Find a minimum cost spanning tree

with Sollin algorithm Readme file

陳昭成

\*Programing language:

使用C++ 撰寫

\*Compilation:

使用Visual Studio 2013內建之compiler

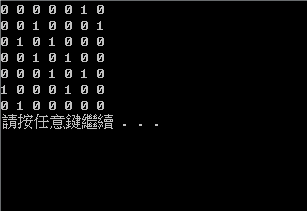
\*Execution:

直接點選Visual Studio 2013 的”開始偵錯”

\*Output:

依下方題目範例所給的7 X 7 Graph 的adjacency matrix

所做出的minimum cost spanning tree

-1 28 -1 -1 -1 10 -1

28 -1 16 -1 -1 -1 14

-1 16 -1 12 -1 -1 -1

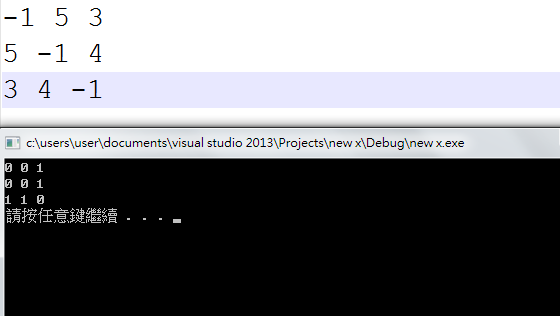
-1 -1 12 -1 22 -1 18

-1 -1 -1 22 -1 25 24

10 -1 -1 -1 25 -1 -1

-1 14 -1 18 24 -1 -1

將Graph的vertex數目改為3後，依下方題目範例所給的3 X 3 Graph 的adjacency matrix，所做出的minimum cost spanning tree



Sollin 演算法時間複雜度:

在此演算法中，對於一擁有V個vertex及E個edge的Graph，則一開始即擁有V棵tree，而每棵tree在每一次的重複步驟中（every stage），都會找其所參與的所有edge中cost最小且該edge的兩個vertex分別屬於不同的tree(或是形成一個迴路)，將此edge納入minimum spanning tree中的一個edge, 亦即將此二tree連接成同一個tree。

在每次的重複步驟中，必須掃瞄過每一顆tree的所有edges，從中找cost最小且連接後不形成迴路的edge，來和其他的tree做連接，並合併此二顆tree的adjacency edge list，並把二顆tree所屬的vertex歸於同一顆tree，接著再繼續往下一顆在本次重複步驟中尚未進行minimum cost edge 選取的樹，重複做一次上述的選取方法，直到目前所有的tree都被處理過為止，此為一次的重複步驟。此重複步驟將會做log V次（請看下述）。

而在最差的情況下，每次的重複步驟，Graph中所有的edge都會被掃瞄過一次，因此，在Graph總共有E個edge的情況下，每個重複步驟所需的時間複雜度為O(E) 。

每做一次選取minimum cost edge的步驟就會將兩棵樹進行連接而合併，而每做完一次的重複步驟，就會使一半的樹因被合併而消失，因此，在初始有V棵樹的情形下，此重複步驟會進行log V次，總共時間複雜度便為O(log V)。

因此，對於一擁有V個vertex及E個edge的Graph而言整體重複步驟會進行log V次，每次需時O(E)，因此Sollin演算法所需得時間複雜度為O(E log V) 。

\*Function Description:

根據題目範例所給的7 X 7 Graph 的adjacency matrix，

因此，我在寫程式時，便從檔案讀入一開始範例的Graph的adjacency matrix，再以此matrix建立7棵tree個別的adjacency edge list，以進行接下來的minimum cost edge的選擇。

此為edge node的宣告:

typedef struct edge \*edgePointer;

typedef struct edge

{

edgePointer right;//指向下一個edge

int cost;//存放該edge的cost值

int u;//start vertx u.

int v;//destination vertex v.

};

此為vertex的宣告:

typedef struct vertex \*vertexPointer;

typedef struct vertex

{

edgePointer right;//指向該vertex的第一個edge node 其他的edge node就連接在此第一個edge之後 串接成該vertex 的edge list

vertexPointer down;//指向下一個vertex node

int value;//存放該vertex的數值

};

Ｆunction:

void read\_input\_from\_file(int read\_in[][vertexNUM])

輸入:在主程式生成的二維陣列

處理:

讀取存在文字檔"adjacency matrix.txt"內的edge cost matrix並存入叫”read\_in”的二維陣列

Ｆunction:

edgePointer attach\_first\_edge(vertexPointer tail, int cost, int i, int j)

處理:

生成接在vertex後面的第一個edge node，並以輸入的cost 及所處再哪兩個vertex間的資訊來初始化

Ｆunction:

edgePointer attach\_other\_edge(edgePointer tail, int cost, int i, int j)

處理:

生成接在vertex後面的除了第一個edge node以外的其他edge node，並以輸入的cost 及所處再哪兩個vertex間的資訊來初始化

Ｆunction:

vertexPointer attach\_vertex(vertexPointer tail, int value\_of\_vertex)

處理:

生成主程式中生成的第一個vertex後面所連接的其他vertex，並以輸入的vertex 的數值來初始化

Ｆunction:

void link\_the\_edges(vertexPointer head, const int read\_in[][vertexNUM])

輸入：指向在主程式中生成的第一個vertex,

存放edge cost matrix的二維陣列”read\_in”

用途：

將edge掛上所屬的vertex，各vertex間也以指標連接，

形成adjacency list

Ｆunction:

void Sollin (vertexPointer v, int joined\_party[], bool min\_cost\_spanning\_tree[vertexNUM][vertexNUM])

用途:

在每次的重複步驟中，必須掃瞄過欲進行連接的樹的所有edges，從中找cost最小且連接後不形成迴路的edge，來和其他的tree做連接，被連接的樹則以改變adjacency edge list中的指標來將兩棵樹進行合併，接著再繼續往下一顆在本次重複步驟中尚未進行minimum cost edge 選取的樹，再做一次上述的選取方法。

處理：

vertex

while(尚未走完所有vertex)

｛

對於每個tree，用指標掃瞄過其所有的edge，並呼叫

＂whether\_different\_group(forward, joined\_party);＂

以確認該edge兩端的vertex是否屬於不同的樹，若是，則將edge 存為min\_cost\_edge，去與下一個同樣兩端的vertex也屬於不同的樹的edge的cost進行比較。若兩端的vertex已屬於同一棵樹時，則不做任何事，跳過此edge，因為若接上此edge將形成迴路或重複接上已經被接上過的edge。

持續以上動作直到掃描完所有該vertex的edge為止，以找出該vertex最小cost且其兩端屬於不同棵樹的edge。

當已找出該vertex的min\_cost\_edge後，

呼叫函式“add\_min\_cost\_tree\_edge(v, min\_cost\_edge, joined\_party, min\_cost\_spanning\_tree);”

將此edge加入到minimum cost spanning tree，並將該edge兩端的vertex注記為屬於同一棵樹，並以函式“add\_min\_cost\_tree\_edge＂中的

函式＂change\_the\_link＂進行合併左右兩個vertex所屬的樹。

往下一的vertex走去

｝

Ｆunction:

vertexPointer add\_min\_cost\_tree\_edge(vertexPointer v, edgePointer min\_cost\_edge, int joined\_party[], bool min\_cost\_spanning\_tree[vertexNUM][vertexNUM])

讀取該edge兩端的vertex數值，以在存放minimum\_cost\_spanning\_tree結果的二維陣列＂min\_cost\_spanning\_tree＂中的對應欄位填上”1”表示此處有edge存在，並將該edge兩端的vertex注記為屬於同一棵樹，

再呼叫函式＂change\_the\_link＂進行合併左右兩個vertex所屬的樹。

Ｆunction:

bool whether\_different\_group(edgePointer forward, int joined\_party[])

輸入：cost 不為”-1”　的 edge（即Graph中真實存的edge）

,記錄各vertex所屬的tree的陣列

輸出：是否兩端vertex屬於不同tree的edge

處理：

比較該edge兩端的vertex的所屬tree陣列中的所屬tree數值是否不同，

回傳是或否

Ｆunction:

edgePointer get\_min\_cost\_edge(edgePointer forward, edgePointer min\_cost\_edge)

輸入：兩端vertex屬於不同tree的edge

輸出：現在所處的vertex中cost最小的edge

處理：

　　找出現在所處的vertex中cost最小的edge

Ｆunction:

print out the minimum cost spanning tree made with Sollin

void print\_out\_min\_cost\_spanning\_tree(bool　min\_cost\_spanning\_tree[][vertexNUM])

輸入：記錄成果：minimum cost spanning tree的二維陣列

輸出：

印出最後的成果：minimum cost spanning tree　made with Sollin algorithm

主程式：

｛

read in the adjacency matrix from file

read in the input from the file and store in the 2D array "read\_in".

//create the first vertex list of the adjacency list

vertexPointer v, tail;

v = (vertexPointer)malloc(sizeof(vertex));

v->right = NULL;

v->down = NULL;

v->value = 0;

tail = attach\_vertex(v, 1);

for (int i = 2; i < vertexNUM; i++)

{

tail = attach\_vertex(tail, i); //attach other head node to the back of the head node list.

}

/\*create the the edge list of adjacency list\*/

link\_the\_edges(v, read\_in);

int joined\_party[vertexNUM];

//to store the party which each vertex joins in and to avoid the multiple copies of edges and cycle's generation.

//initialize the all the vertex's joined party as the vertex itself.

int joined\_party[vertexNUM];

//to store the party which each vertex joins in and to avoid the multiple copies of edges and cycle's generation.

for (int i = 0; i < vertexNUM; i++)

{

joined\_party[i] = i;

}

//initialize the all the vertex's joined party as the vertex itself.

/\*show the minimum cost spanning tree\*/

//initialize the output minimum cost spanning tree to 0 in all field.

bool min\_cost\_spanning\_tree[vertexNUM][vertexNUM];

for (int i = 0; i < vertexNUM; i++)

{

for (int j = 0; j < vertexNUM; j++)

{

min\_cost\_spanning\_tree[i][j] = 0;

}

}

while (spanning tree 的edge總數　< spanning tree 的vertex總數- 1)

{

Sollin(v, joined\_party, min\_cost\_spanning\_tree);

}

print\_out\_min\_cost\_spanning\_tree(min\_cost\_spanning\_tree);

｝