename="KNQTS" +
                     str(expBest.fit) + "_FinalCircuit.png")
print(expBest.op)
```

補充說明:使用 compare_circuit 程式的方法

在最上面 TARGET 中輸入要比對的電路名稱(名稱設定同**合成電路的方法**中的電路編碼)，接著在 presentAns()此 function 中找到對應的位置將電路輸入，執行後看最下面印出數目是否為 8 即可。