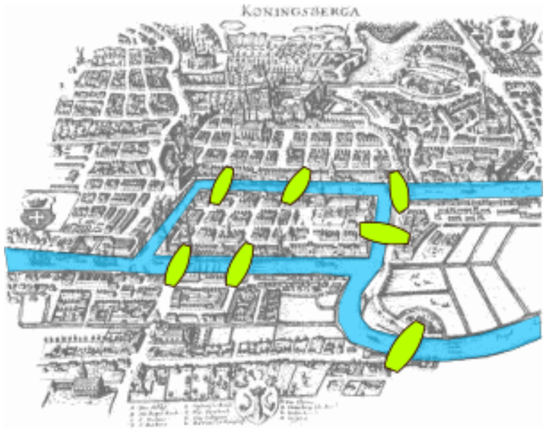
GRAPH BASICS

Michael Tsai

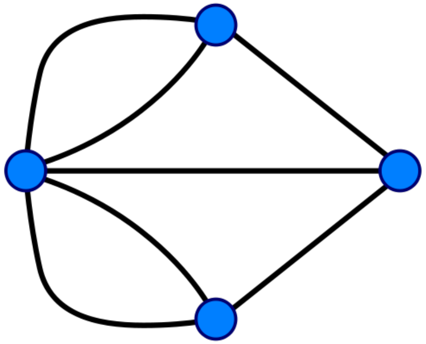
2017/5/23

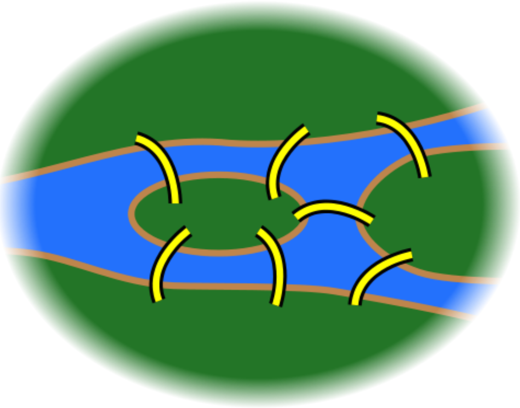
Königsberg Seven Bridge Problem

- 西元1736年, Euler嘗試著要解答這個問題:



- 右邊地圖中, 有沒有可能找出一條路徑, 使得每一條橋都走過一次之後又回到出發點?



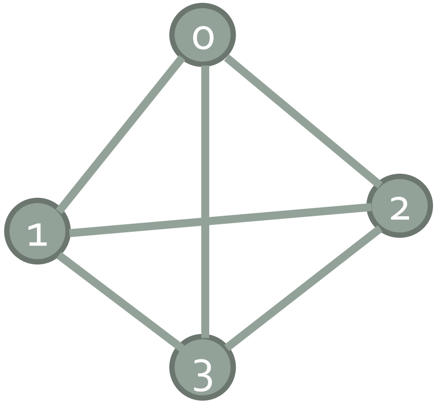


答案: Graph必須要是connected & 必須有0個或2個node有odd degree

A graph

- G: a graph, consists of two sets, V and E.
- V: a finite, nonempty set of vertices.
- E: a set of pairs of vertices.

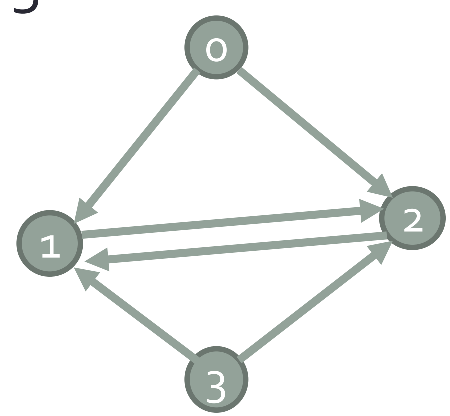
- G=(V,E)
- V={0,1,2,3}
- E={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}



Directed & undirected graph

- Graph中, edge有方向的叫做directed graph, 沒方向的叫做undirected graph
- Directed graph 通常又叫digraph
- Undirected graph 通常就只叫做graph

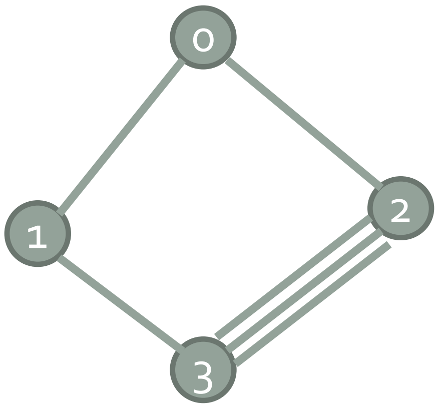
- (1,2) 和 (2,1) 在undirected graph是一樣的edge
- <1,2>和<2,1>在digraph是不一樣的edge
- V={0,1,2,3}
- E=?
- E={<0,1>,<0,2>,<1,2>,<2,1>,<3,1>,<3,2>}



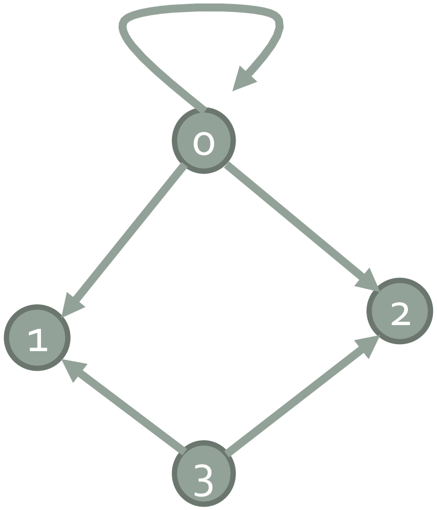
註: 課本裡面digraph也使用() 小括號, 不過我比較喜歡用<>角括號來表示digraph的edge, 覺得這樣比較清楚.

Self Edge & Multigraph

Multigraph



Graph with self-edges <v,v>

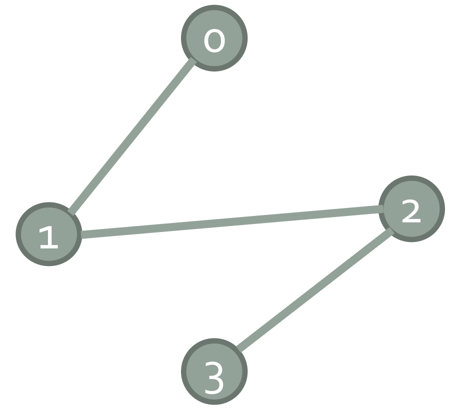


Maximum number of edges

- 一個有n vertices的graph, 最多有幾個edges?
- 一個有n vertices的digraph, 最多有幾個edges?

- 答案: graph: $\frac{n(n-1)}{2}$,
- digraph: $n(n-1)$

怎麼這麼多名詞XD



- 相鄰(adjacent):
  如果有edge (u,v), 那麼u, v兩vertices就是adjacent.

- 如果有edge <u,v> 那麼我們說
  v is adjacent to u.
  (有些資料結構課本用相反的定義T_T)

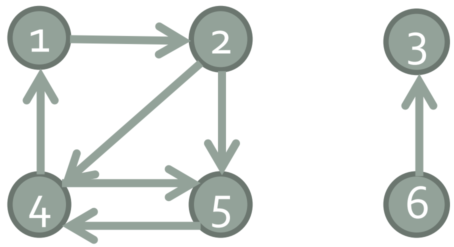
- 作用(incident): 如果有edge (u,v) 那麼u, v兩vertices就是incident (作用) on (在) u和 v上面
- <u,v> is incident **from** u and is incident **to** v.
- <u,v> **leaves** u and **enters** v.

- Subgraph: 如果G=(V,E), G'=(V',E')是G的subgraph, 則$V' \subseteq V$且$E' \subseteq E$

Degree of vertex

- Vertex的**degree**:
- 有幾個edge連在vertex上

- Digraph中
- 又可分為in-degree and out-degree
- **in-degree**: 進入vertex的edge數
- **out-degree**: 出去vertex的edge數
- **degree**=in-degree+out-degree



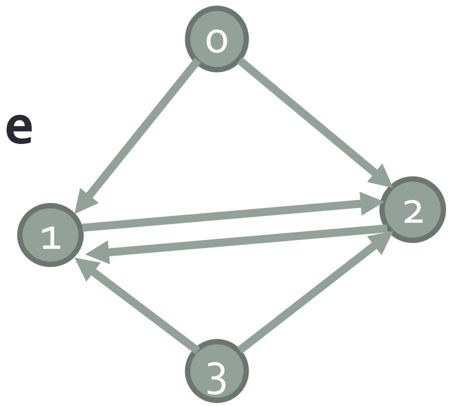
- 一個degree=0的vertex可稱為isolated

- Edge數和degree的關係: $e = \frac{\left(\sum_{i=0}^{n-1} d_i\right)}{2}$

路徑 (path)

- **Path**: 一條從u到v的path, 是一連串的vertices, $u, i_1, i_2, \dots, i_k, v$, 而且$(u, i_1), (i_1, i_2), \dots, (i_{k-1}, i_k), (i_k, v)$都是edge.

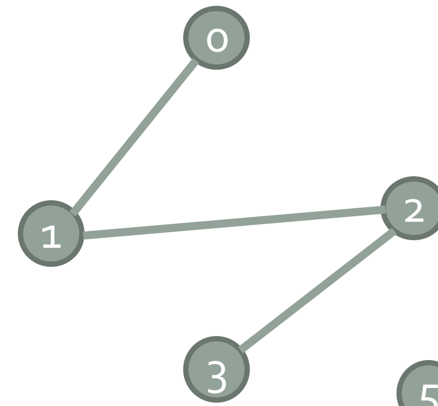
  (digraph的定義自行類推)
- 如果有一條path p從u到u', 則我們說u' is **reachable** from u via p.

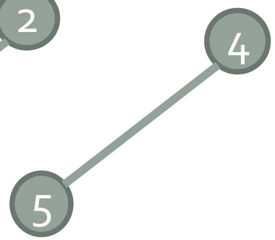


- 以上的path也可以寫成$u, i_1, i_2, \dots, i_k, v$.
- Path的**length**: 裡面有幾條edge

- **Simple path**: 除了u, v(起點和終點)以外, 其他的vertices都沒有重複過.
- **Cycle**: 一個u和v一樣的simple path
- **Subpath**:??
- 答案: 定義path的vertex sequence裡面的連續的一段.

有連接的(connected)





- **In undirected graph:**
- Vertices u and v are said to be **connected** iff
- there is a path from u to v. (graph & digraph)

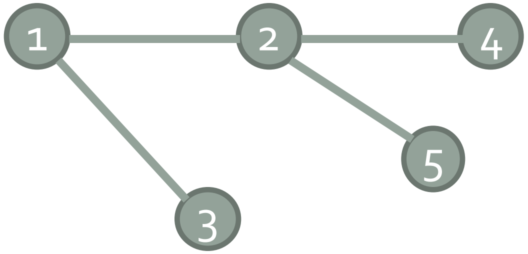
- **Connected graph**:
- iff every pair of distinct vertices u and v in V(G) is connected

- **Connected component** (也有人直接叫component):
- A maximal connected subgraph

- **Maximal**: no other subgraph in G is both connected and contains the component.

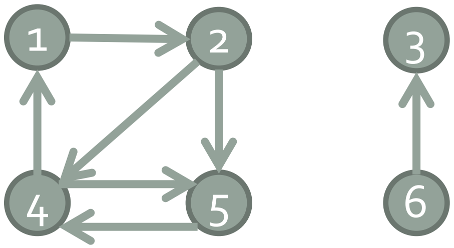
- 問: Tree是一個怎麼樣的graph?
- 答: connected and acyclic (沒有cycle) graph

- 問: Forest是一個怎麼樣的graph?
- 答: acyclic graph



強連接(strongly connected)

- **In digraph:**



- **Strongly connected**:
- G is strongly connected iff
- for every pair of distinct vertices u and v in V
- there is a directed path from u to v and also a directed path from v to u.

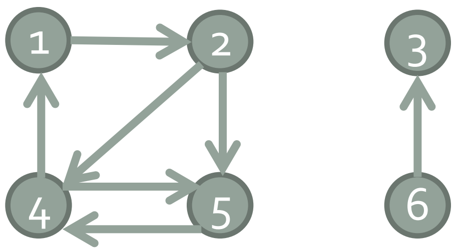
- **Strongly connected component**:
- A maximal subgraph which is strongly connected.

要怎麼表示一個graph呢?

- 主流表示法:
- Adjacency matrix (用array)
- Adjacency lists (用linked list)

- 兩者都可以表示directed & undirected graph

表示法I 用Array



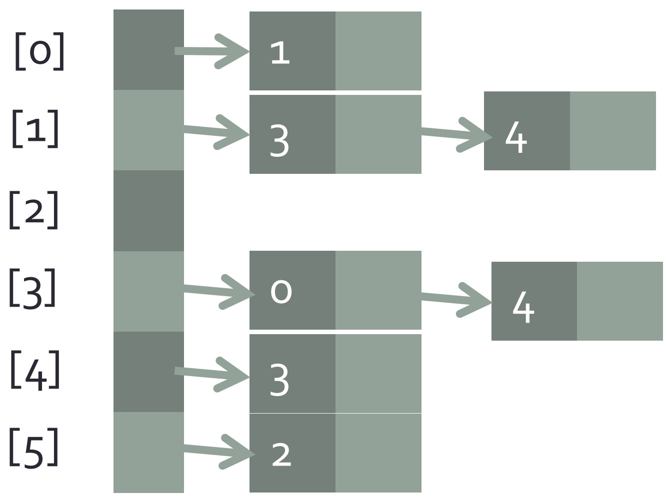
- 本方法叫做adjacency matrix
- index當作vertex號碼
- edge用matrix的值來表示:
- 如果有(i,j)這條edge, 則a[i][j]=1, a[j][i]=1. (graph)
- 如果有<i,j>這條edge, 則a[i][j]=1. (digraph)
- 對undirected graph, $a = a^T$ (也就是對對角線對稱)

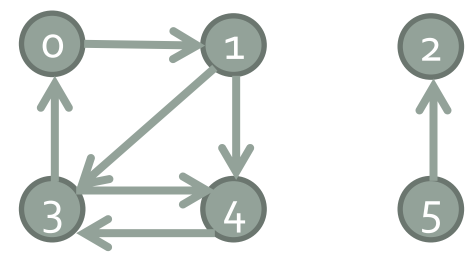
- 請同學解釋要怎麼建adjacency matrix☺

- 粉簡單. 那麼來看看好不好用:
- 如果要看總共有多少條edge? O(??)
- 答: $O(|V|^2)$
- 有沒有可能跟edge數(e)成正比呢?
- 通常$|E| \ll |V|^2$ 所以adjacency matrix裡面很空

表示法II 用linked list

- 本方法叫做adjacency lists
- 建立一個list array a[n] (n為vertex數目)
- 每個list裡面紀錄通往別的vertex的edge
- 問: 怎麼算in-degree?

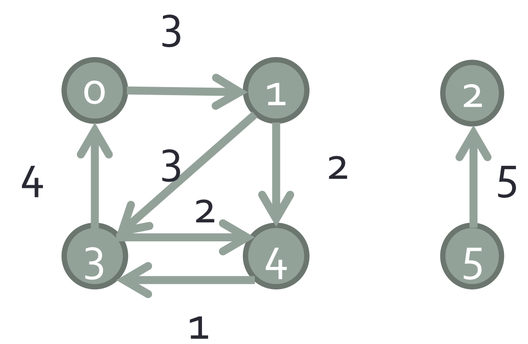




- 答: 如果要比較容易的話, 要另外建"inverse adjacency lists"
- 請同學告訴我怎麼畫☺

Weighted Edge

- Edge可以有"weight"
- 表示長度, 或者是需要花費
- 一個edge有weight的graph
- 又叫做network



- 想想看: 用剛剛的representation要怎麼儲存weight?
- 答:
- adjacency matrix可以用array的值來存
- adjacency list可以在list node裡面多開一個欄位存

兩者比較

- Adjacency lists:
- 用來表示sparse graphs時使用比較少空間
- 使用空間: $O(|V|+|E|)$

- Adjacency matrix:
- 要找某個edge (u,v)有沒有在graph裡面比較快
- 一個entry只需要1 bit (unweighted graph)
- 簡單容易, graph小的時候用adjacency matrix比較方便
- 使用空間: $O(|V|^2)$

Breadth-First Search (BFS)

- 給一個graph G=(V,E)及一個source vertex s
- 找出所有從s reachable的vertices
- 計算從s到每一個reachable的vertex的最少edge數目
- 產生breadth-first tree, s為root, 而其他reachable的vertices都在樹裡面
- Directed graph & undirected graph皆可

- 此方法會先找到所有距離s distance為k的vertex, 然後再繼續找距離為k+1的vertex

BFS Pseudo-code

```
BFS(G,s)
for each vertex u ∈ G.V − {s}
        u.color=WHITE
        u.d=∞
        u.pi=NIL
s.color=GRAY
s.d=0
s.pi=NIL
Q={}
ENQUEUE(Q,s)
while Q!={}
        u=DEQUEUE(Q)
        for each v ∈G.Adj[u]
                if v.color==WHITE
                        v.color=GRAY
                        v.d=u.d+1
                        v.pi=u
                        ENQUEUE(Q,v)
        u.color=BLACK
```

初始所有vertex的值

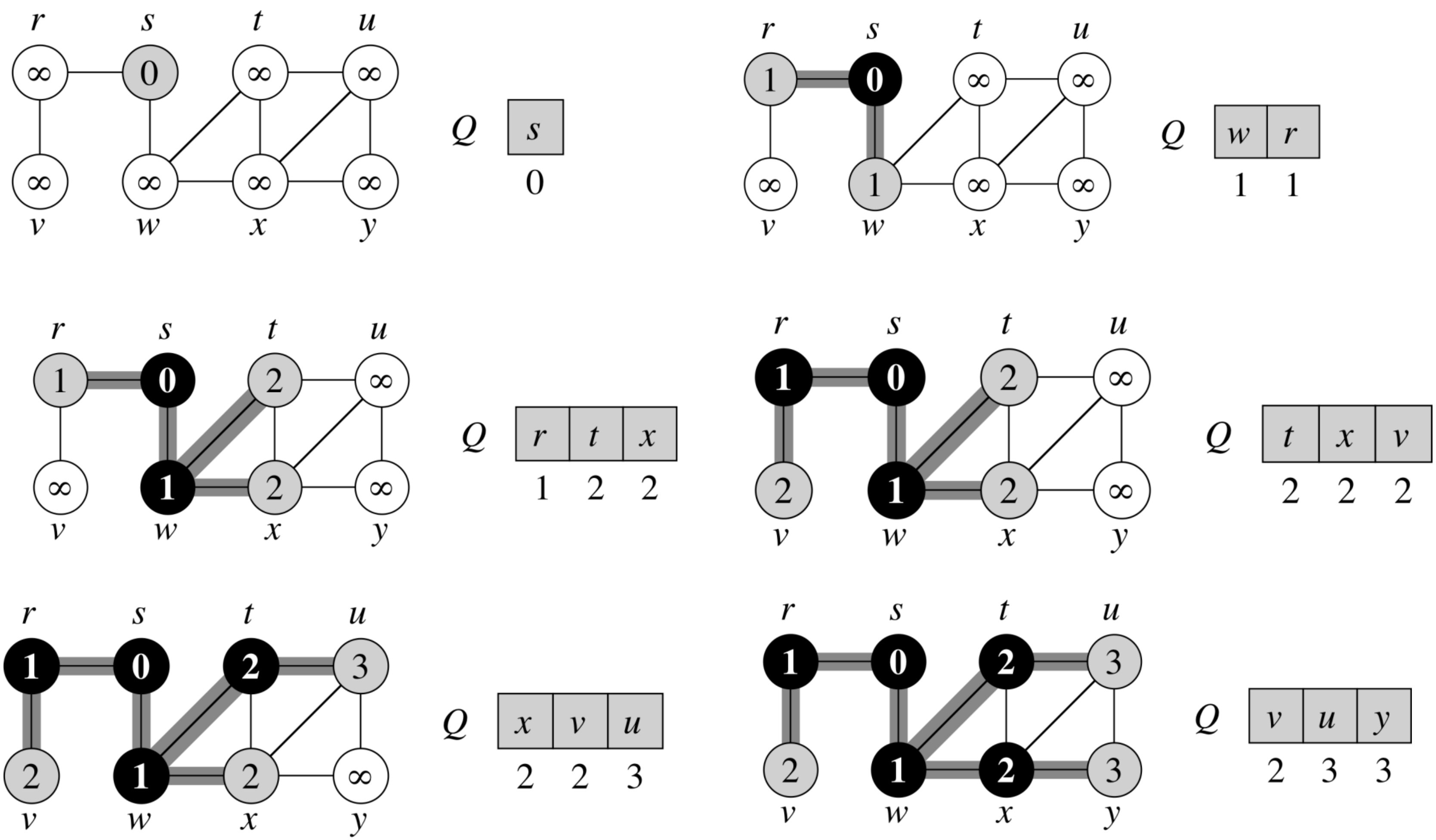
初始開始search的vertex s的值

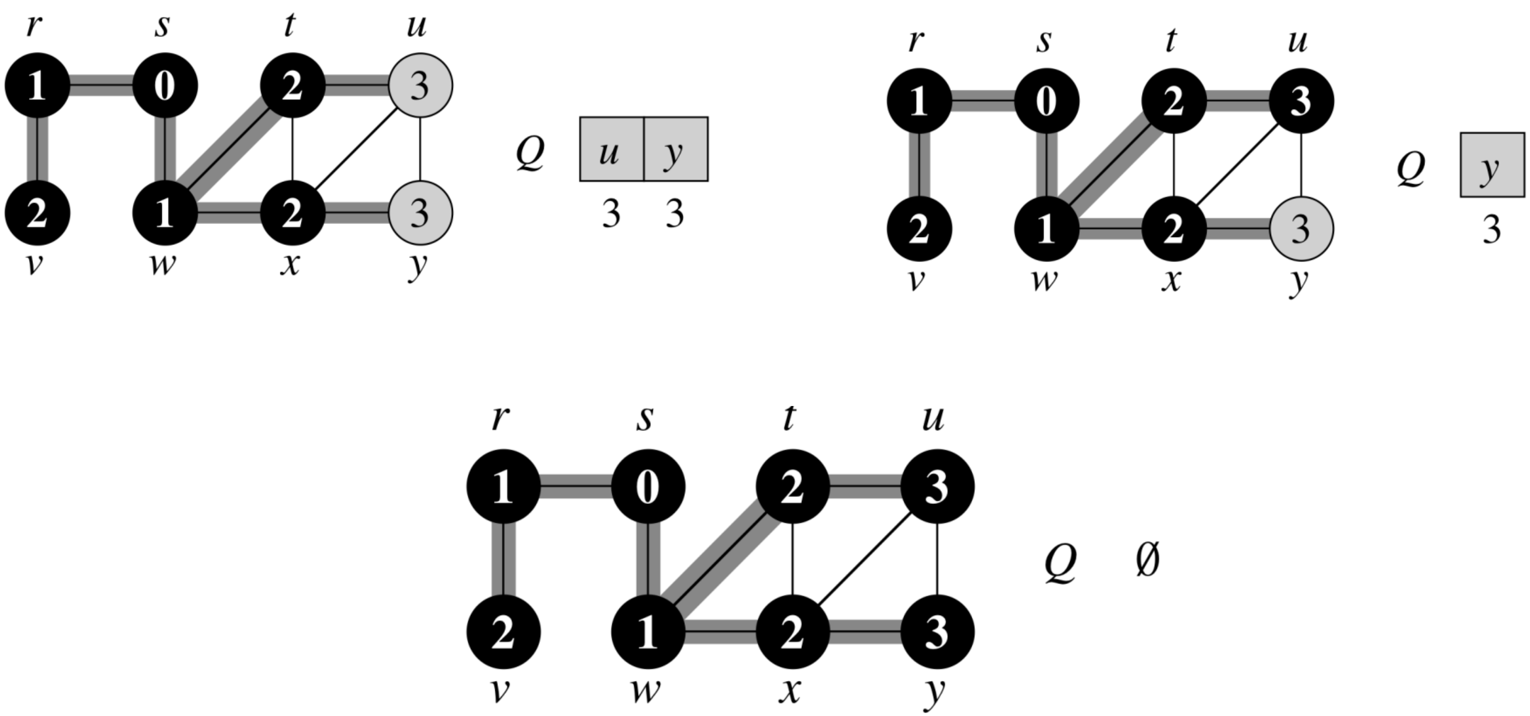
對每一個和u相連vertex

v.color: 用顏色來區別discover的狀況
WHITE: 還沒discovered
GRAY: discovered了, 但是和該vertex相連的鄰居還沒有都discovered
BLACK: discovered了且和該vertex相連的鄰居都已discovered

v.d: 和root的距離

v.pi: 祖先 (predecessor)


r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
r s t u
1 0 2 3
2 1 2 3
v w x y
Q Ø

BFS Pseudo-code

```
BFS(G,s)
for each vertex u ∈ G.V − {s}
        u.color=WHITE
        u.d=∞
        u.pi=NIL
s.color=GRAY
s.d=0
s.pi=NIL
Q={}
ENQUEUE(Q,s)
while Q!={}
        u=DEQUEUE(Q)
        for each v ∈G.Adj[u]
                if v.color==WHITE
                        v.color=GRAY
                        v.d=u.d+1
                        v.pi=u
                        ENQUEUE(Q,v)
        u.color=BLACK
```

$\boxed{O(V)}$

$\boxed{O(V+E)}$

$\boxed{O(E)}$

證明BFS可以找到最短路徑

- 定義: $\delta(s, v)$: s到v的最短路徑的長度(邊的數目)

- Lemma 22.1: G=(V,E)是一個directed或undirected graph. s是一個任意vertex. 則對任何edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.
- 證明:
- 如果從s開始, u是reachable, 那麼v也是.
- 這個狀況下, s→v的最短路徑只可能比$\delta(s, u) + 1$短 (最短路徑可能不經由u過來).不等式成立.
- 如果u不是reachable的話, 那麼不等式一定成立.

證明BFS可以找到最短路徑

- Lemma 22.2: G=(V,E)是一個directed或undirected graph. 對G及vertex s跑BFS. 則結束的時候, 每一個$v \in V$, BFS計算的$v.d \geq \delta(s,v)$.
- 證明:
- 使用歸納法證明. 假設為"每一個$v \in V$, BFS計算的$v.d \geq \delta(s,v)$".
- 一開始把s丟進queue的時候, 成立. $s.d = \delta(s,s) = 0$. 而其他的vertex $v.d = \infty > \delta(s,v)$.
- 當找到由edge (u,v)找到vertex v時, 我們可以假設$u.d \geq \delta(s,u)$. 且我們知道$v.d = u.d + 1 \geq \delta(s,u) + 1 \geq \delta(s,v)$.
- 每個v都只做以上步驟(更改v.d值)一次, 歸納法證明完成.

證明BFS可以找到最短路徑

- Lemma 22.3: 假設BFS執行在G=(V,E)上, queue裡面有以下vertices $\langle v_1, v_2, \ldots, v_r \rangle$, v_1是queue的頭, 而v_r是queue的尾. 則$v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1,2, \ldots, r-1$.
- 證明:
- 使用歸納法.
- 當queue一開始裡面只有s的時候成立.
- 現在我們必須證明每次dequeue或enqueue的時候, 都還是成立.
- dequeue的時候, v_2變成新的queue頭. 但$v_1.d \leq v_2.d$. 且$v_r.d \leq v_1.d + 1 \leq v_2.d + 1$.其他不等式都不變. 因此成立.
- enqueue的時候, 新加入的vertex v變成v_{r+1}.
- 此時我們剛剛把u拿掉(當時是queue的頭). 所以應該$v_1.d \geq u.d$. 所以$v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$.
- 且$v_r.d \leq u.d + 1$, so $v_r.d \leq u.d + 1 = v.d = v_{r+1}.d$.
- 其他的不等式都不變, 因此成立.

證明BFS可以找到最短路徑

- Corollary 22.4: vertices v_i 和 v_j在執行BFS時被enqueue且v_i在v_j之前被enqueue. 則當v_j被enqueue的時候$v_i.d \leq v_j.d$.
- 證明: 直接從Lemma 22.3就可以得到. 因為v.d只會被指定值一次(enqueue之時).

證明BFS可以找到最短路徑

- Theorem 22.5: 證明BFS正確性. BFS執行在G=(V,E)上, 從$s \in V$開始. BFS執行的時候會找出所有從s reachable的vertex $v \in V$. 結束的時候, 每個$v.d = \delta(s, v), \forall v \in V$.
- 證明:
- 假設有一些v.d不是$\delta(s, v)$. 讓v是其中$\delta(s, v)$最小的一個.
- Lemma 22.2說 $v.d \geq \delta(s, v)$, 所以現在$v.d > \delta(s, v)$.
- 此時v一定是從s reachable, 不然$\delta(s, v) = \infty \geq \mathrm{v.d}$.
- 假設u是s→v最短路徑上v的前一個vertex, 則$\delta(s, v) = \delta(s, u) + 1$
- 因為$\delta(s, u) < \delta(s, v)$, 所以$u.d = \delta(s, u)$ (已經假設v是其中$\delta(s, v)$最小的一個).
- $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$

證明BFS可以找到最短路徑

- 上頁得到$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$
- 考慮BFS從Queue裡面把u dequeue出來的時候.
- u的鄰居v們, 可能是WHITE, GRAY, 或BLACK
- 如果是WHITE, 則會設$v.d = u.d + 1$, 矛盾.
- 如果是BLACK, 則它之前已經被dequeue過. Corollary 22.4說$v.d < u.d$, 矛盾.
- 如果是GRAY, 則它是剛剛dequeue某個vertex w的時候被改成GRAY的(比u dequeue的時間早). 所以$v.d = w.d + 1$. 但Corollary 22.4說$w.d \le u.d$, 因此$v.d = w.d + 1 \le u.d + 1$, 矛盾.
- 因此原假設不成立.證明完畢.

Breadth-First Tree

- BFS做出一棵樹: Breadth-First Tree
- 這個樹其實也就是BFS產生出來的predecessor subgraph of G:
- $G_\pi = (V_\pi, E_\pi)$
- $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$
- 從s到任何$v \in V_\pi$, 都只有一條path, 而此path即為s到v的最短路徑.

Depth-first Search

- 當可能的時候, 就往更深的地方找去 → Depth-first

- 和Breadth-first Search不同的地方:
- 會長出一個forest (多棵樹)
- 會記錄timestamp: 開始找到的時間(變成灰色)和完成所有和它相鄰的vertex的時間(變成黑色)

使用的structure

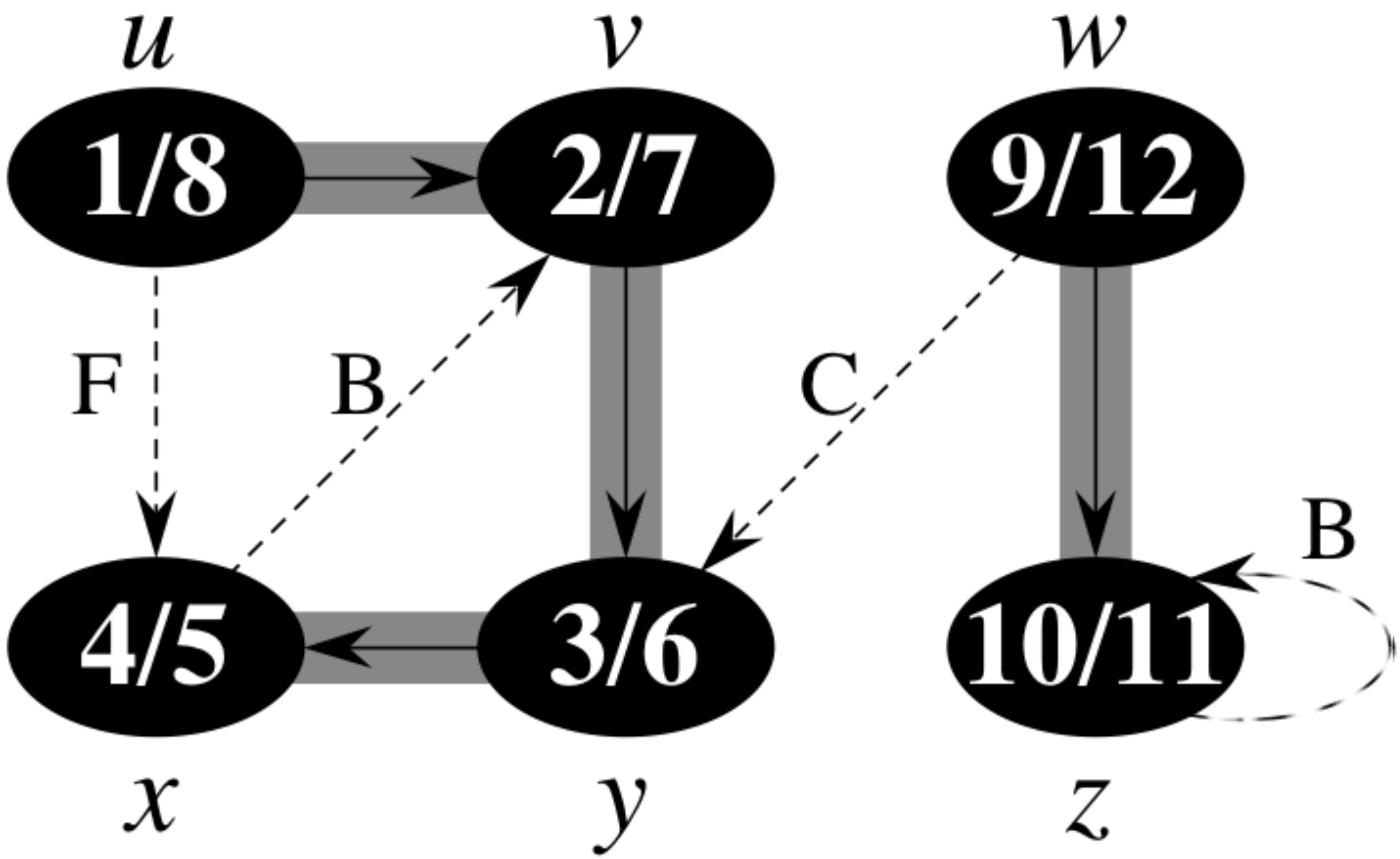
- 每一個vertex u有以下的structure:
- u.color: 紀錄目前這個vertex的狀態
  - WHITE: 這個vertex還沒被discover
  - GRAY: 這個vertex被discover了, 但是和它相連的vertex還沒都被discover
  - BLACK: 這個vertex及和它相連的vertex都已經被discover了
- u.pi: 紀錄它的前一個vertex (祖先) 是誰
- u.d: discover的時間 (從WHITE→GRAY的時間)
- u.f: finish的時間 (從GRAY→BLACK的時間)

- 時間(timestamp)總共會從1跑到2$|V|$, 因為每個vertex discover或finish會各增加一次timestamp.

Pseudo-Code

```
DFS(G)
for each vertex u ∈ G.V
  u.color=WHITE
  u.pi=NIL
time=0
for each vertex u ∈ G.V
      if u.color==WHITE
            DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u)
time=time+1
u.d=time
u.color=GRAY
for each v ∈G.Adj[u]
      if v.color==WHITE
            v.pi=u
            DFS-VISIT(G,v)
u.color=BLACK
time=time+1
u.f=time
```



如果邊的排列方式(Adjacency List)不同, 很可能會造成DFS出來的結果不同

試試看, 如果<u,x>比<u,v>先被走過, 請問會變怎麼樣?

執行時間

```
DFS(G)
for each vertex u ∈ G.V
  u.color=WHITE
  u.pi=NIL
time=0
for each vertex u ∈ G.V
      if u.color==WHITE
            DFS-VISIT(G,u)
```

$\boxed{\Theta(V)}$

$\boxed{\Theta(V)}$

執行時間

總合起來: $\Theta(V+E)$

```
DFS-Visit(G,u)
time=time+1
u.d=time
u.color=GRAY
for each v ∈G.Adj[u]
        if v.color==WHITE
            v.pi=u
            DFS-VISIT(G,v)
u.color=BLACK
time=time+1
u.f=time
```

$\Theta(V)$

$\Theta(E)$

$\Theta(V)$

DFS-Visit會執行|V|次

這個部分對每個vertex v會執行|Adj[v]|次 (vertex v的adjacency list長度)

把所有vertex的adjacency list長度加起來=|E|

動腦時間

- 如果我們改用Adjacency Matrix的話, 執行時間會變怎麼樣呢?

```
DFS-Visit(G,u)
time=time+1
u.d=time
u.color=GRAY
for each v ∈G.Adj[u]
        if v.color==WHITE
                v.pi=u
                DFS-VISIT(G,v)
u.color=BLACK
time=time+1
u.f=time
```

??

Depth-first Forest

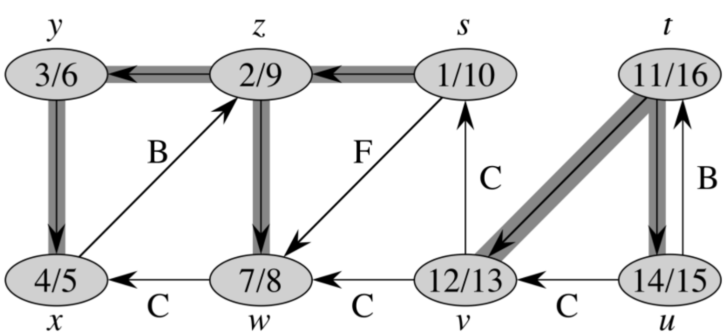
- 類似於breadth-first search, depth-first search也可以產生predecessor subgraph:
- $G_\pi = (V, E_\pi)$
- $E_\pi = \{(v.\pi, v): v \in V \ and \ v.\pi \neq NIL\}$.

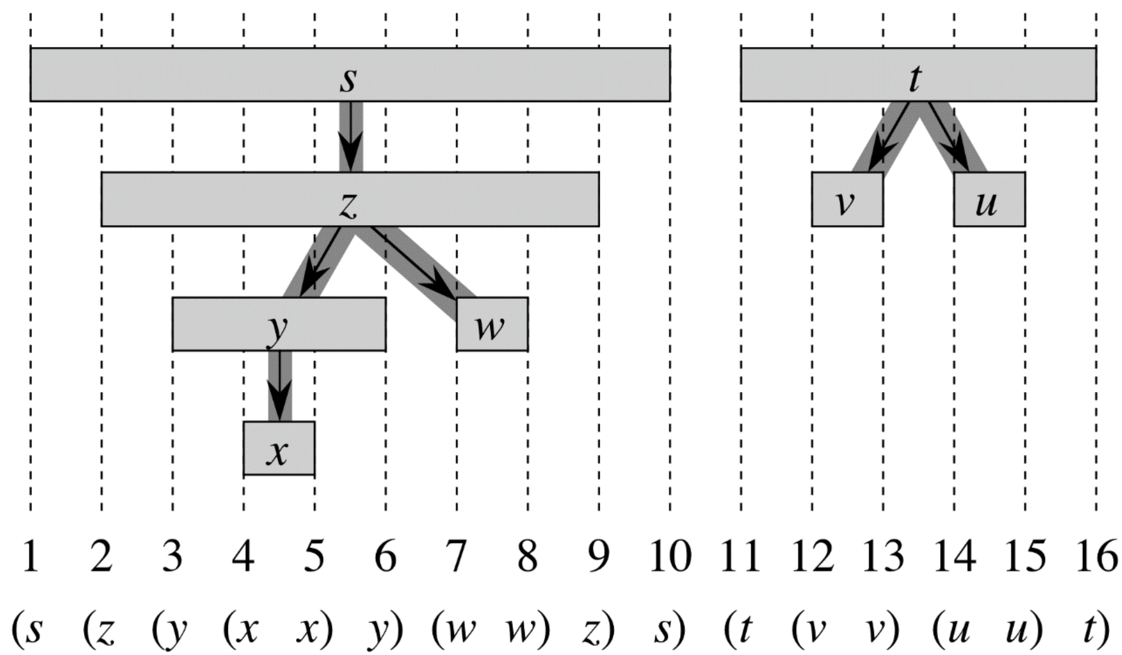
- 而Predecessor subgraph正好可以產生一個由多棵depth-first tree組成的 depth-first forest.

- 在E_π裡面的edge叫做tree edge.

Depth-first Search的括號結構性質

- Parenthesis structure: (括號結構)
- 如果我們用 (代表 vertex u的discovery被找到
- 用)代表 vertex u 的結束
- 則整個dfs的尋找歷史會使得vertex們的左括號和右括號都
  會不是互相包含
  就是相互分離





Reading assignment:
證明請見Cormen課本p.607-608

白路(White-path) 性質

從括號結構性質可得:
在depth-first forest of G裡面, v是u的子孫 iff u.d<v.d<v.f<u.f.

在G的depth-first-search forest裡面:

v是u的子孫 ⟺ 設定u.d的時候, u到v有一條全白的路徑

- (➔) :
- 如果v=u的話, 那設定u.d的時候, u還是白的.
- 如果v真的是u的子孫的話, u.d<v.d (v discover的時間比較晚)
- 此時v一定還是白的 (才在設定u.d而已)
- 既然v可以是任何u的子孫的話, 表示在u到v的路上(都是u的子孫)都也應該是白的

白鷺鷥?

u到v的全白路徑, 但是v不是u的子孫.
假設在白路徑上, 每個v之外的node, 都變成u的子孫.
(意思其實就是取v使得他是u->v白路徑上第一個不是u子孫的node)
取w為v的u→v路徑上的前一個 (u和w有可能是同一個點)
(1)由括號性質得$w.f \leq u.f$
(2)既然有白路徑, 所以$u.d < v.d$
(3)v會在w結束前被找到(因為v和w有一條edge) 所以$v.d < w.f$
合併以上$u.d < v.d < w.f \leq u.f$ → v.d被包含在 u.d和u.f中間了
該為包含在 [u.d, u.f]裡面的狀況

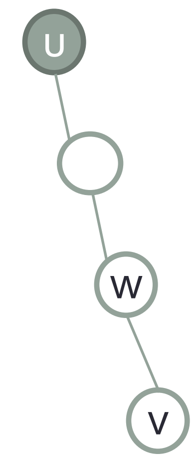
白路(White-path) 性質

從括號結構性質可得:
在depth-first forest of G裡面, v是u的子孫 iff u.d<v.d<v.f<u.f.

在G的depth-search forest裡面:

v是u的子孫 ⟷ 設定u.d的時候, u到v有一條全白的路徑

- (←) :
- 假設在u.d的時候, 有一條從u到v的全白路徑, 但是v不是u的子孫.
- 假設在白路徑上, 每個v之外的node, 都變成u的子孫.
  (意思其實就是取v使得他是u->v白路徑上第一個不是u子孫的node)
- 取w為v的u→v路徑上的前一個 (u和w有可能是同一個點)
- (1)由括號性質得$w.f \leq u.f$
- (2)既然有白路徑, 所以$u.d < v.d$
- (3)v會在w結束前被找到(因為v和w有一條edge) 所以$v.d < w.f$
- 合併以上$u.d < v.d < w.f \leq u.f$ → v.d被包含在 u.d和u.f中間了
- 由此可見, [v.d v.f] 應該為包含在 [u.d, u.f]裡面的狀況
- v為u之子孫, 矛盾



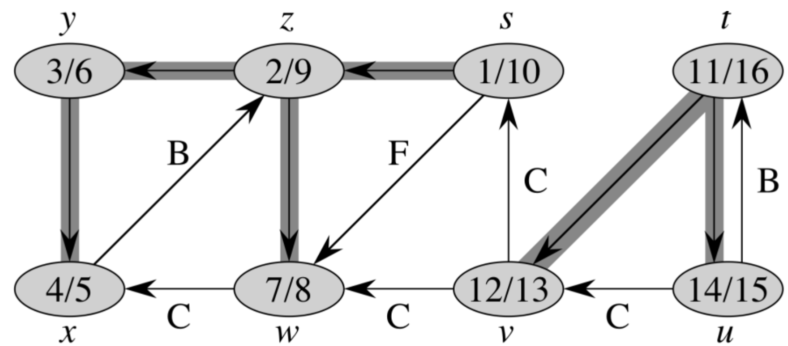
Depth-first forest 中的edge

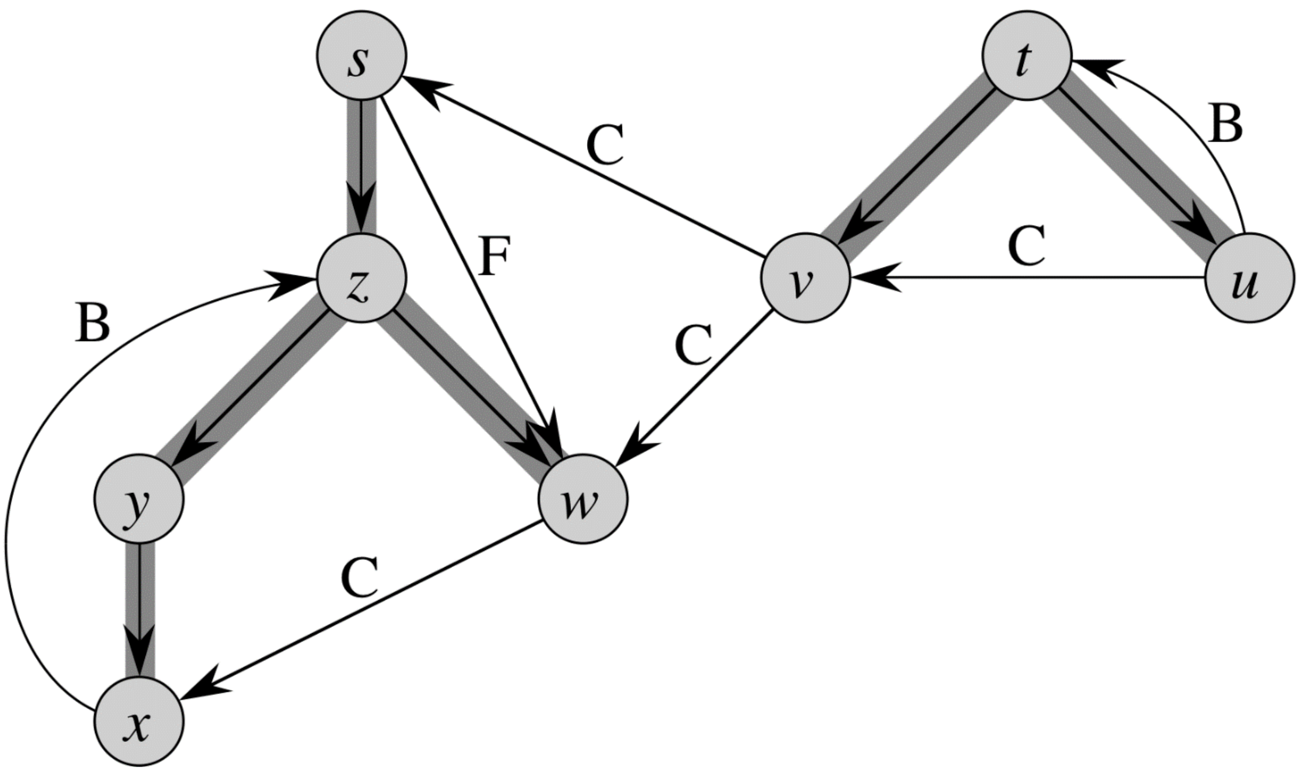
- 有四種:

1. Tree edge: 在depth-first forest裡面的邊叫做tree edge. 如果v是經由(u,v) discover的, 那(u,v)就是tree edge.
2. Back edge: 連接u到它的祖先v的邊(u,v)叫做back edge. Self-loop也算做是back edge的一種.
3. Forward edge: 連接u到它的子孫v的nontree edge (u,v).
4. Cross edge: 所有其他的edge. 可以是連接同一棵depth-first tree的邊, 或者是連接不同depth-first tree的邊.

例子


我是栗子


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


s
t
C
B
F
C
z
v
u
B
C
y
w
C
x

如何分辨是什麼邊呢?

- 當我們第一次碰到edge (u,v)的時候, v的顏色告訴我們它是什麼邊:

1. WHITE → tree edge
2. GRAY → back edge
3. BLACK → forward 或 cross edge
    1. u.d<v.d的話就是一條forward edge
    2. u.d>v.d的話就是一條cross edge

- 在undirected graph的depth-first forest 裡面沒有forward edge or cross edge. 想想看為什麼?