Ch6-Graph

定義:

G=(V,E) => 代表Graph是由兩個非空集合V、E組成

其中: V => 代表頂點(Vertex)集合

E => 代表邊(Edge)集合(假定圖中無重複邊)

種類:

1. 無向圖(Undirected Graph)

邊不具備方向性，即(I,j)=(j,i)代表同一邊

2. 有向圖(Directed Graph)

邊具備方向性，即<i,j> != <j,i>

相關術語:

1. Eulerian Cycle: 從任何一點開始，經過每個邊一次，在回到原出發點，稱之

=> 每個頂點的Degree必須為偶數

2. Eulerian Chain: 從任何一個頂點開始，經過每個邊一次，但不一定要回到原出發點，稱之

=> 有2個頂點的Degree必須為奇數，其他均為偶數

3. Complete Graph: 圖具有最多邊數，稱之

無向圖 => n個頂點，最多邊數 = n(n-1)/2

有向圖 => n個頂點，最多邊數 = n(n-1)

4. 子圖(subgraph or Component)

5. Path(路徑): 只一個由頂點組成的序列

6. Path Length(路徑的長度): 路徑上所包含的邊之數目

7. Simple Path: 除了起點與終點可能相同，其餘經過的頂點皆不相同

8. Cycle: 起點跟終點相同的Simple Path

9. Connected(連通的): 針對無向圖而言

=> 無向圖中，每個頂點對之間皆有Path存在

10. Connected Component(連通的子圖)

11. Strongly Connected: 針對有向圖而言

=> 在有向圖中，任何頂點對(i,j)之間，i有path可以到j，j有path可以到i

12. Strongly Connected Component

13. Degree(分支度)

=> 無向圖: 頂點的Degree是頂點連接邊數

=> 有向圖: 頂點的Degree分為 Out-Degree: 頂點射出之邊數；In-Degree: 射入頂點之邊數

14. G=(V,E)，|V|= n，|E|= e

=> 無向圖: e = 1/2 \*

=> 有向圖 e = (in-Degree)

無向圖中，所有的Degree加總必定為偶數 => True

圖形的表示方法

Adjacency Matrix (相鄰矩陣)

定義:

G=(V,E)，|V|= n，則準備一個n\*n的Matrix，叫A

則A[i,j] => 1 if (i,j)邊存在

* 0 otherwise

Space需求: O(n平方)

無向圖:

無向圖之相鄰矩陣，必為對稱矩陣(Symmetric Matrix)

即A[i,j] = A[j,i]

基本操作:

1. 判斷邊(i,j)是否存在:

If (i,j) == 1 then 存在

else 不存在

Time: O(1)

2. 求i之Degree

求第i列元素之加總 or 第i行之加總

Time: O(n)

3. 求G之邊數

套公式

Time: O(n平方)

有向圖:

有向圖之相鄰矩陣不一定是對稱矩陣

基本操作:

1. 判斷邊(i,j)是否存在

同無向圖

2. 求i之Out-Degree and In-Degree

Out-Degree: 求第i列之元素加總

In-Degree: 求第i行之元素加總

3. 求G之邊數

套公式

Time: O(n平方)

Adjacency Lists (相鄰串列)

定義:

G=(V,E)，|V|= n，|E| = e，則準備一個Vertex[1…n] of Pointer，其中Vertex[i]代表頂點i之相鄰串列紀錄與i相鄰之頂點編號

=> Node Structure: Vertex No、Next

無向圖:

無向圖中所有的串列之Node總數 = 2\*e

空間需求 = O(n+e)

因為無向圖(i,j) = (j,i) 所以i會加到j串列，而j也會加到i串列，

故一個邊生出兩個Node

基本操作:

1. 判斷(I,j)是否存在

去Vertex[i]找是否有Node j存在

Time: O(Vi串列長度) <= Node總數 => O(e)

2. 求Vi之Degree

求Vertex[i]串列長度，Time => O(e)

3. 求圖形邊數

加總每條串列之長度/2

因為有n條串列，全部串列之Node總和 = 2\*e

故Time: O(n+e)

有向圖

串列之Node總數 = e

基本操作:

1. 判斷邊(I,j)是否存在

同無向圖

2. 求Vi之Out-Degree、In-Degree

Out-Degree: Vertex[i]之串列長度=Out-Degree => O(e)

In-Degree: 統計所有串列之Nodei出現的次數 => O(n+e)

3. 求圖形邊數

加總每條之長度 => Time: O(n+e)

|  |  |  |
| --- | --- | --- |
| 比較表 | Adjacency Matrix | Adjacency Lists |
| 若G中頂點數很多，但邊數很少 | 不適合，因為空間為O(n平方)，會形成Sparse Matrix(稀疏矩陣) | 適合  因為空間為O(n+e) |
| 邊數很多 | 適合 | 不適合，因為滿邊，邊的數量會變成n平方  =>O(n+e)=O(n+O(n平方) |
| 判斷邊是否存在 | 適合  O(1) | 不適合  Time<=O(e) |
| 求圖中邊數  判斷連通與否  DFS/BFS  判斷有無Cycle | 不適合  O(n平方) | 適合  因為e通常不會經常是滿邊的狀態  所以Time: O(n+e) |

Adjacency MultiLists

圖中每一個邊皆用一個Node表示，而Node Structure如下

Vi、Vj、Link for Vi、Link for Vj

Link for Vi(Vj) => 指標指向下一個包含Vi(Vj)之Node

此外，有一個Vertex[1…n] of Pointer，其中Vertex[i]指向第一個包含Vi之Node

Incidence Matrix [Algo版]

定義:

G=(V,E), |V| = n, |E| = e，則準備一個n\*e Matrix，令為A，假設ek = (i,j)，則A[i,k]=A[j,k]=1，其他為0

Graph Traversal(圖形追蹤)

拜訪圖中每個頂點一次

分為兩種方法:

DFS(Depth First Search):深度/縱向

BFS(Breadth First Search):廣度/橫向

|  |  |  |  |
| --- | --- | --- | --- |
| 比較表 | | DFS | BFS |
| 輔助之Data Structure | | Stack | Queue |
| 時間 | 相鄰矩陣 | O(n平方) | O(n平方) |
| 相鄰串列 | O(e)(DS版本) | O(e)(DS版本) |
| O(n+e)(Alg版本) | O(n+e)(Alg版本) |

DFS(DS版)(無向圖)

Note:

1.注意Start Vertex為何

2.DFS Order並非唯一，通常會規定依Vertex No小者優先，則DFS Order唯一

BFS (DS版)(無向圖)

Note: BFS Order也非唯一

BFS (Algo版)(無向圖)

說明:

1. u.color =>

white:尚未Visited

Gray:已Visited，但尚未完成BFS之檢視

black:已完成BFS之檢視

2. u.d =>

Start 到 u 之Distance(邊數)

3. u.Ω =>

u 之父點，即從哪個點來拜訪u

Note:

1.針對無向圖，求S到其他各點之Short Path length，則用BFS(G,S)即可求出

2.Ω的目的，方便繪圖

DFS (Algo版)(有向圖)

說明:

1. u.d = u的discovered time

2. u.f = u的finished time

DFS對有向圖

邊的種類可分為4類

1. Tree Edge(樹邊)

2. Back Edge(後退邊)

3. Forward Edge(前進邊)

4. Cross edge(跨越邊)

Tree Edge: DFS追蹤經過的邊

Rule: u -> v if v 是 白色(white)

Then (u,v) is Tree Edge

Back Edge: 指回祖先(Ancestor)邊

Rule: u -> v if v 是灰色(Gray)

Then (u,v) is Back Edge

Forward Edge: 指向子孫(後代)(descendent)邊

Cross Edge: 跨越不同子樹/樹之Edge或除了上述三種以外的Edge

如何區別Forward Edge、Cross Edge

利用Discovered Time

u -> v if v 黑色(Black)

then if u.d < v.d

then Forward Edge

else

Cross Edge

對無向圖而言，只有Tree Edge(經過) and Back Edge(未經過)

Graph Traversal 應用

1. 判斷有向圖是否有Cycle

Ans: 原則:只要有Back Edge，則必有Cycle，反之沒有，Time: O(n+e) if 相鄰串列

2. 判斷無向有是否有Cycle

Ans: 原則:只要有Back Edge，則必有Cycle，反之沒有。但Back Edge判定與有向圖不同

For u-> v

=> if (v.color == Gray)

If (u.Ω != v)

Then (u,v) is Back Edge

3. 判斷無向圖是否Connected?

Ans: Assume用DS版本的DFS/BFS algo

Steps:

1. 用BFS/DFS 追蹤無向圖G

2. 檢查Visited[1…n]是否皆為True，若是，則為Connected，否則Return Unconnected

Time:

陣列 串列

Step1 O(n平方) O(n+e)

Step2 O(n) O(n)

Total O(n平方) O(n+e)

Spanning Tree(展開樹)

定義:

給定一個Connected 無向圖 G = (V,E)，令S=(V,T)為G的其中一個Spanning Tree，則S滿足:

1. E = T + B，T為Tree Edge、B為Back Edge

2. 自B中任取一邊，加入S中，必形成Cycle

3. 在S中，任何頂點對之間存在unique simple path

Note:

1. 若G is Unconnected，則必無Spanning Tree

2. Spanning Tree >= 1棵，If G is connected

3. 若G = (V,E) , |V| = n ，則Spanning Tree 邊數 = n -1

Min Spanning Tree(最小成本展開樹)(MST)

定義:

令G = (V,E)為Connected 無向圖，且邊上附有成本(cost or weight)

則在G的所有Spanning Tree中，具有最小的邊成本和者，稱之

Note:

1. MST可能>=1棵，因為G中可能有多個邊具有相同成本值

2.若G中每一邊成本皆不同時，則MST為unique

應用:

1. 電路布局成本最小化

2. 連結n個城市之最少交通建設成本(or 票價)

Model: 欲把n個點連結起來，最少要n-1條邊，選擇哪些n-1條邊成本和是最低的

Alg求Min Spanning Tree

1. Kruskal’s Algo

2. Prim’s Algo

3. Sollin’s Algo

此三個Algo皆採用Greedy策略

Kruskal's algo:

令G=(V,E)，|E|=e，|V|=n

Steps:

1. 自E中排出最小成本的邊(u,v)

2. 判斷(u,v)加入Spanning Tree中，是否會形成Cycle，若會，則放棄(u,v)，否則加入S中

3. Repeat 1~2 Until 挑出(n-1)條邊或E為空

4. if S has < (n-1)條邊 then “No Spanning Tree”

如何判斷(u,v)加入S中是否會形成Cycle:

作法 => 利用Disjoint Sets之Find(X)及Union(i,j)運作

Find(X) => Find-with-path-compression:O(1)

Union(i,j) => Union-by-Height:O(1)

時間分析: 令有V個點，E條邊in G

[DS版]:

Kruskal’s Algo最多作E回合，而每一回合主要有兩個動作:

1. 自邊集合中Delete-min cost edge(u,v)

=> 令G中所有邊成本用Heap來維持，所以Delete-min cost花O(logE)

2. 判斷(u,v)加入S中是否形成Cycle

=> 用union(i,j)及Find(u)/(v)之Algo，皆只花O(1)time

所以每一回合頂多花O(logE)

Total Time: E\*O(logE) = O(ElogE)

[Algo版]:

1.先將每邊成本由小->大排序

Time: O(ElogE)

2.make Sets for each Vertex

Time: O(V)

3.挑最小成本邊(u,v) => O(1)

判斷(u,v)加入S是否會形成cycle => O(1)

此動作做E回合

所以 time: E\*O(1) =O(E)

Total Time: O(ElogE)

又已知E < V平方，因為E最多是V(V-1)/2

所以logE < logV平方 = 2logV

O(ElogE) => O(ElogV)

Prim's Algo:

[DS版]

令G=(V,E)是Connected無向圖，V={1,2,3...n}，U={1} // 視為Start Vertex，could be any Vertex

Steps:

1.挑出最小成本的邊(u,v)，且u ε U，v ε V-U

2.加入(u,v)到S中，且將v點至V-U中移除，加入U中

3.Repeat 1~2 Until U=V or E為空

4.if S has < (n-1)條邊 then "No Spanning Tree"

Time:O(n平方) or O(V平方)，因為Based on Adjacency Matrix

[Algo版]

Time分析:

[法一]:priority Queue用Binary Heap製作(Min-Heap)

Time = O(V)(初值) + O(V)(建Heap) + O(logV)\*V次(Extract-Min(Q)) + O(logV)\*2E(decrease key動作)

一般而言，E會大於V (因為E>=(V-1))

所以Prim's Time: O(ElogV) => 與Kruskal's algo相同

[法二]:priority Queue用Fib. Heap製作，可加速decrease key之動作 => O(1)Time in amorized cost

Time = O(V) + O(V) + O(logV)\*V次 + O(1)\*2E = O(V) + O(logV)\*V次 + O(E)

= O(VlogV + E)

Sollin's Algo

[DS版本才有]

Initially, 各頂點視為獨立Tree之Root

Steps:

1.每棵Tree各自挑出最小成本之樹邊

2.刪除重複挑出的邊，只保留一份即可

3.Repeat 1~2 Until 剩下一棵Tree or E為空

4.if S has < (n-1)條邊 then "No Spanning Tree"