Pre-processing

1. 程式架構

- 三種struct型別

  1. 用來記錄邊Edge資訊

```
struct Edge {
 string 邊的名稱 ;
 int 權重 ;
 string 來自哪個節點 ;
 string 到哪個節點 ;
} ; // Edge
```

  2. 用來記錄節點Vertex資訊

```
struct Vertex {
 string 名字 ;
 string 此節點型別 ;
 string 此節點顏色 ; // 用於DFS
 int 與起始點距離 ; // 用於DFS
 string 父節點 ;
 int 結束時間 ;
 vector<Edge> 相鄰邊 ;
 vector<Vertex> 子節點 ;
} ; // Vertex
```

  3. 用來記錄整張圖Graph的資訊

```
struct G {
 vector<Vertex> V ; // 紀錄所有節點
 vector<Edge> E ; // 記錄所有邊
};
```

- 兩個class

  1. 用來處理兩個圖走訪演算法class Graph

```
class Graph {
 private:
     G g ; // 紀錄用來處理DFS的圖資訊
     G g_dijkstra ; // 紀錄用來處理Single source shortest path的圖資訊
     vector<Vertex> dijkstraResult ; // 紀錄Dijkstra的結果
     int time ; // 用於DFS走訪記錄當前時間

     void DFS_visit( string nextVectexName ) ; // DFS走訪下個節點

 public:
     void init() ; // 從讀檔結果初始化圖資料
     void clear() ; // 釋放記憶體
     void showVertex( vector<Vertex> vList ) ; // 印出vList所有節點資料
     void showEdge( vector<Edge> eList ) ; // 印出eList所有邊的資料
     Edge findEdge( string name ) ; 透過邊的name回傳此邊的資訊
     int findVertexIndex( string name ) ; // 透過節點的name找到此節點在vertex list的index
     void buildAdjList() ; // 建立所有節點的相鄰串列
     void DFS() ; 進行DFS走訪
     void printDFSResult() ; // 印出DFS結果

     int getEdgeWeight( string from, string to ) ; // 透過起始點即結束點的名稱獲得此兩點間的邊權重
     void initialSingleSource() ; // Dijkstra演算法的初始化
     void relax( string uName, string vName, vector<Vertex> &queue ) ; // Dijkstra演算法中根據相鄰
```

```
邊權重更新節點資訊
    void updateQueue( vector<Vertex> &queue, Vertex v ) ; // Dijkstra演算法更新目前Queue的值
    void Dijkstra() ; // 進行Dijkstra演算法
    void printDijkstraResult() ; // 印出Dijkstra演算法結果
} ;
```

2. 用來處理讀檔存檔的class Tool

```
class Tool {
 private:
    fstream file ; // 處理檔案
    string circuitName ; // 此電路名稱
    vector<Vertex> vList ; // 節點資訊
    vector<Edge> eList ; // 邊資訊

    bool pureStr( string str ) ; // 判斷此字串是否為純數字(不包含特殊符號)
    void setEdgeDest( string edgeName, string vertexName ) ; // 設邊的結束點
    void setEdgeSource( Edge edge ) ; // 設邊的起始點
    string Instance() ; // 在讀到"INSTANCE"後處理點和邊的資訊, 邊讀邊存, 並回傳下一個讀到的字串
    bool Circuit() ; // 在讀到"CIRCUIT"後處理後續資料
    bool checkEdges() ; // 確認每個邊是否都有起始點和結束點

 public:
    void clear() ; // 釋放記憶體
    bool isSpace( char ch ) ; // 判斷ch是否為空白
    bool isDel( char ch ) ; // 判斷ch是否為特殊符號
    char skipSpace() ; // 跳過空白,並回傳下一個不識空白的字元
    void openFile() ; // 開檔
    void closeFile() ; // 關檔
    void readFile() ; // 讀檔
    string GetToken() ; // 切token得到下一個字串
} ; // Tool
```

2. 運作流程

- Step1. 使用者輸入測試檔名稱(input.txt)
- Step2. 開檔並讀檔
- Step3. 以字串為單位讀取檔案資料, 並檢查是否有資料遺漏
- Step4. 若測試檔資料無誤則建立相鄰串列
- Step5. 執行DFS走訪
- Step6. 執行Dijkstra演算法
- Step7. 輸出兩個結果

3. 執行方式

```
[vlsi17@dhcpb202 Louis]$ g++ finalProject.cpp -o finalProject
[vlsi17@dhcpb202 Louis]$ ./finalProject
Please enter a number (0)Quit (1)Continue: 1
Please enter the testing input file name (ex: input.txt): input.txt

Successfully open < input.txt > !

*************  Result - start *************

[DFS traverse order]:

S -> V1 -> V3 -> V7 -> V9 -> D -> V10 -> V5 -> V4 -> V2 -> V6 -> V8

-----------------------------------------

[Distance from Source to each vertex]:

Source - S   : 0
Source - V1  : 3
Source - V2  : 2
Source - V3  : 4
Source - V4  : 3
Source - V5  : 6
Source - V6  : 4
Source - V7  : 5
Source - V8  : 8
Source - V9  : 8
Source - V10: 6
Source - D   : 9

**************  Result - end  **************

Please enter a number (0)Quit (1)Continue: 0

Quit the program!
[vlsi17@dhcpb202 Louis]$
```

Shortest Path

1. 程式說明

- **資料結構**
  - 三種struct型別
    1. 用來記錄邊Edge資訊

```
struct Edge {
 string 邊的名稱 ;
 int 權重 ;
 string 來自哪個節點 ;
 string 到哪個節點 ;
} ; // Edge
```

    2. 用來記錄節點Vertex資訊

```
struct Vertex {
 string 名字 ;
 string 此節點型別 ;
 string 此節點顏色 ; // 用於DFS
```

```
    int 與起始點距離 ; // 用於DFS
    string 父節點 ;
    int 結束時間 ;
    vector<Edge> 相鄰邊 ;
    vector<Vertex> 子節點 ;
  } ; // Vertex
```

3. 用來記錄整張圖Graph的資訊

```
  struct G {
    vector<Vertex> V ; // 紀錄所有節點
    vector<Edge> E ; // 記錄所有邊
  };
```

- **作法說明 Pseudo code:**

```
DFS(G)
    for 所有包含在G.V的節點 ( 以u代表 )
        u.color = "WHITE" ; // 初始化為白色(尚未走訪)
        u.parent = NIL ; // 父節點未定

    time = 0 ; // 初始化走訪時間

    for 所有包含在G.V的節點 ( 以u代表 )
        if u.color == "WHITE" // 尚未走訪
            DFS-Visit(u) ; // 呼叫DFS-Visit()繼續往下走
```

```
DFS-Visit(u)
    time += 1 ; // 時間往下走
    u.d = time ; // 設定走到u點當下的時間為time
    u.color = "GRAY" ; // 設定u節點顏色狀態為灰色(第一次走訪)
    for v為所有u的相鄰串列節點(走訪所有u的鄰居)
        if v.color == "WHITE" // 尚未走訪
            v.parent = u ; // 設定v的父節點為u
            DFS-Visit( v ) ; // 繼續以v為起始往下走

    u.color = "BLACK" ; // 當走完u的所有相鄰節點, 將u設為黑色(已走完)
    time += 1 ; // 時間繼續往下走
    u.finistTime = time ; // 設定u的結束走訪時間
```

2. 執行結果

輸出結果如下: 走訪節點的順序為從左至右

```
[DFS traverse order]:

S -> V1 -> V3 -> V7 -> V9 -> D -> V10 -> V5 -> V4 -> V2 -> V6 -> V8
```

3. 執行方式

- 執行方式首先 `g++ finalProject.cpp -o finalProject` 編譯 finalProject.cpp
- 再 `./finalProject` (Enter)執行此檔
- 輸入數字 1 以繼續執行
- 再輸入測試檔的名稱input.txt (Enter)

- 確認後直接跑出DFS執行結果

```
[vlsi17@dhcpb202 Louis]$ ./finalProject
Please enter a number (0)Quit (1)Continue: 1
Please enter the testing input file name (ex: input.txt): input.txt

Successfully open < input.txt > !

*************  Result - start *************

[DFS traverse order]:

S -> V1 -> V3 -> V7 -> V9 -> D -> V10 -> V5 -> V4 -> V2 -> V6 -> V8

-----------------------------------------
```

Dijkstra Algorithm

1. 程式說明

- **資料結構**
  - 三種struct型別
    1. 用來記錄邊Edge資訊

```
struct Edge {
 string 邊的名稱 ;
 int 權重 ;
 string 來自哪個節點 ;
 string 到哪個節點 ;
} ; // Edge
```

    2. 用來記錄節點Vertex資訊

```
struct Vertex {
 string 名字 ;
 string 此節點型別 ;
 string 此節點顏色 ; // 用於DFS
 int 與起始點距離 ; // 用於DFS
 string 父節點 ;
 int 結束時間 ;
 vector<Edge> 相鄰邊 ;
 vector<Vertex> 子節點 ;
} ; // Vertex
```

    3. 用來記錄整張圖Graph的資訊 ``` struct G { vector V ; // 紀錄所有節點 vector E ; // 記錄所有邊 };

- **作法說明 Pseudo code:**

```
INITIALIZE-SINGLE-SOURCE():
    for 所有包含在G.V的節點 ( 以u代表 )
        u.d = 無限大 ; // 初始化每個節點距離source無限遠
        u.parent = NIL ; // 父節點未定

    s.d = 0 ; // source本身與source無距離
```

```
//  若目前v與source的距離大於父節點u加上u, v間邊的權重, 則更新v與source的距離
RELAX(u, v, w): // u 為 v 的父節點
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.parent = u
```

```
DIJKSTRA( w, s):
    INITIALIZE-SINGLE-SOURCE()
    S = NULL // 用來放已走的節點, 走過就放進來
    Q = G.V // 已 min-priority queue的結構存節點資料 (在此以vector實作)
    while Q is not empty:
        u = Q 中最小的節點
        put u into S // 將u視為已走訪
        for 所有 u 的相鄰串列節點 (以 V 代表)
            RELAX( u, v, w ) // 檢查並更新距離
```

2. 執行結果

從上至下為, 依照讀入之節點順序輸出每個節點與Source的距離

```
[Distance from Source to each vertex]:

Source - S   : 0
Source - V1  : 3
Source - V2  : 2
Source - V3  : 4
Source - V4  : 3
Source - V5  : 6
Source - V6  : 4
Source - V7  : 5
Source - V8  : 8
Source - V9  : 8
Source - V10: 6
Source - D   : 9
```

3. 執行方式

- 執行方式首先 `g++ finalProject.cpp -o finalProject` 編譯 finalProject.cpp
- 輸入 `./finalProject` (Enter)執行此檔
- 輸入數字 1 以繼續執行
- 再輸入測試檔的名稱input.txt (Enter)
- 直接跑出DFS和Dijkstra演算法的執行結果

```
[vlsi17@dhcpb202 Louis]$ g++ finalProject.cpp -o finalProject
[vlsi17@dhcpb202 Louis]$ ./finalProject
Please enter a number (0)Quit (1)Continue: 1
Please enter the testing input file name (ex: input.txt): input.txt

Successfully open < input.txt > !

*************  Result - start *************

[DFS traverse order]:

S -> V1 -> V3 -> V7 -> V9 -> D -> V10 -> V5 -> V4 -> V2 -> V6 -> V8

----------------------------------------

[Distance from Source to each vertex]:

Source - S   : 0
Source - V1  : 3
Source - V2  : 2
Source - V3  : 4
Source - V4  : 3
Source - V5  : 6
Source - V6  : 4
Source - V7  : 5
Source - V8  : 8
Source - V9  : 8
Source - V10: 6
Source - D   : 9

**************  Result - end  **************

Please enter a number (0)Quit (1)Continue: 0

Quit the program!
[vlsi17@dhcpb202 Louis]$
```