程式碼架構

架構: 本次作業將題目編碼為組合最佳化問題，在假設最多需要m個邏輯閘的情況下，給予每個量子位元選擇 0-control、1-control、NOT 邏輯閘或三者皆非的複製位元之機率，藉由 KNQTS 演算法搜尋出合法的最佳電路，表 1 為 KNQTS 演算法流程，其參數定義為：g表示目前的世代數，P_{total}表示每一代所產生的解(電路)數目，$delta$則為在 KNQTS 演算法中，用來調整機率矩陣的自適應參數值。

KNQTS 演算法流程
1. 初始化世代數$g \leftarrow 0$
2. 初始化機率矩陣 Q(g)
3. 初始化歷史最佳解 gb 與歷史最差解 gw
4. **while** (尚未達成終止條件) **do**
5. $g \leftarrow g + 1$
6. 根據 Q(g-1)測量產生解
7. 修復不合理的解 (一個量子邏輯閘中具有多個 NOT 者)
8. 計算適應值
9. 找出當代最佳解 sb、當代最差解 sw 並更新歷史最佳解 gb、歷史最差解 gw
10. 計算出 sb 與 sw 的漢明距離
11. 調整$delta$值
12. 依據歷史最佳解 gb、歷史最差解 gw 更新機率矩陣 Q(g)
13. **end while**

表 1：KNQTS 演算法流程

編碼: 假設一個 n-bit 的電路最多由m個量子可逆邏輯閘所組成，而 G 代表m個邏輯閘中所有可能的廣義 Toffoli 邏輯閘之集合。因為是 n-bit 的電路，所以每一個廣義的 Toffoli 邏輯閘均由 n 顆量子位元所組成。而每顆量子位元皆為 0-control、1-control、NOT 邏輯閘和複製位元(非控制位元與目標位元者)的疊加狀態。

使用公式

$$|q\rangle \equiv |q\rangle_{12} \equiv |q\rangle_1 \otimes |q\rangle_2 = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

將兩顆單顆量子位元表示成一組量子位元$\left|q^p{}_{ij}\right\rangle$，其中$i$表示一個邏輯閘中的第$i$顆量子位元，$j$表示一組電路中的第 j 個邏輯閘，p 則表示在一個世代中的第p組電路。其中$i = 1,2,3, \dots n, j = 1,2,3 \dots m, p = 1,2,3, \dots P_{total}$。

根據量子的特性，測量前的量子位元會是所有可能的疊加態，而測量過後就會陷落在特定的狀態，為了方便區分，編碼時我們將測量前的量子位元稱為$\left|q^p{}_{ij}\right\rangle$，測量後的量子位元稱為$c^p{}_{ij}$。 KNQTS 演算法會對每一組$\left|q^p{}_{ij}\right\rangle$做測量，如圖 1 所示，每組量子位元都各自有$\|\alpha\|^2$、$\|\beta\|^2$、$\|\gamma\|^2$、$\|\delta\|^2$的機率陷落到$|00\rangle$、$|01\rangle$、$|10\rangle$、$|11\rangle$ 的狀態，將這些狀態表示成十進制的話則可分別代表「0」、「1」、「2」、「3」。當$c^p{}_{ij} = 0$時，代表 0-control 的控制位元；$c^p{}_{ij} = 1$時代表 1-control 的控制位元；$c^p{}_{ij} = 2$時代表無作用的複製位元；而$c^p{}_{ij} = 3$時則代表 NOT 邏輯閘。

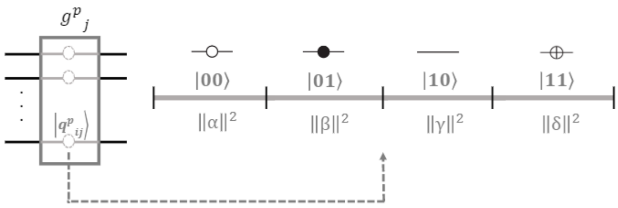


圖 1：量子位元編碼示意圖

對電路中的邏輯閘編碼時，會將第 j 個邏輯閘編碼為$g^{p}{}_{j}\left(c^{p}{}_{1j}, c^{p}{}_{2j}, \ldots c^{p}{}_{ij} \ldots, c^{p}{}_{nj}\right)$，其中$g^{p}{}_{j} \in G, c^{p}{}_{ij} \in \{0,1,2,3\}$ (圖 2)。

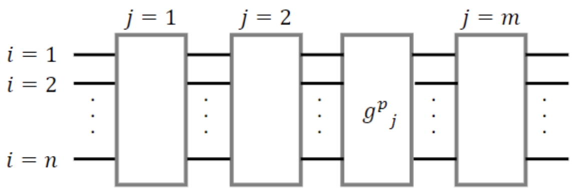


圖 2：電路編碼表示圖

程式碼流程

- <u>相關參數設定</u>:此處主要為 KNQTS 演算法所需的相關參數，以及一些實驗的相關設定。Q 表示演算法中的機率矩陣，而 X 為每個世代所產生的所有電路，fit 則是用來記錄每組電路的適應值。

```
// parameter for KNQTS and Exp
#define rand_seed 114
#define population 100 // population 100
#define loop 5000      // generation 5000
#define test 50
#define delta 0.002
#define delta_change 0.001
#define mMAX 70
#define nMAX 10
#define FunctionNum 1

int m = 30, n = 7;

bool changeBest = false;
/* about Q matrix */
double Q[nMAX][mMAX][4] = {0};

int x[population][nMAX][mMAX] = {0};

/* about fitness */
double fit[population] = {0};
int best = 0, worst = 0; // population of best and worst
double b = 0.0, w = 100;
int gb[nMAX][mMAX] = {0}, gw[nMAX][mMAX] = {0};
int bestAns = 2000000; // Find the best ans in the 50Exp

vector<int> output;
int allgb[nMAX][mMAX] = {0};
int mingb[nMAX][mMAX] = {0};

// about parameter of KNQTS
int last_ham = INT_MAX;
double adaptive_delta = delta;
```

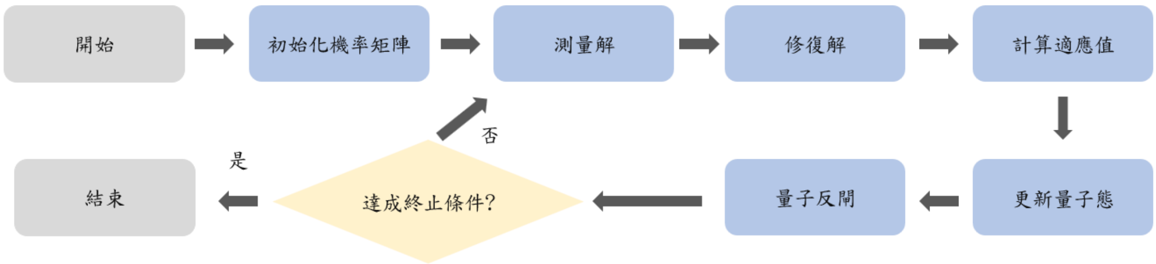
註:前文中的P_{total}在程式碼中為參數 population

- <u>主函式</u>: 首先可以藉由輸入函數來選擇要找的函數和預設的m值。

```
int main()
{
    input_target();
    srand(rand_seed);
    int total = 0;
    int generation = 0;

    cout << "m = ";
    cin >> m;
```

接下來便是 KNQTS 演算法的主要流程，全部流程如下圖所示，後面會一一解釋各個步驟。



➢ *初始化機率矩陣*

概念: 首先，我們需要初始化一個 $n \times m$ 的機率矩陣 Q(0)(圖 3)，裡面的每一個元素$Prob_{ij}$=[$\|\alpha_{ij}\|^2$ $\|\beta_{ij}\|^2$ $\|\gamma_{ij}\|^2$ $\|\delta_{ij}\|^2$]用以紀錄$n \times m$個$\left|q^p{}_{ij}\right\rangle$的機率狀態分布。而矩陣中的每一行都可以對應到一個邏輯閘。

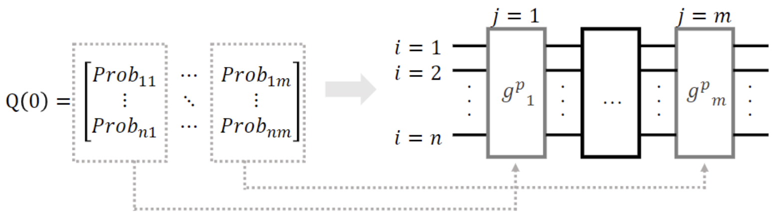


圖 3：機率矩陣 Q(0)與電路的對應

此外，在初始化階段時會將$\|\alpha_{ij}\|^2$、$\|\beta_{ij}\|^2$、$\|\gamma_{ij}\|^2$、$\|\delta_{ij}\|^2$初始化為0.25，表示選中 0,1,2,3 的機率皆相等。

程式碼:

```
void init()
{
    /* initialize Q matrix */
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            for (int k = 0; k < 4; k++)
            {
                Q[i][j][k] = 0.25;
            }
        }
    }
}
```

➢ *測量(產生)解*

概念: 再來我們會產生P_{total}組亂數矩陣，用以代表隨機產生的電路。對每組電路我們會隨機產生$n \times m$個範圍在[0, 1]之間的隨機變數$r^p{}_{ij}$，而其測量結果如下所示。

$$c^p{}_{ij} = \begin{cases} 0, & 0 \le r^p{}_{ij} \le \|\alpha_{ij}\|^2 \\ 1, & \|\alpha_{ij}\|^2 < r^p{}_{ij} \le \|\alpha_{ij}\|^2 + \|\beta_{ij}\|^2 \\ 2, & \|\alpha_{ij}\|^2 + \|\beta_{ij}\|^2 < r^p{}_{ij} \le \|\alpha_{ij}\|^2 + \|\beta_{ij}\|^2 + \|\gamma_{ij}\|^2 \\ 3, & \|\alpha_{ij}\|^2 + \|\beta_{ij}\|^2 + \|\gamma_{ij}\|^2 < r^p{}_{ij} \le 1 \end{cases}$$

程式碼:

```
void ans()
{
    for (int i = 0; i < population; i++) // population
    {
        for (int j = 0; j < n; j++) // each component
        {
            for (int k = 0; k < m; k++)
            {
                double r = (double)rand() / RAND_MAX;   // create random number

                /* 0 ??? 0-control, 1 ??? 1-control, 2 ??? copy bit, 3 ??? not */
                if (r <= Q[j][k][0])
                {
                    x[i][j][k] = 0;
                }
                else if (r > Q[j][k][0] && r <= (Q[j][k][0] + Q[j][k][1]))
                {
                    x[i][j][k] = 1;
                }
                else if (r > (Q[j][k][0] + Q[j][k][1]) && r <= (Q[j][k][0] + Q[j][k][1] + Q[j][k][2]))
                {
                    x[i][j][k] = 2;
                }
                else
                {
                    x[i][j][k] = 3;
                }
            }
        }
    }
}
```

➢ *修復解*

概念: 在上一步隨機生成電路時，我們可能產生一個邏輯閘中有不只一個目標位元的不合法現象(圖 4)，此步驟中會參考機率矩陣 Q 中的機率分布來修復。首先，會先挑出此類邏輯閘中所有選中 NOT 元件的位元，接著比較它們選中 NOT 元件的機率，將機率最大的保留，並將剩餘的替換為除了 NOT 元件之外，選中機率最高的元件。

圖 4: 測量解時產生的不合理邏輯閘

程式碼:

```
void repair()
{
    for (int i = 0; i < population; i++)
    {
        for (int j = 0; j < m; j++) // gate
        {
            int maxIndex = -1;
            double maxProb = 0;
            for (int k = 0; k < n; k++)
            {

                if (x[i][k][j] == 3) // not
                {

                    /* find the max prob. of not */
                    if (Q[k][j][3] > maxProb)
                    {
                        /* repair last not */
                        if (maxIndex != -1) // not first
                        {
                            double max = 0.0;
                            int index = 0;
                            for (int a = 0; a < 3; a++)
                            {
                                if (Q[maxIndex][j][a] > max)
                                {
                                    index = a;
                                    max = Q[maxIndex][j][a];
                                }
                            }
                            x[i][maxIndex][j] = index;
                        }

                        maxIndex = k;
                        maxProb = Q[k][j][3];
                    }
                    else
                    {
                        double max = 0.0;
                        int index = 0;
                        for (int a = 0; a < 3; a++)
                        {
                            if (Q[k][j][a] > max)
                            {
                                index = a;
                                max = Q[k][j][a];
                            }
                        }
                        x[i][k][j] = index;
                    }
                }
            }
        }
    }
}
```

此處的程式碼較為繁瑣，故在此附上這個 function 的 pseudocode

Procedure Repair

begin

初始化所有選中 not 位元的${c^p}_{ij}$裡選中 not 機率最高的 maxProbIndex ← −1

初始化所有選中 not 位元的${c^p}_{ij}$裡選中 not 機率最高的 maxProb ← 0

for all ${c^p}_{ij} \in {g^p}_j$ **do**

  if ${c^p}_{ij} = 3$ **then**

    if $\|\delta_{ij}\|^2$ > maxProb **then**

      if maxProbIndex $\neq -1$ **then**

$c_{maxProbIndexj} \leftarrow$ $Prob_{ij}$中 $\max\left\{\|\alpha_{maxProbIndexj}\|^2、\|\beta_{maxProbIndexj}\|^2、\|\gamma_{maxProbIndexj}\|^2\right\}$的索

      end if

      maxProbIndex ← i

      maxProb ← $\|\delta_{ij}\|^2$

    else

      ${c^p}_{ij} \leftarrow Prob_{ij}$中 $\max\left\{\|\alpha_{ij}\|^2、\|\beta_{ij}\|^2、\|\gamma_{ij}\|^2\right\}$的索引值

    end else

    end if

  end if

end for

end

橘色字體部分是在尋找目前選中 NOT 元件機率最高者

綠色字體部分則是此位置選中 NOT 的機率不是最高的，所以會被換成除了 NOT 之外機率最高者

- *計算適應值*

概念: 適應值函數為 KNQTS 演算法中幫助我們更有效率找到最佳解的關鍵。而在本次作業中主要研究在給定的條件下，如何以最小成本完成電路，故我們必須考慮一個電路的正確性及其成本代價。

此處將第p^{th}個電路的適應函數分為 fit_1^p 和 fit_2^p。fit_1^p用來判斷目前電路的正確程度，定義如下：

$$fit_1^p = \mathrm{COP}/2^n$$

其中 COP(Correct Output-bits)代表電路和可逆函數值做比較後的正確輸出數目，而2^n表示可逆函數的總輸出數，例如 3-bit 的電路會有$2^3 = 8$個輸出。

而fit_2^p用來判別解的好壞，在量子可逆電路中，若一個邏輯閘不具有目標位元 NOT，便無法構成一個量子邏輯閘，我們將邏輯閘的數目加總後稱為 WG(Wire Gate)，並定義fit_2^p為：

$$fit_2^p = \mathrm{WG/m}$$

代表在假設最多需要m個邏輯閘的情況下，有 WG 個邏輯閘是沒有作用，可以被消除的。因此僅需要$m - WG$個邏輯閘便可完成電路。

接著我們會將fit_1^p和fit_2^p合併成適應函數Fit^p，其定義為：

$$Fit^p = fit_1^p + W^p \times fit_2^p$$

程式碼:

```
void fitness()
{
    for (int i = 0; i < population; i++)
    {
        double fit1 = 0.0, fit2 = 0.0, w1 = 1, w2 = 0.0;
        int COP = cntCOP(i);
        int gate = cntGate(i);
        int WG = m - gate;

        fit1 = (double)COP / (double)pow(2, n);
        if (fit1 == 1)
        {
            w2 = 0.4;
        }

        fit2 = 0.0;

        /* count fit2 */
        if (gate == 0) // not correct -> change to a big number
        {
            fit2 = (double)(1 - WG);
        }
        else
        {
            fit2 = (double)WG / (double)m;
        }

        fit[i] = w1 * fit1 + w2 * fit2;
    }
}
```

這裡的 cntCOP 作法是:將 0~2^n依序作為輸入到電路(透過 correct function)中，再判斷輸出結果是否與最初輸入的 output 相同。

```
int cntCOP(int indexOfx)
{
    int cnt = 0;
    for (int i = 0; i < pow(2, n); i++)
    {
        if (correct(indexOfx, i, output[i])) // is correct
        {
            cnt++;
        }
    }

    return cnt;
}
```

correct function:將輸入的數字轉成 2 進制，透過程式模擬經過電路的過程，再將最後的輸出換成 10 進制來和 output 比對

```
bool correct(int indexOfx, int in, int out)
{
    int *qin = toBinary(in);

    for (int i = 0; i < m; i++) // through m gates
    {
        int hasNOT = -1;             // whether it has not or not and it's index
        bool chg = true;             // change the not bit or not
        for (int j = 0; j < n; j++) // through n bits of a gate
        {
            if (x[indexOfx][j][i] == 0 || x[indexOfx][j][i] == 1)
            {
                if (qin[j] != x[indexOfx][j][i]) // not map the gate ??? not change
                {
                    chg = false;
                    break;
                }
            }
            else if (x[indexOfx][j][i] == 3) // has not gate
            {
                hasNOT = j;
            }
        }

        if ((hasNOT >= 0) && chg)
        {
            qin[hasNOT] = !qin[hasNOT];
        }
    }
    if (toDecimal(qin) == out)
    {
        delete qin;
        return true;
    }

    delete qin;
    return false;
}
```

註:紅框部分為模擬電路通過的過程

➢ 更新量子態

概念與程式碼: 此步驟會先根據適應值選出當代最佳解 sb 和當代最差解 sw，並和歷史最佳解 gb 與最差解 gw 進行更新。

$$\begin{cases} gb = sb \quad if\ Fit^{sb} \geq Fit^{gb} \\ gw = sw \quad if\ Fit^{sw} \leq Fit^{gw} \end{cases}$$

```
/* ??? find local best(sb) and local worst(sw) ??? */
double max = 0, min = 100;
int sb = 0, sw = 0;

for (int i = 0; i < population; i++)
{
    /* find best */
    if (fit[i] >= max)
    {
        max = fit[i];
        sb = i;
    }

    /* find worst */
    if (fit[i] <= min)
    {
        min = fit[i];
        sw = i;
    }
}
/* ??? find local best(sb) and local worst(sw) ??? */
```

```
/* ??? update global value b and w ??? */
if (max >= b)
{
    changeBest = true;
    b = max;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            gb[i][j] = x[sb][i][j];
        }
    }
}

if (min <= w)
{
    w = min;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            gw[i][j] = x[sw][i][j];
        }
    }
}
/* ??? update global value b and w ??? */
```

再來會根據上一步所選出的歷史最佳解與歷史最差解，使用參數$delta$更新機率矩陣 Q(g)。首先我們會計算出 sb 和 sw 間的漢明距離，並和上一代的漢明距離做比較。此處將漢明距離簡稱為HD，如下方公式所示，若此代的距離較上代大，代表目前整體較不收斂，會對參數$delta$乘以一個比 1 大的數字μ；反之，若此代的距離較上代小，代表目前整體趨於收斂，會對$delta$乘以(2-μ)以減少參數$delta$的值。另外，當此代漢明距離與前一代相同時，則維持相同的$delta$值。

$$\begin{cases} delta = delta \times \mu & if\ HD_g > HD_{g-1} \\ delta = delta \times (2 - \mu) & if\ HD_g < HD_{g-1} \\ delta & if\ HD_g = HD_{g-1} \end{cases}$$

```
// hamming distance between sb and sw

int ham = 0;
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        if (x[sb][i][j] != x[sw][i][j])
        {
            ham++;
        }
    }
}

// update delta
last_ham == INT_MAX ? last_ham = ham : last_ham = last_ham;

// compare to last generation
if (ham > last_ham) // the difference become bigger
{
    adaptive_delta *= 1.001;
}
else if (ham < last_ham)
{
    adaptive_delta *= 0.999;
}
else
{
    adaptive_delta = adaptive_delta;
}
/* ??? update delta ??? */

last_ham = ham;
```

調整完$delta$後，再來會用它更新機率矩陣 Q(g)的值。我們會比較歷史最佳解 gb 與歷史最差解 gw 的每一顆量子位元${c^{gb}}_{ij}$與${c^{gw}}_{ij}$，由於我希望選中最佳解的機率可以越高越好，而選中最差解的機率可以越低越好，所以若${c^{gb}}_{ij} \neq {c^{gw}}_{ij}$，則增加 Q(g)中$Prob_{ij}$選中${c^{gb}}_{ij}$的機率並減少選中${c^{gw}}_{ij}$的機率。

```
/* update Q matrix */
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        if (gb[i][j] != gw[i][j]) // have to update
        {
            Q[i][j][gb[i][j]] += adaptive_delta;
            Q[i][j][gw[i][j]] -= adaptive_delta;
        }

        /* ??? repair ans ??? */
        if (Q[i][j][gw[i][j]] <= 0)
        {
            Q[i][j][gw[i][j]] = 0;

            /* Q[i][j][gb[i][j]] = 1 - remain */
            Q[i][j][gb[i][j]] = 1;
            for (int k = 0; k < n; k++)
            {
                if (k != gb[i][j])
                {
                    Q[i][j][gb[i][j]] -= Q[i][j][k];
                }
            }
        }
        /* ??? repair ans ??? */
```

註:更新機率矩陣時可能會將最差解的機率扣到變成負的，這會導致在第二步中的測量出現問題，所以在這邊如果扣完的機率小於 0，就會做修復機率的動作，將選中${c^{gw}}_{ij}$的機率改成 0，而為了確保相加完的機率等於 1，所以會同時將選中${c^{gb}}_{ij}$的機率改成 1-其他機率。

➢ *量子反閘*

概念: 由於我希望選中最佳解的機率可以越高越好，所以當目前 Q 矩陣裡選中${c^{gb}}_{ij}$的值不到 0.25 的時候，會對$Prob_{ij}$中的機率做量子反閘，將裡面最高的機率值和${c^{gb}}_{ij}$ 互換，同時將裡面的機率最小值和${c^{gw}}_{ij}$互換。

程式碼:

```
if (Q[i][j][gb[i][j]] < 0.25) // NOT
{
    /* find max */
    int maxIndex = 0, minIndex = 0;
    double max = Q[i][j][0], min = Q[i][j][0];
    for (int k = 1; k < 4; k++)
    {
        if (Q[i][j][k] > max)
        {
            max = Q[i][j][k];
            maxIndex = k;
        }
        else if (Q[i][j][k] < min)
        {
            min = Q[i][j][k];
            minIndex = k;
        }
    }

    /* swap the Prob. of gb and max */
    double tmp = Q[i][j][maxIndex];
    Q[i][j][maxIndex] = Q[i][j][gb[i][j]];
    Q[i][j][gb[i][j]] = tmp;

    /* swap the Prob. of lw and min */
    tmp = Q[i][j][minIndex];
    Q[i][j][minIndex] = Q[i][j][gw[i][j]];
    Q[i][j][gw[i][j]] = tmp;
}
```

土黃框: 找目前最大和最小的機率

綠框: 做交換機率

劃出電路圖及真值表

```cpp
void draw_circuit()
{
    cout << "\n==== Circuit Diagram ====\n(○: low-level control bit)\n(●: high-level control bit)\n(⊕: not-gate)\n";
    getTheLines();
    for (int i = 0; i < n; i++)
    {
        cout << "q" << i;
        for (int j = 0; j < bestAns; j++)
        {
            switch (mingb[i][j])
            {
            case 0:
                cout << "--○-";
                break;
            case 1:
                cout << "--●-";
                break;
            case 3:
                cout << "--⊕-";
                break;
            case 2:
                cout << "-----";
                break;
            }
            if (j == bestAns - 1 && i != n - 1)
            {
                cout << "\n";
                for (int k = 0; k < bestAns; k++)
```

判斷是矩陣裡面去決定印出來的 gate

```cpp
            if (j == bestAns - 1 && i != n - 1)
            {
                cout << "\n";
                for (int k = 0; k < bestAns; k++)
                {
                    if (k == 0)
                    {
                        if (line[i][k])
                            cout << "    │ ";
                        else
                            cout << "      ";
                    }
                    else if (line[i][k])
                    {
                        cout << "  │ ";
                    }
                    else
                    {
                        cout << "     ";
                    }
                }
                cout << "\n";
            }
        }
    }
    cout << endl;
```

這邊的條件判斷是印出中間的連接線

```cpp
void getTheLines()
{
    // draw or not
    for (int i = 0; i < bestAns; i++) // through m gates
    {
        bool lineStart = false;
        int start = 0, end = 0;
        for (int j = 0; j < n; j++) // through n bits
        {
            if (!lineStart)
            {
                if (mingb[j][i] == 2)
                {
                    start++;
                }
                else
                {
                    end = start;
                    lineStart = true;
                }
            }
            else // find end
            {
                if (mingb[j][i] != 2)
                {
                    end = j - 1;
                }
            }
        }
        for (int j = start; j <= end; j++)
        {
            line[j][i] = true;
        }
    }
}
```

這裡判斷 wire gate 時就不印出中間的連接線。

```cpp
void TruthTable()
{
    int OutputLength = output.size();
    int tmp = OutputLength;
    int BitsNumber = 0;
    int tmpinput[nMAX][mMAX] = {0};
    while (tmp != 1)
    {
        BitsNumber += 1;
        tmp /= 2;
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < bestAns; ++j)
        {
            tmpinput[j][i] = mingb[i][j];
        }
    cout << endl
         << endl
         << "====== Truth Table ======\n";

    for (int i = 0; i < pow(2, BitsNumber); i++)
    { // pow(2,qubit)
        string request = toBinaryString(i, BitsNumber);
        string output;
        int flag = 1;
        int notbit = 0;
        for (int j = 0; j < BitsNumber; j++)
        {
            cout << request[BitsNumber - j - 1] << " ";
        }
        cout << "   ->   ";
        for (int j = 0; j < bestAns; j++)
        { // 0 for 0 control,1 for 1 control, 2 for wire, 3 for not
            flag = 1;
            for (int k = 0; k < BitsNumber; k++)
            {
                if (tmpinput[j][k] == 1 && request[BitsNumber - k - 1] != '1')
```

列印出該電路的真值表

程式使用說明

- 執行程式後，會顯示提示讓使用者輸入目標的可逆函數

```
選取 C:\Users\user\Desktop\量子作業\期末報告\Quantu
Input the target circuit: 7 0 1 3 4 2 6 5
Your Input:7 0 1 3 4 2 6 5
```

- 再來就會開始執行 5 次實驗，而每次做完實驗，都會印出此次找到的最少邏輯閘數目與適應值

```
====== experiment1 ======
number of gate = 4
best fitness = 1.3

====== experiment2 ======
number of gate = 6
best fitness = 1.25

====== experiment3 ======
number of gate = 4
best fitness = 1.3

====== experiment4 ======
number of gate = 4
best fitness = 1.3

====== experiment5 ======
number of gate = 4
best fitness = 1.3
```

- 在程式的最後會印出 5 次實驗裡所找到的最佳電路 (修改過電路圖)

```
==== Circuit Diagram ====
(○: low-level control bit)
(●: high-level control bit)
(⊕: not-gate)
q0--○---⊕--------○-
    |   |        |
q1------○---⊕----●-
    |   |   |    |
q2--⊕---●---●----⊕-
```

- 並且印出電路的真值表

```
====== Truth Table ======
0 0 0   ->    1 1 1
0 0 1   ->    0 0 0
0 1 0   ->    0 0 1
0 1 1   ->    0 1 1
1 0 0   ->    1 0 0
1 0 1   ->    0 1 0
1 1 0   ->    1 1 0
1 1 1   ->    1 0 1
```