

# Algoritma BFS dan DFS

wijanarto

# Algoritma Graph

- Algoritma traversal di dalam graf adalah mengunjungi simpul-simpul dengan cara yang sistematis.
- **Pencarian Melebar** (*Breadth First Search* atau BFS),
- **Pencarian Mendalam** (*Depth First Search* atau DFS).

# Pencarian Melebar

## *(Breadth First Search* atau BFS)

- Idenya mirip dengan algo prim dan dijkstra
- Traversal dimulai dari simpul  $v$ .
- Algoritma:
  - Kunjungi simpul  $v$ ,
  - Kunjungi semua simpul yang bertetangga dengan simpul  $v$  terlebih dahulu.
  - Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.
- Jika graf berbentuk pohon berakar, maka semua simpul pada aras  $d$  dikunjungi lebih dahulu sebelum simpul-simpul pada aras  $d + 1$ .

# Graph Searching Algorithm

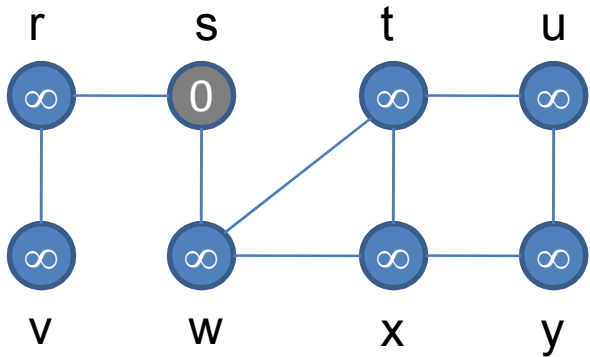
- Pencarian Sistemik pada setiap edge dan vertek dari graph G
- Graph  $G=(V,E)$ , directed atau undirected
- Aplikasi
  - Compiler
  - Graphics
  - Maze
  - Mapping
  - Network : routing, searching, clustering, dsb

# Representasi BFS

- Pada umumnya graf di representasikan baik secara array ataupun list
- Dalam kuliah ini menggunakan Link LIST dan **queue**

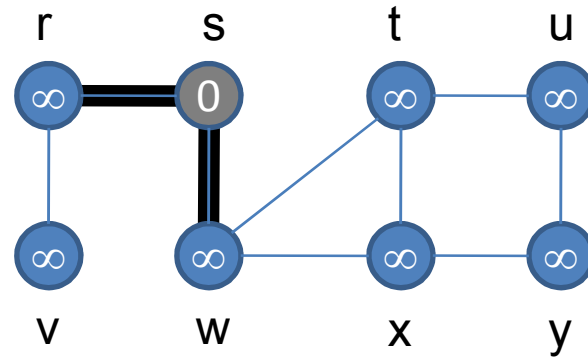
```
struct node {  
    int data;  
    struct node *link;  
};
```

# Contoh bfs



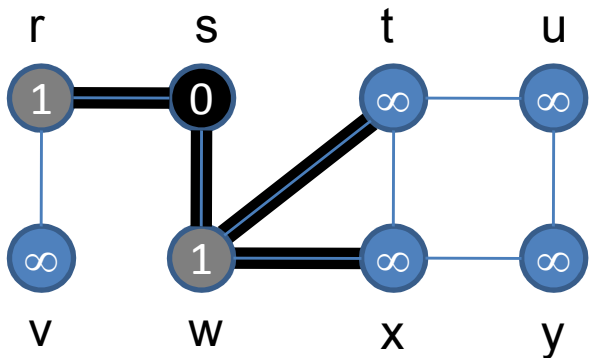
Q

s
0



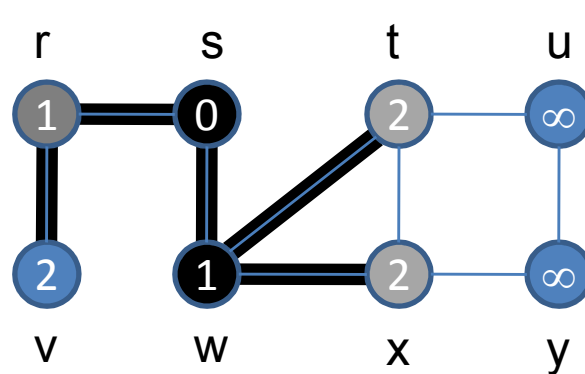
Q

s		
0		



Q

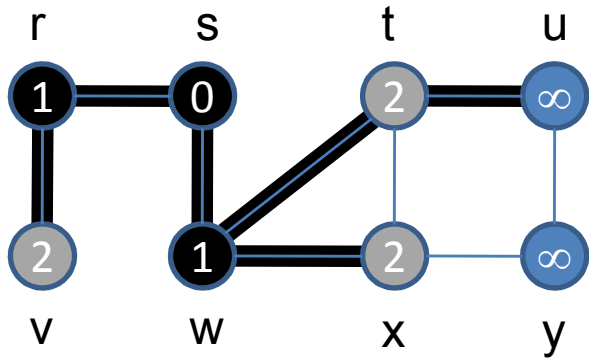
w	r	
1	1	



Q

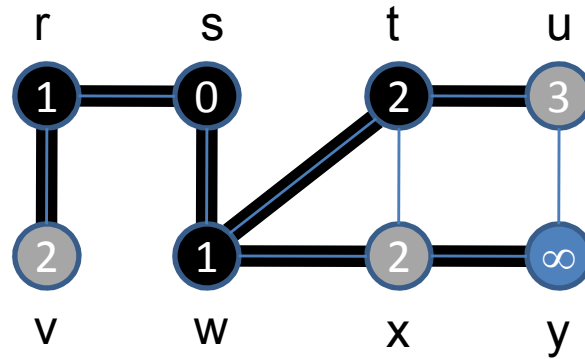
r	t	x
1	2	2

# Contoh bfs



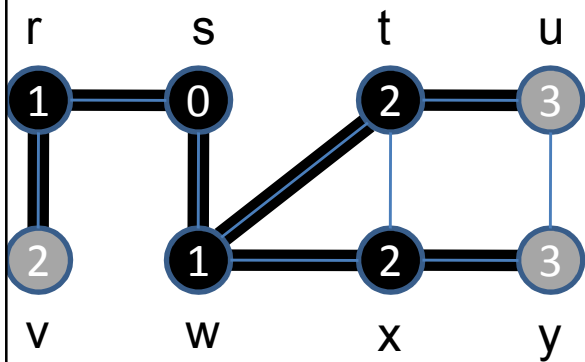
Q

t	x	v
2	2	2



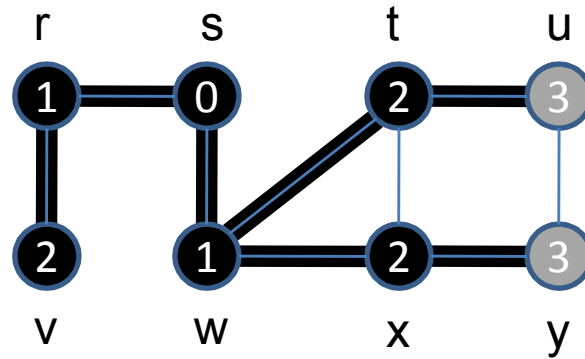
Q

x	v	u
2	2	3



Q

v	u	y
2	3	3

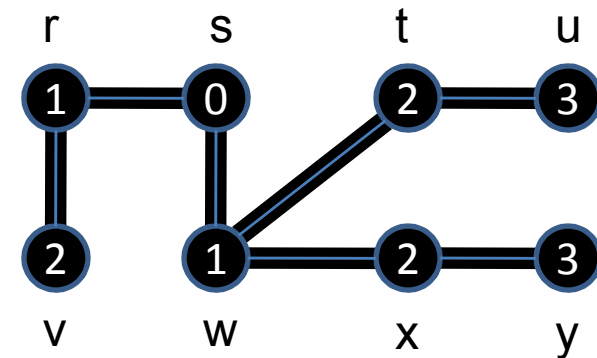


Q

u	y	
3	3	

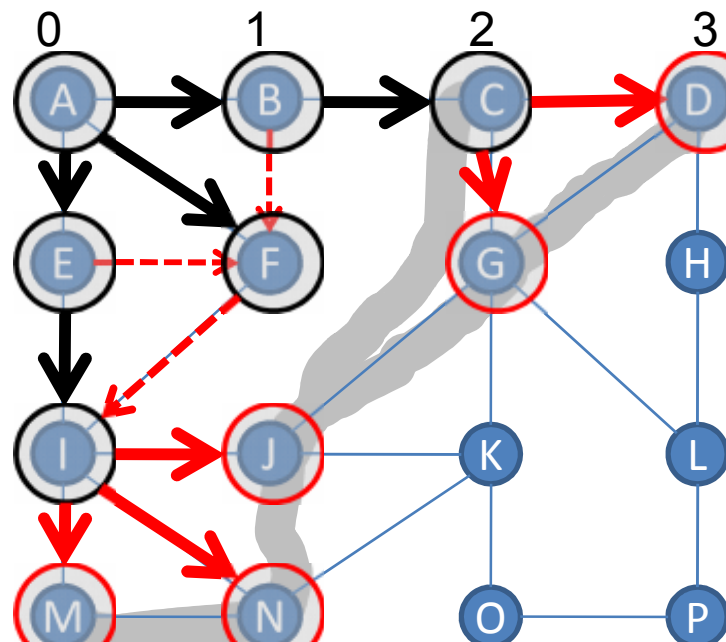
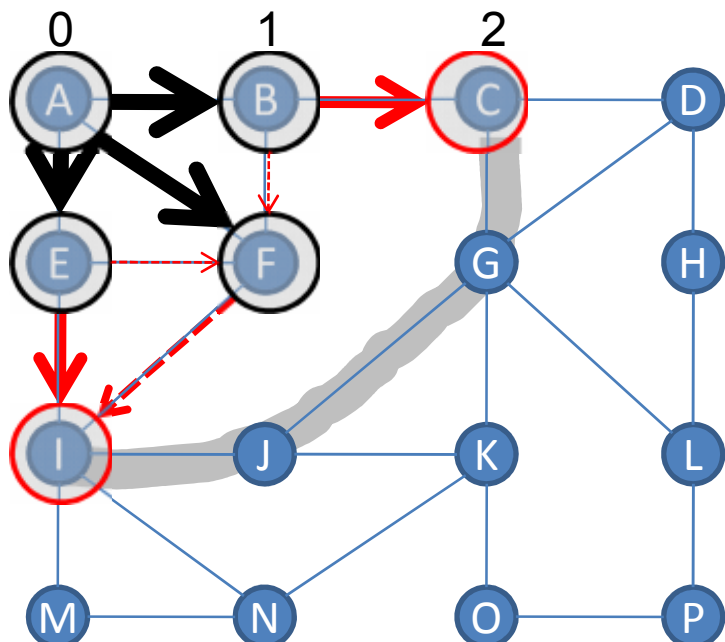
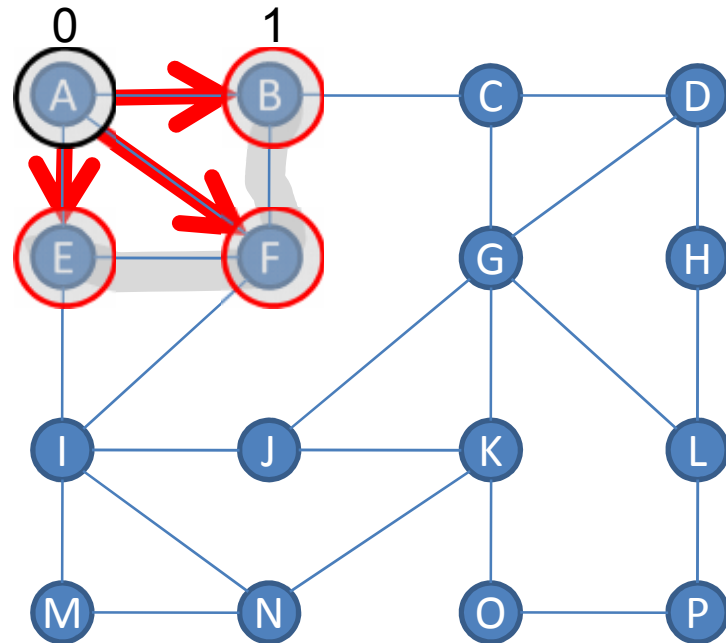
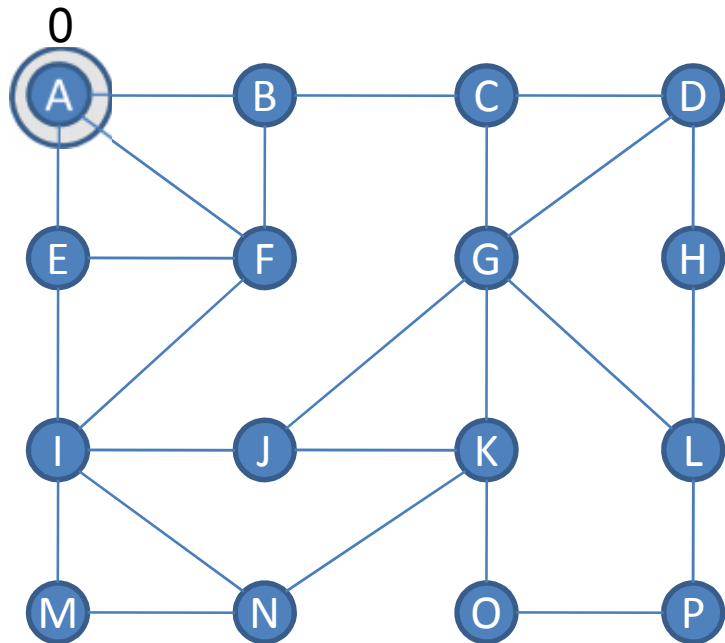
# Contoh bfs

- 1,2,3 adalah label vertek dalam G
- 1 merupakan panjang shortest path dari s  
(source)  $\rightarrow$  (s-w,s-r)
  - 2 berarti ada 2 edge
    - (s-t,s-x,s-v)
  - 3 berarti ada 3 edge
    - (s-u,s-y)

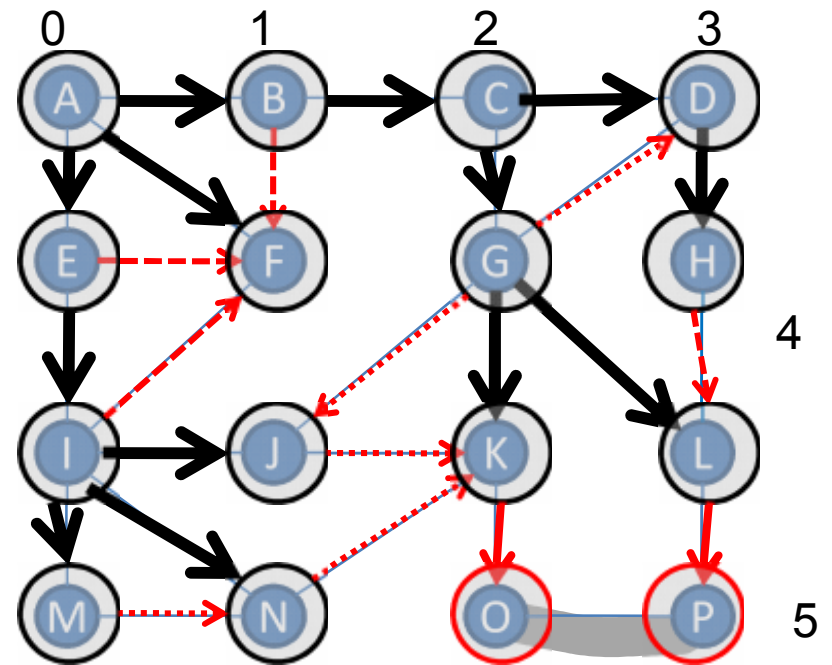
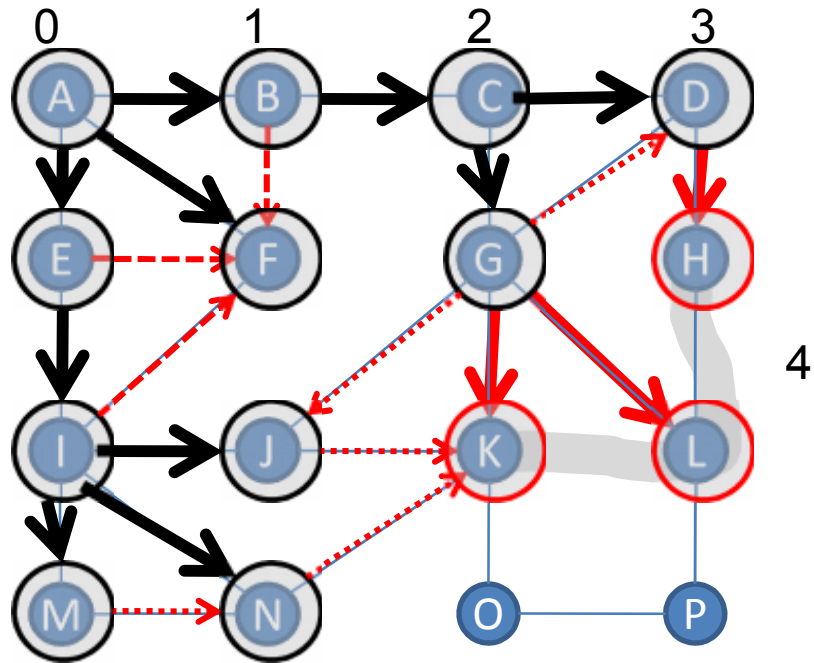




# Bfs secara grafikal



# Bfs secara grafikal



# Algoritma bfs 1

BFS (G, s)

```
For each vertek  $u \in V[G] - \{s\}$  do
```

```
    color[u] ← putih
```

```
    d[u] ←  $\infty$  {tdk di labeli}
```

```
    pi[u] ← nil {predesesor vertek}
```

INIT semua  
VERTEK

```
Color[s] ← abu2
```

```
D[s] ← 0
```

```
Pi[s] ← nil
```

```
Q ← {s}
```

INIT BFS  
dengan s  
(source)

```
While  $Q \neq \emptyset$  do
```

```
    u ← head[Q]
```

```
    for each  $v \in \text{adj}[u]$  do
```

```
        if color[v] ← putih then
```

```
            color[v] ← abu2
```

```
            d[v] ← d[u] + 1
```

```
            pi[v] ← u
```

```
            ENQUEUE (Q, v)
```

Tangani  
seluruh anak s  
SEBELUM  
menangani  
anak dari  
anaknya

```
DEQUEUE (Q)
```

```
color[u] ← Hitam
```

# Algoritma bfs 2

```
Bfs (v) {  
    Q ← ∅;  
    mark[v] ← visited;  
    Q ← enqueue (v);  
    while Q ≠ ∅ do {  
        u ← first(Q)  
        dequeue (u)  
        for w adj pada u do {  
            if mark[w] ≠ visited then  
                mark[w] ← visited  
            Q ← Enqueue (w)  
        }  
    }  
}
```

# Algoritma Dalam List (1)

```
void BFS(VLink G[], int v) {
    int w; VISIT(v); /*visit vertex v*/
    visited[v] = 1; /*tandai v, telah di kunjungi dengan : 1 */
    ADDQ(Q,v);
    while(!QMPTYQ(Q)) {
        v = DELQ(Q); /*Dequeue v*/
        w = FIRSTADJ(G,v); /*cari tetangga pertama, return -1 if tidak ada */
        while(w != -1) {
            if(visited[w] == 0) {
                VISIT(w); /*visit vertex v*/
                ADDQ(Q,w); /*Enqueue vertek yang sedang di kunjungi w*/
                visited[w] = 1; /*tandai w telah di kunjungi*/
            }
            /*cari tetangga selanjutnya, return -1 if tidak ada*/
            w = NEXTADJ(G,v);
        }
    }
}
```

# Algoritma Dalam List (2)

```
void TRAVEL_BFS(VLink G[],int visited[],int n) {
    int i;
    /* Inisialisasi seluruh vertek
       dengan visited[i] = 0 */
    for(i = 0; i < n; i ++) {
        visited[i] = 0;
    }
    /* Lakukan BFS ke seluruh vertek dlm G*/
    for(i = 0; i < n; i ++)
        if(visited[i] == 0)    BFS(G,i);
}
```

# BFS

## Properti dan running time

- $O(V+E)$
- $G=(V,E)$ , bfs mencari seluruh vertek yg dapat di raih dari source  $s$
- Untuk setiap vertek pada level  $l$ , path bfs tree antara  $s$  dan  $v$  mempunyai  $l$  edge dan selain path dlm  $G$  antara  $s$  dan  $v$  setidaknya mempunyai  $l+1$  edge
- Jika  $(u,v)$  adalah edge maka jumlah level  $u$  dan  $v$  di bedakan setidaknya satu tingkat
- Bfs menghitung seluruh jarak terpendek ke seluruh vertek yang dapat di raihnya.

# Kegunaan BFS

- Memeriksa apakah graph terhubung
- menghitung spanning forest graph
- Menghitung, tiap vertex dlm graph, jalur dg jumlah edge minimum antara vertex awal dan current vertex atau ketiadaan path.
- Menghitung cycle dlm graph atau ketiadaan cycle.
- $O(V + E)$ .



# Algoritma BFS dg matrik 1

```
void buildadjm(int adj[][MAX], int n) {  
    int i,j;  
    printf("enter adjacency matrix \n",i,j);  
    for(i=0;i<n;i++)  
        for(j=0;j<n;j++)  
            scanf("%d",&adj[i][j]);  
}
```

# Algoritma BFS dg matrik 2

```
struct node *addqueue(struct node *p,int val) {
    struct node *temp;
    if(p == NULL) {
        p = (struct node *) malloc(sizeof(struct node));
        /* insert the new node first node*/
        if(p == NULL) { printf("Cannot allocate\n"); exit(0); }
        p->data = val; p->link=NULL;
    } else {
        temp= p;
        while(temp->link != NULL) { temp = temp->link; }
        temp->link = (struct node*)malloc(sizeof(struct node));
        temp = temp->link;
        if(temp == NULL) { printf("Cannot allocate\n"); exit(0); }
        temp->data = val; temp->link = NULL;
    }
    return(p);
}
```

# Algoritma BFS dg matrik 3

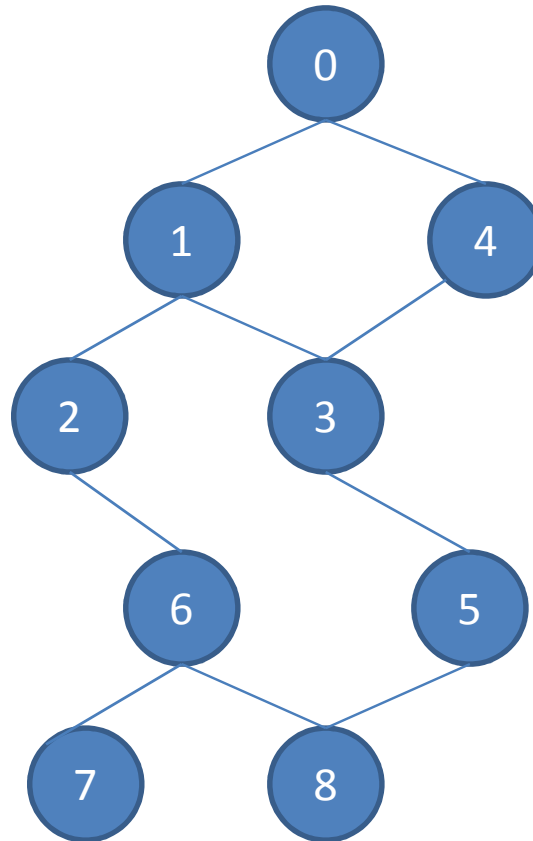
```
struct node *deleteq(struct node *p,int *val) {
    struct node *temp;
    if(p == NULL) {
        printf("queue is empty\n");
        return(NULL);
    }
    *val = p->data;
    temp = p;
    p = p->link;
    free(temp);
    return(p);
}
```

# Algoritma BFS dg matrik 4

```
void bfs
(int adj[][MAX],int x,int visited[],int n, struct node **p){
int y,j,k;
*p = addqueue(*p,x);
do{ *p = deleteq(*p,&y);
    if(visited[y] == 0){
        printf("\nnode visited = %d\t",y);
        visited[y] = 1;
        for(j=0;j<n;j++)
            if((adj[y][j] ==1) && (visited[j] == 0))
                *p = addqueue(*p,j);
    }
}while((*p) != NULL);
}
```

# Contoh pada Matrik 9X9

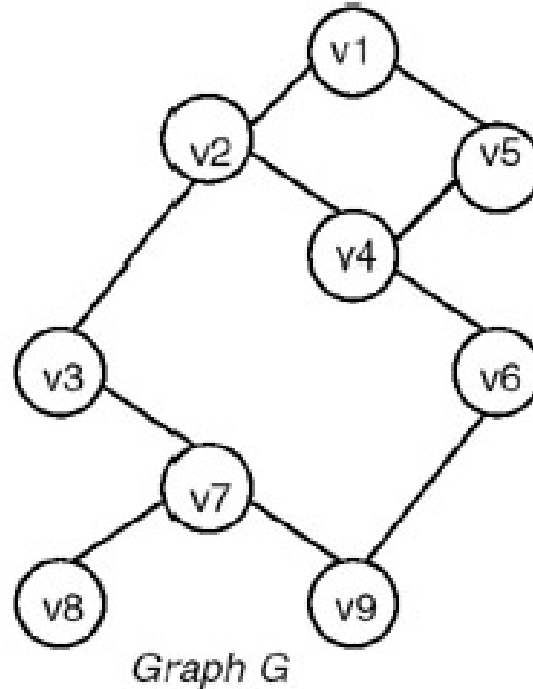
	1	2	3	4	5	6	7	8	9
1	0	1	0	0	1	0	0	0	0
2	1	0	1	1	0	0	0	0	0
3	0	1	0	0	0	0	1	0	0
4	0	1	0	0	1	1	0	0	0
5	1	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	1
7	0	0	1	0	0	0	0	1	1
8	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	1	1	0	0



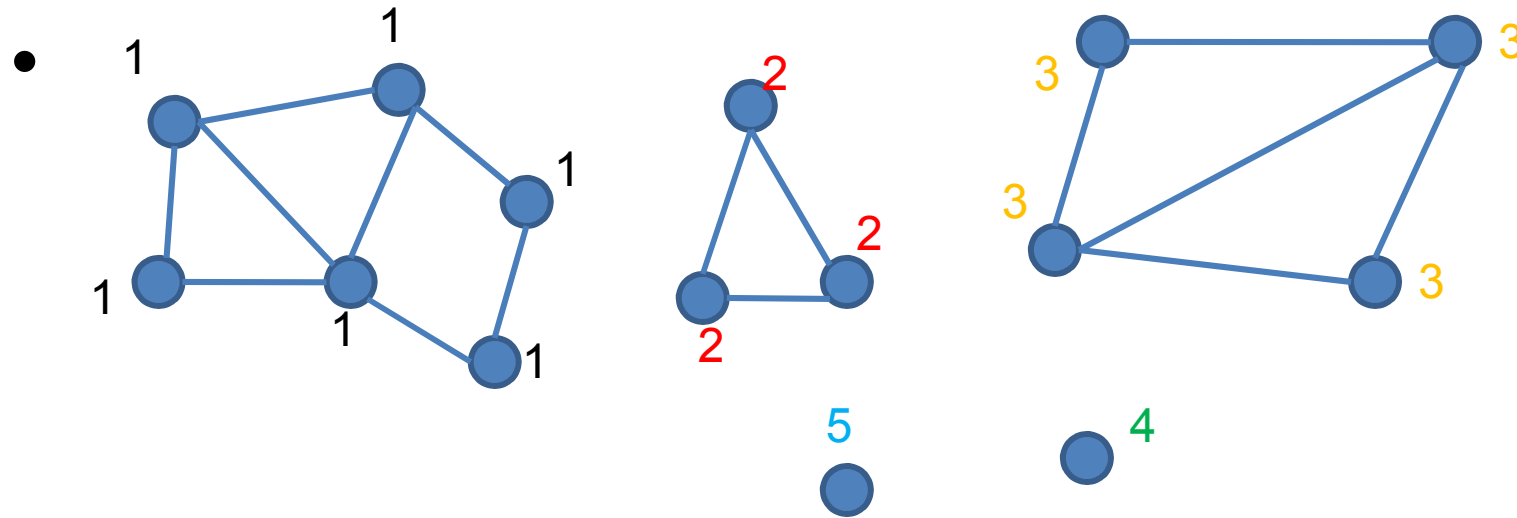
node visited = 0  
node visited = 1  
node visited = 4  
node visited = 2  
node visited = 3  
node visited = 6  
node visited = 5  
node visited = 7  
node visited = 8

# Contoh lain

- Awal simpul adalah V1, dari graf G di bawah
- Kunjungan BFS menghasilkan :
- v1,v2,v5,v3,v4,v7,v6,v8,v9



# Aplikasi bfs (connected component)



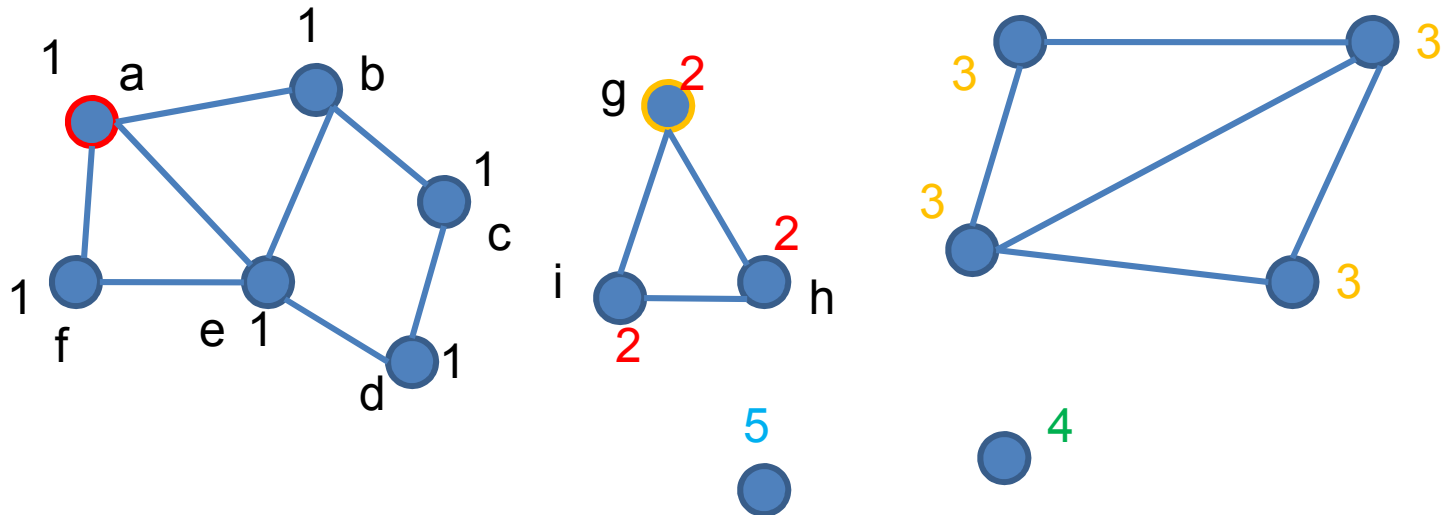
No CC

1	1	1	2	2	2	5	4	3	3	3
---	---	---	---	---	---	---	---	---	---	---

- Jika ada label yang elemennya sama berarti terdapat CC dan Sebaliknya
- BAGAIMANA KITA MEMBUATNYA ??

# Aplikasi bfs (connected component)

- Inisialisasi seluruh Vertek dlm G dengan 0
- Mulai dari sembarang vertek dengan nilai 0 dalam CC lalu lakukan bfs
- Cari vertek dg nilai 0 selanjutnya dan lakukan bfs lagi



No CC	d	g	a	h	b	f	i	e	j	k	l	m	n	o
	1	2	1	2	1	0	1	2	1	0	1	0	1	0



# Aplikasi bfs (connected component)

- Running Time =  $O(m+n)$
- $M = \# \text{ edge}$
- $N = \text{scanning array untuk mencari CC}$
- Terdapat  $m$  saat kita melakukan bfs, sekaligus  $n$  saat melabeli CC dalam array

# Aplikasi bfs

## (Bipartite graph )

- Bipartite graph : undirected graph  $G = (V, E)$  dimana  $V$  dapat di bagi menjadi 2 himpunan  $V_1$  dan  $V_2$  sehingga  $(u, v)$  menyebabkan baik  $u \in V_1$  dan  $v \in V_2$  atau  $u \in V_2$  dan  $v \in V_1$ . Sehingga seluruh edge ada diantara 2 himpunan  $V_1$  dan  $V_2$ .

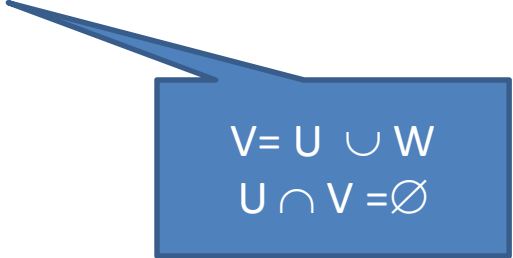
# Algoritma

## **ALGORITHM: BIPARTITE (G, S)**

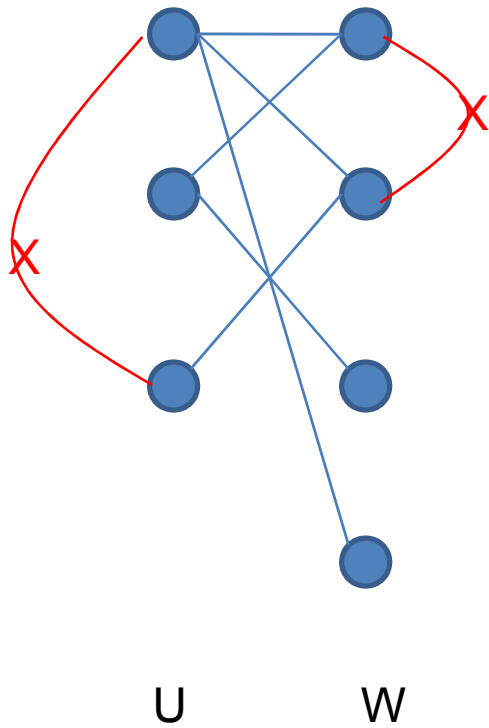
```
For each vertex  $U \in V[G] - \{s\}$  do
    Color[u] = WHITE
    d[u] =  $\infty$ 
    partition[u] = 0
Color[s] = gray
partition[s] = 1
d[s] = 0
Q = [s]
While Queue 'Q' is not empty do
    u = head [Q]
    for each v in Adj[u] do
        if partition [u] = partition [v] then
            return 0
        else
            if color[v] WHITE then
                color[v] = gray
                d[v] = d[u] + 1
                partition[v] = 3 - partition[u]
                ENQUEUE (Q, v)
DEQUEUE (Q)
Color[u] = BLACK
Return 1
```

# Bipartite Graph

- $G=(V,E)$  undirected graph
- $G$  adalah BG jika **ada** suatu **partisi** dari  $V$  ke dalam  $U,W$
- sedemikian rupa sehingga
- Setiap edge memiliki **sat**u end point dalam  $U$  dan lainnya dalam  $W$


$$V = U \cup W$$
$$U \cap W = \emptyset$$

# contoh



# Pencarian Mendalam

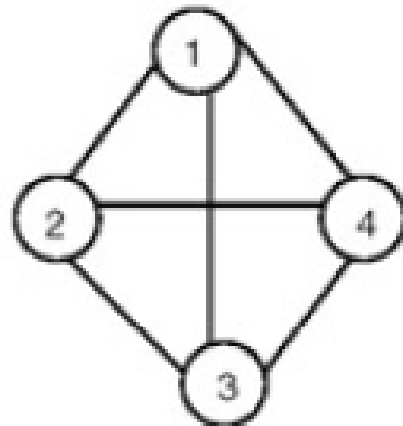
## *(Depth First Search* atau DFS).

- Traversal dimulai dari simpul  $v$ .
- Algoritma:
  - Kunjungi simpul  $v$ ,
  - Kunjungi simpul  $w$  yang bertetangga dengan simpul  $v$ .
  - Ulangi DFS mulai dari simpul  $w$ .
  - Ketika mencapai simpul  $u$  sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul  $w$  yang belum dikunjungi.
  - Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

# Representasi Array

- Salah satu representasi graph adalah dengan matrik  $n^2$  (matrik dengan  $n$  baris dan  $n$  kolom, artinya baris dan kolom berhubungan ke setiap vertex pada graph).
- Jika ada edge dari  $v_i$  ke  $v_j$  maka entri dalam matrik dengan index baris sebagai  $v_i$  dan index kolom sebagai  $v_j$  yang di set ke 1 ( $\text{adj}[v_i, v_j] = 1$ , jika  $(v_i, v_j)$  adalah suatu edge dari graph  $G$ ).
- Jika  $e$  adalah total jumlah edge dalam graph, maka ada entri  $2e$  yang di set ke 1, selama  $G$  adalah graph tak berarah.
- Jika  $G$  adalah graph berarah, hanya entri  $e$  yang di set ke-1 dalam matrik keterhubungan.

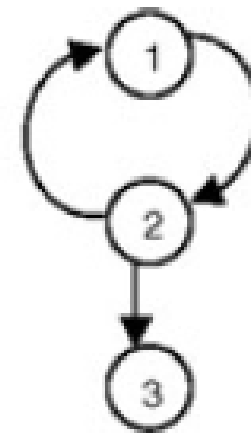
# Representasi Array



$G_1$

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

*Adjacency matrix of  $G_1$*



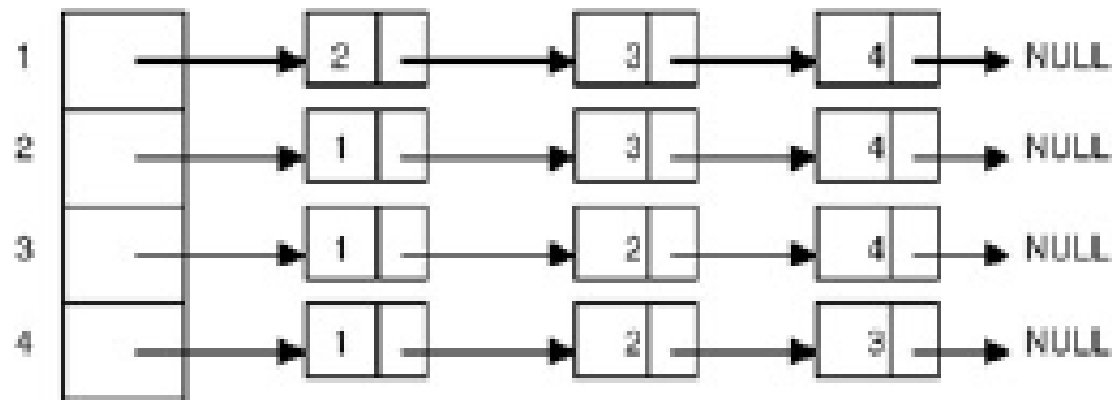
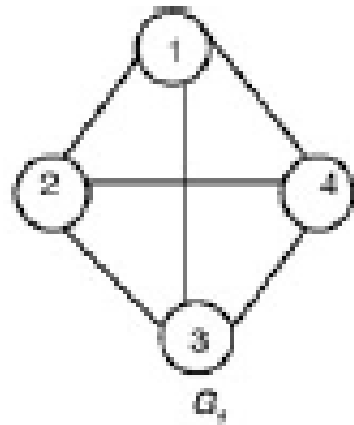
$G_2$

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

*Adjacency matrix of  $G_2$*



# Representasi Linked List

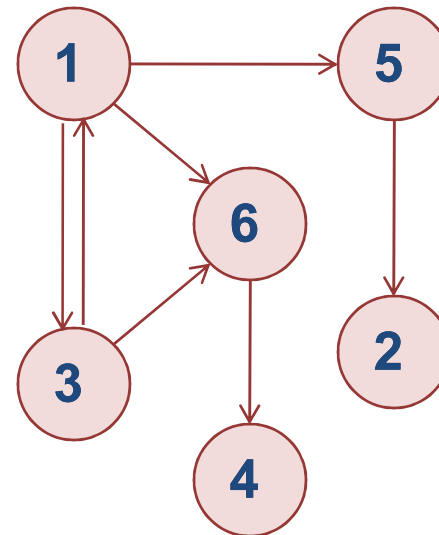


# Contoh Lain directed

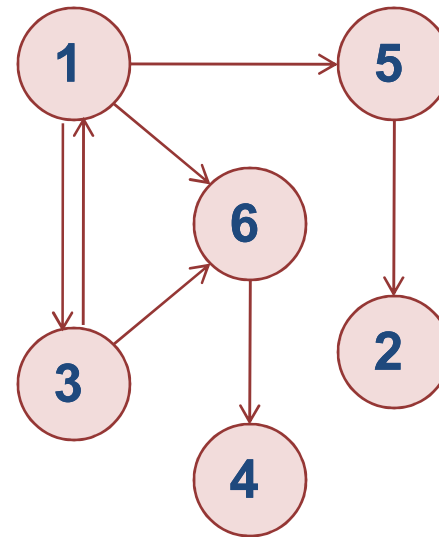
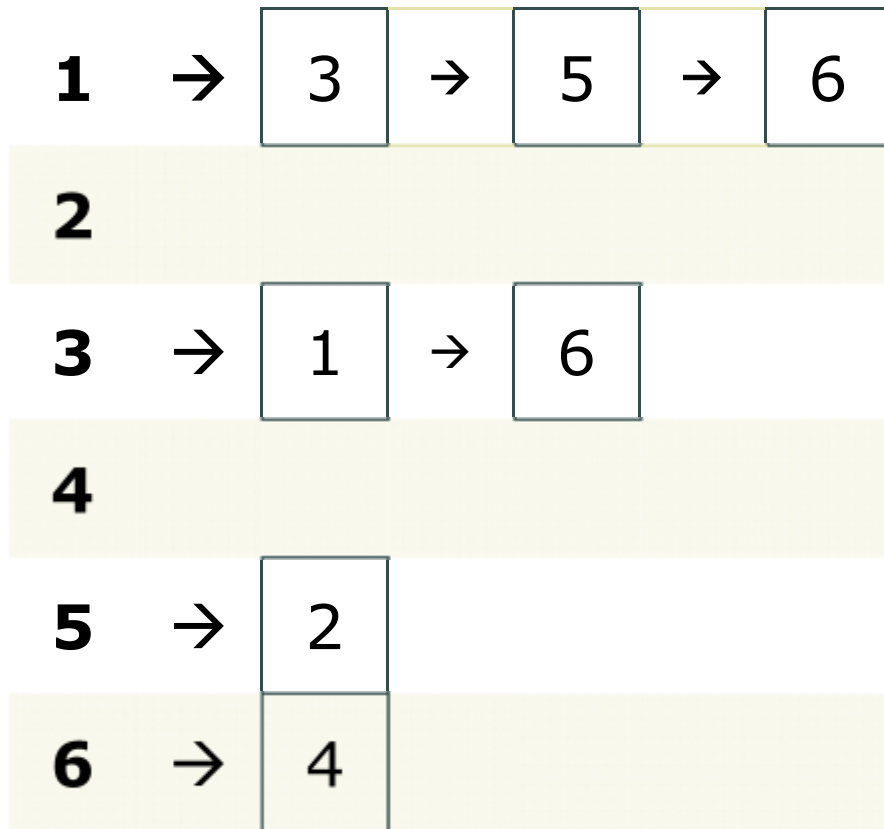
---

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>
<b>1</b>	0	0	1	0	1	1
<b>2</b>	0	0	0	0	0	0
<b>3</b>	1	0	0	0	0	1
<b>4</b>	0	0	0	0	0	0
<b>5</b>	0	1	0	0	0	0
<b>6</b>	0	0	0	1	0	0

---

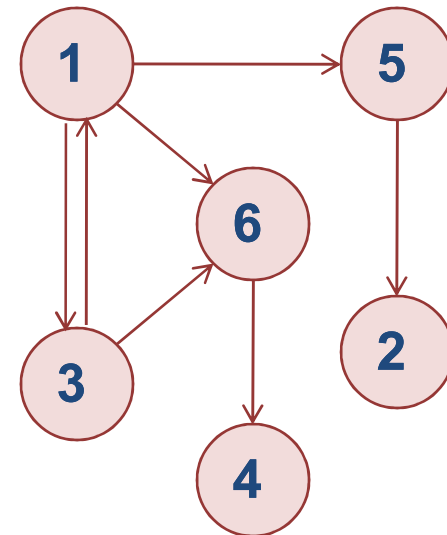


# Contoh Lain directed



# Contoh Lain directed

i	e[i][1]	e[i][2]	i	s[i]
<b>1</b>	1	3	<b>1</b>	1
<b>2</b>	1	5	<b>2</b>	4
<b>3</b>	1	6	<b>3</b>	4
<b>4</b>	3	1	<b>4</b>	6
<b>5</b>	3	6	<b>5</b>	6
<b>6</b>	5	2	<b>6</b>	7
<b>7</b>	6	4		

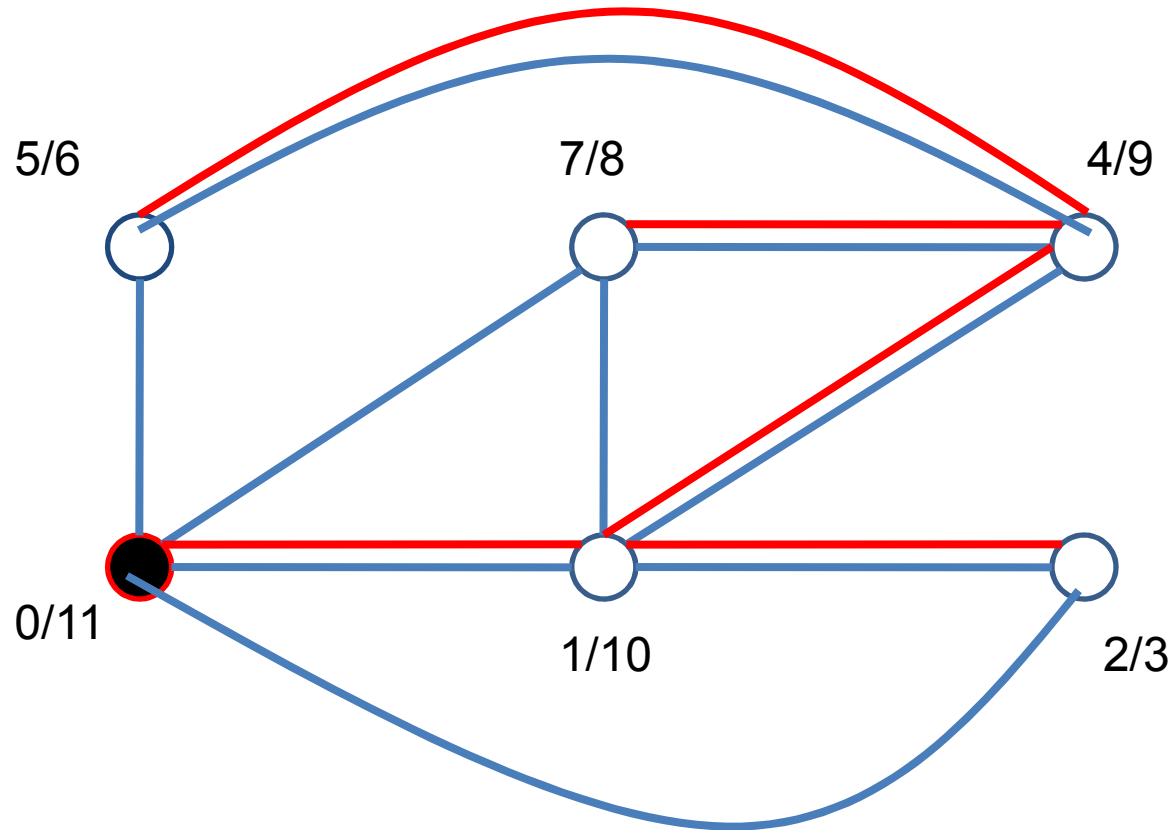


# Performa

	<b>Adjacency Matrix</b>	<b>Adjacency Linked List</b>	<b>Edge List</b>
<b>Memory Storage</b>	$O(V^2)$	$O(V+E)$	$O(V+E)$
<b>Check whether <math>(u,v)</math> is an edge</b>	$O(1)$	$O(\text{deg}(u))$	$O(\text{deg}(u))$
<b>Find all adjacent vertices of a vertex <math>u</math></b>	$O(V)$	$O(\text{deg}(u))$	$O(\text{deg}(u))$

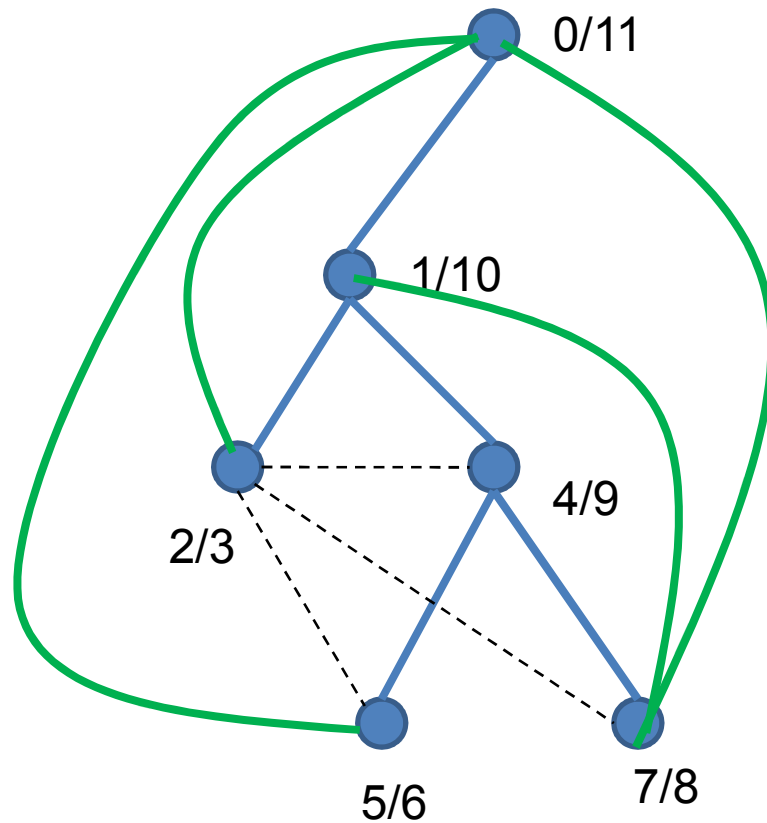
deg(u): # edge terhubung dg vertex  $u$

# DFS Graph



- Jika node ini putih maka explorasi node tersebut
- Jumlah edge merah sama dengan  $n-1$
- Edge merah membentuk subgraf terhubung
- Edge merah membentuk suatu tree (dfs tree)

# Dfs tree

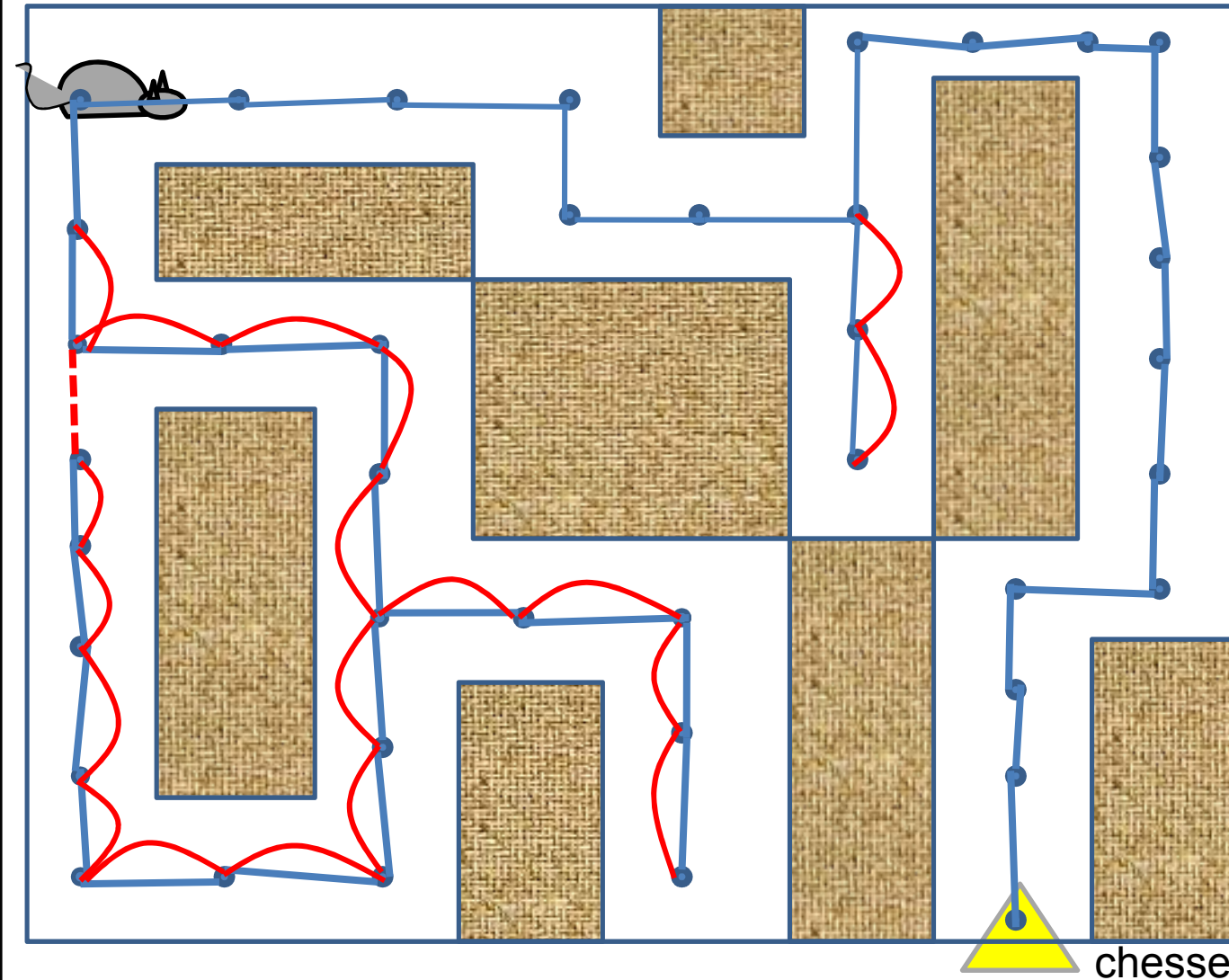


**Edge tree**

**Back tree** adalah  
**Suatu edge dari node**  
**ke ancestornya**

**DFS menentukan setiap edge sebagai suatu tree atau back tree**

# Depth First Search



**Properti tikus :**

Tahu arah

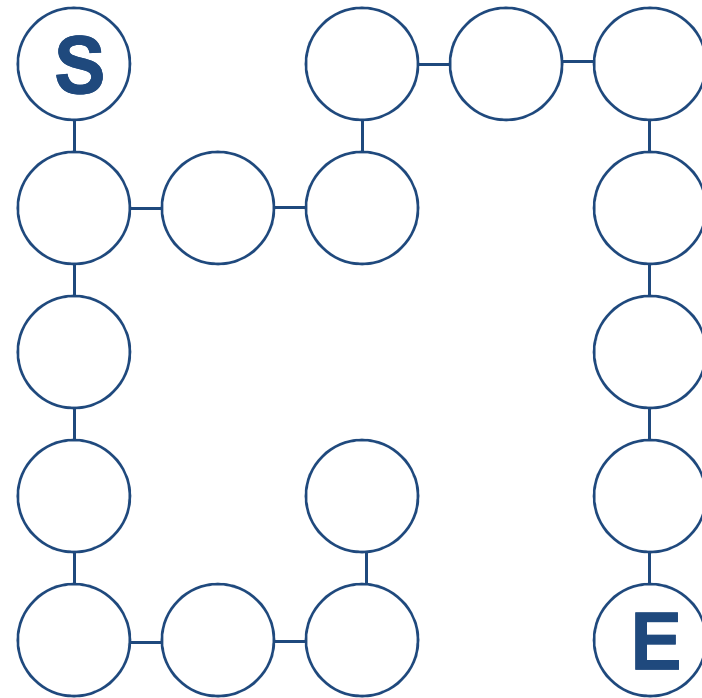
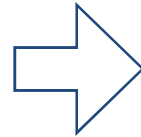
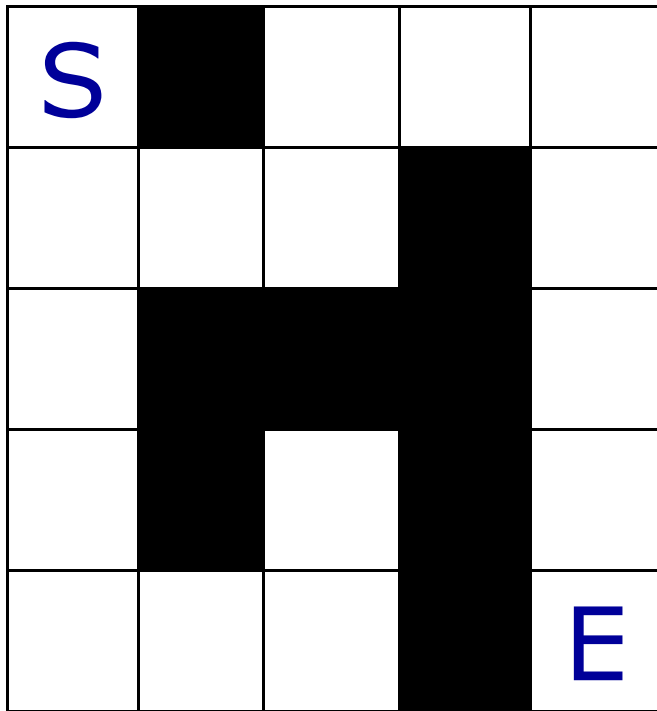
Memori node

**Backtrack :**

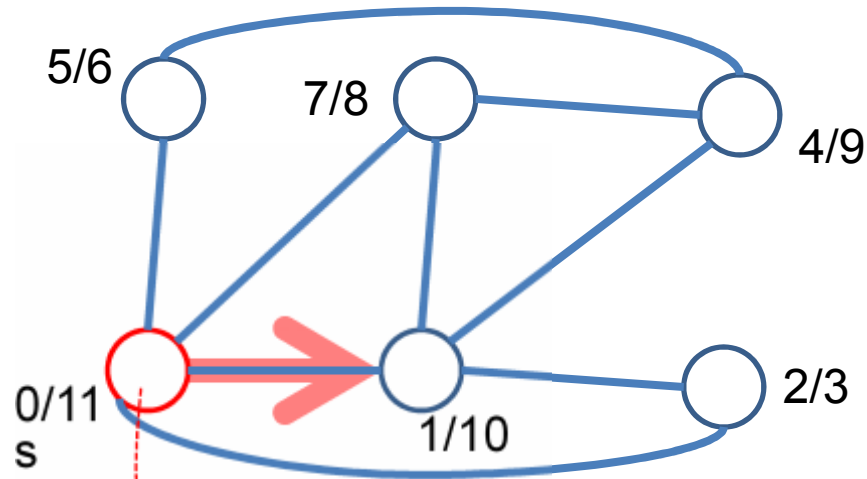
Kembali ke node  
sebelumnya yang  
sudah di kunjungi



# Model Graph



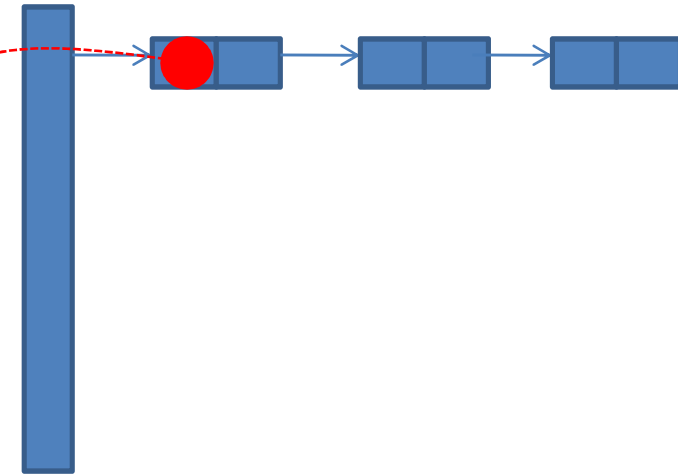
# DFS graph



Source node di labeli 0, lalu pilih node baru dan labeli 1, lalu pilih node baru labeli 2, pada node ini tidak ada lagi node yang dapat di kunjungi, kalau ke label 0 maka terjadi cycle dan sudah pernah di kunjungi. Maka kita lakukan backtrack, dan pada node yang baru di kunjungi ini kita naikan nilai label menjadi 3.

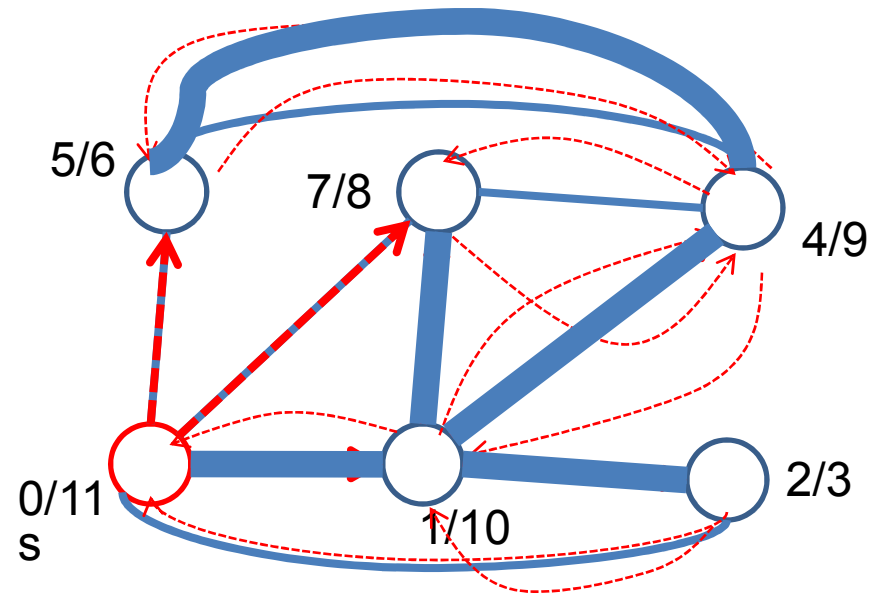
Setiap kali kita mengunjungi node, perlu di catat dan di naikan nilai labelnya

Representasi datanya memakai Adjacent list



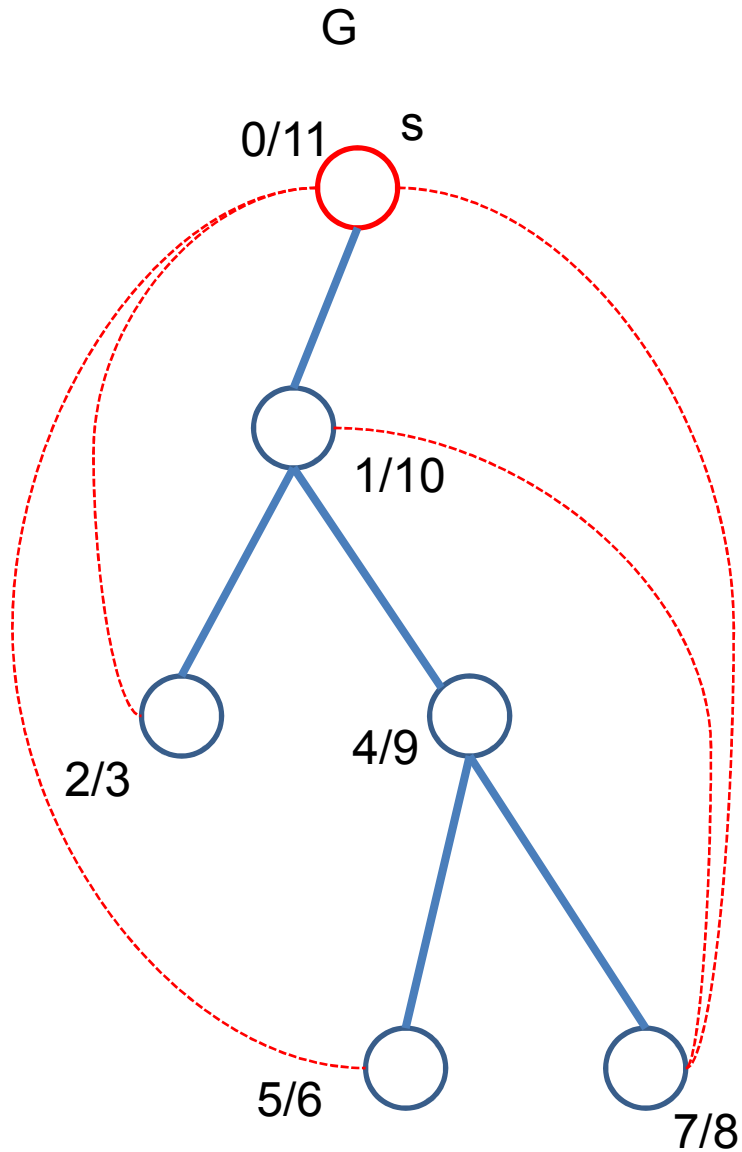
Jika node berwarna putih  
Kita akan kunjungi node tsb

# Contoh lain Dfs graph



Edge yang terbentuk dari dfs ini  
Adalah  $n-1$ ,  $n$  adalah node

# Dfs tree



Garis biru adalah **tree edge**

Garis merah putus adalah **back edge** ?

**back edge** : suatu edge dari node ke ancestor

**Jadi DFS, mengklasifikasikan setiap edge sebagai TREE atau BACK EDGE**

# Implementasi dfs rekursif

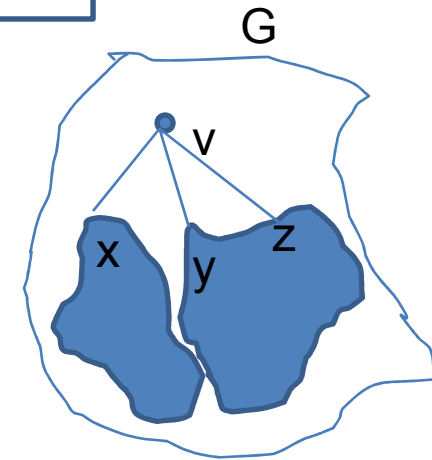
- Stack dan rekursi
- Buat array, visited=1, not visited=0



v 0/1

```
DFS (v) {  
    visited[v]=1; {source}  
    for all w adj. to v do  
        if !visited[w] then  
            DFS (w)  
}
```

Bagaimana kalo kita akan menghitung  
Kunjungan pada tiap node ?



Dfs(v)  
|  
dfs(x)  
|  
dfs(y)  
|

# Modifikasi dfs rekursif

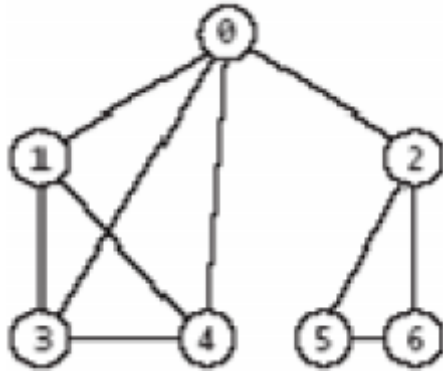
- Buat 2 array utk menandai setiap edge
  - **a**(arrival), untuk  $a[v]$  adalah saat kunjungan vertek
  - **d**(departure), dan  $d[v]$  adal saat meninggalkan vertek
  - **time** untuk counter tiap kunjungan dan saat meninggalkan vertek

```
a;d;time=0;  
DFS (v) {  
    visited[v]=1; {source}  
    a[v]=time++;  
    for all w adj. to v do  
        if !visited[w] then  
            DFS(w); (v,w)=tree edge  
    d[v]=time++;  
}
```

# Algoritma DFS iteratif

```
Dfs (v) {  
    P ← ∅  
    mark[v] ← visited; //boolean value  
    P ← Push (v);  
    While P ≠ ∅ do {  
        while (Ada vertek w yg mrp adj top P dan  
            mark[w] ≠ visited) do {  
            mark[w] ← visited;  
            P ← push(w);  
        }  
    }  
    pop (P);  
}
```

# Contoh dengan stack iteratif



Anak root kiri

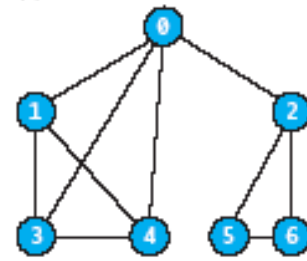
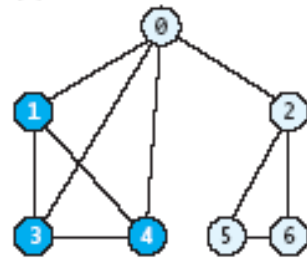
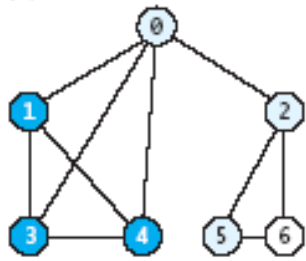
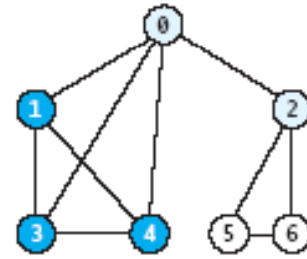
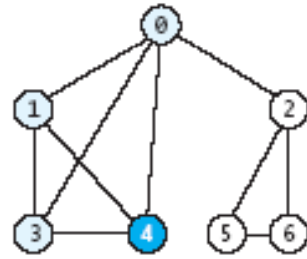
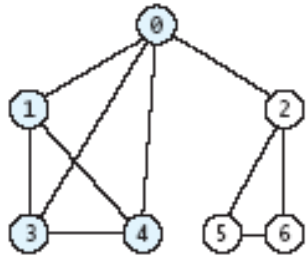
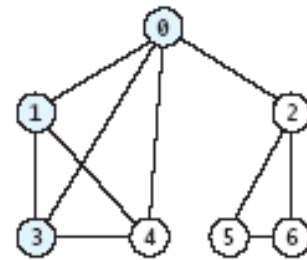
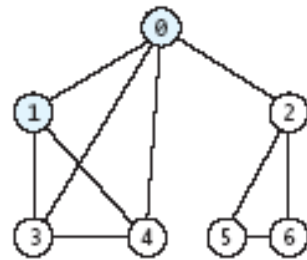
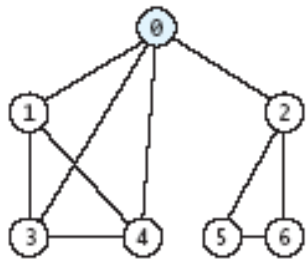
Anak root kanan

Stack yang terbentuk

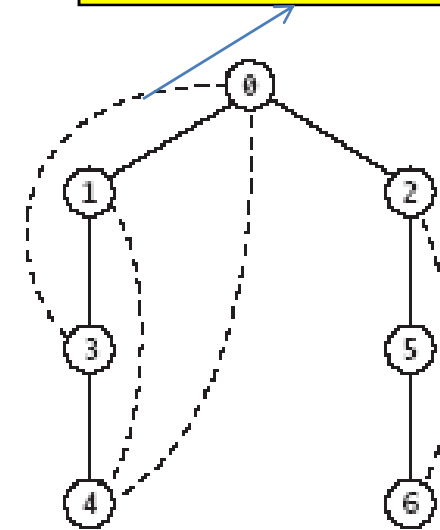
Operation	Adjacent Vertices	Discovery (Visit) Order	Finish Order
Visit 0	1, 2, 3, 4	0	
Visit 1	0, 3, 4	0, 1	
Visit 3	0, 1, 4	0, 1, 3	
Visit 4	0, 1, 3	0, 1, 3, 4	
Finish 4	Masuk stack		4 stack
Finish 3	Masuk stack		4, 3 stack
Finish 1	Masuk stack		4, 3, 1 stack
Visit 2	0, 5, 6	0, 1, 3, 4, 2	
Visit 5	2, 6	0, 1, 3, 4, 2, 5	
Visit 6	2, 5	0, 1, 3, 4, 2, 5, 6	
Finish 6	Masuk stack		4, 3, 1, 6
Finish 5	Masuk stack		4, 3, 1, 6, 5
Finish 2	Masuk stack		4, 3, 1, 6, 5, 2
Finish 0	Masuk stack		4, 3, 1, 6, 5, 2, 0



# Lanjutan



Garis putus  
adalah  
Back edge



Tree hasil DFS

# Algoritma DFS dg matrik

```
void buildadjm(int adj[][max], int n)
{
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) {
            printf("Masukan 1 jika ada edge
                    dari %d ke %d, kalau tidak
                    0 \n", i,j);
            scanf("%d",&adj[i][j]);
        }
}
```

# Algoritma DFS dg matrik

```
void dfs(int x,int visited[],
        int adj[][max],int n) {
    int j;
    visited[x] = 1;
    printf("Node yang di kunjungi %d\n",x);
    for(j=0;j<n;j++)
        if(adj[x][j] ==1 && visited[j] ==0)
            dfs(j,visited,adj,n);
}
```

# Contoh

**Input**

	0	1	2	3	4	5	6	7	8
0	0	1	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
2	0	1	0	0	0	0	1	0	0
3	0	1	0	0	1	1	0	0	0
4	1	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0	0	1
6	0	0	1	0	0	0	0	0	1
7	0	0	0	0	0	0	1	0	1
8	0	0	0	0	0	1	1	0	0

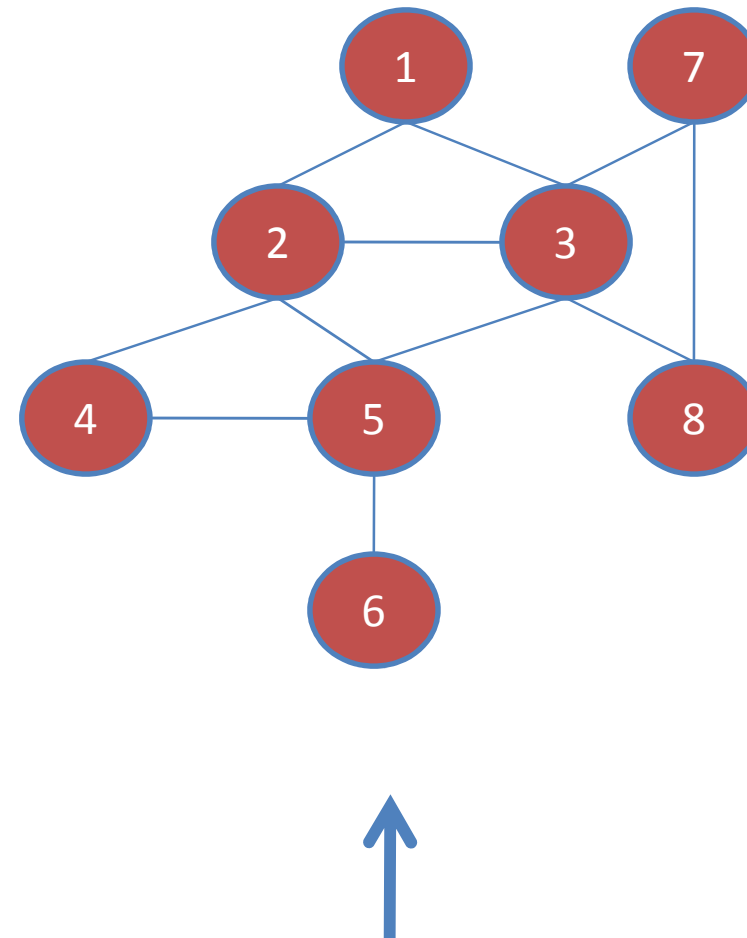
**Output**

0, 1, 2, 6, 8, 5, 3, 4, 7

[source](#) [simulasi](#)

# Gambarkan Graph yang mungkin dg algoritma DFS ?

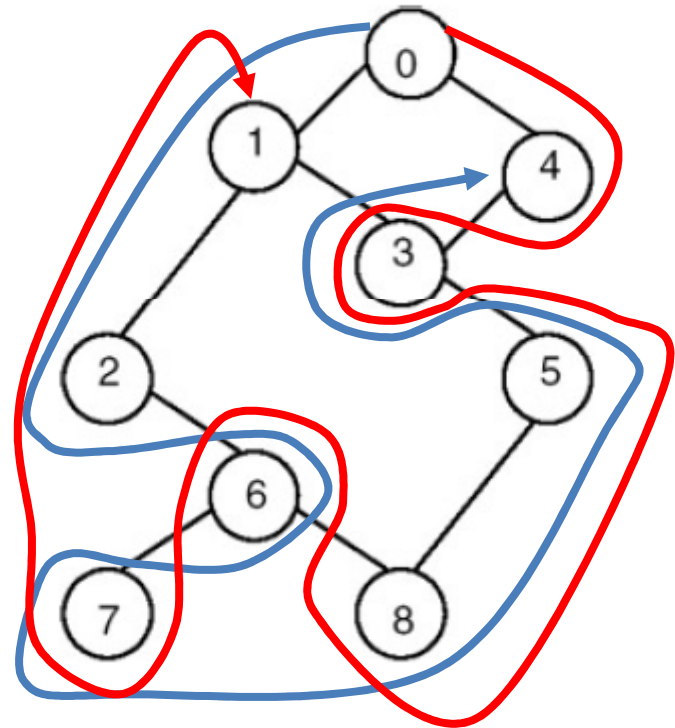
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



# Contoh Lain

- Kunjungan Awal vertex 0
- Hasilnya
  - 0 1 2 6 7 8 5 3 4
  - 0 4 3 5 8 6 7 2 1

Transformasikan ke dalam  
Matrik ?



# Analisis DFS

- **Jika** graph G di aplikasikan dengan depth-first search (dfs) yang di representasikan dengan list keterhubungan, maka vertek y yang berhubungan ke x dapat di tentukan dengan list keterhubungan dari setiap vertek yang berhubungan.
- Dengan demikian pencarian for loop untuk vertek keterhubungan memiliki total cost  $d_1 + d_2 + \dots + d_n$ , dimana  $d_i$  adalah derajat vertek  $v_i$ , karena jumlah node dalam list keterhubungan dari vertek  $v_i$  adalah  $d_i$ .

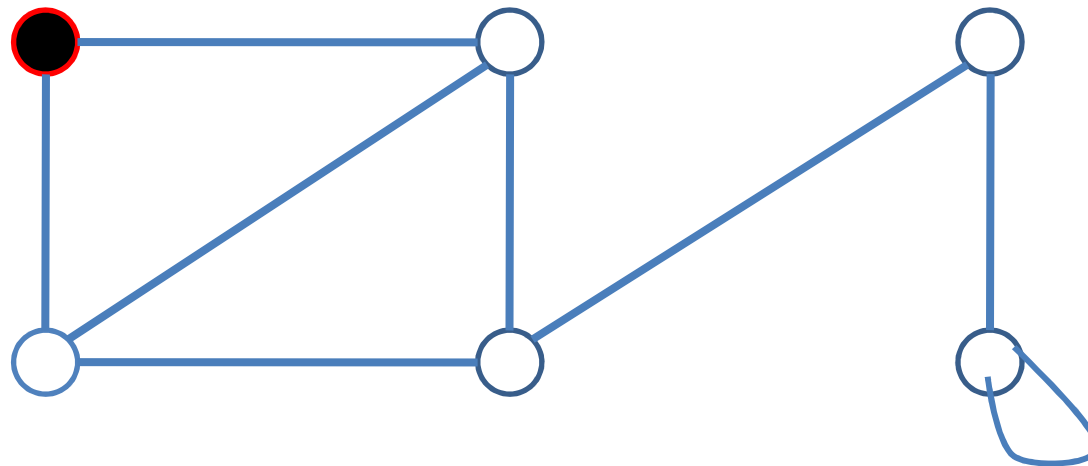
# Analisis DFS

- Jika graph  $G$  memiliki vertek  $n$  dan edge  $e$ , maka jumlah derajat tiap vertek ( $d_1 + d_2 + \dots + d_n$ ) adalah  $2e$ . Dengan demikian, ada total  $2e$  node list dalam list keterhubungan  $G$ . Jika  $G$  adalah directed graph, maka jumlah total  $e$  adalah node list saja.
- Waktu tempuh yang di perlukan untuk melakukan pencarian secara lengkap adalah  $O(e)$ , dengan  $n \leq e$ . Jika menggunakan matrik keterhubungan untuk merepresentasikan graph  $G$ , maka waktu tempuh untuk menentukan seluruh vertek adalah  $O(n)$ , dan karena seluruh vertek di kunjungi, maka total waktunya adalah  $O(n^2)$ .



# Soal

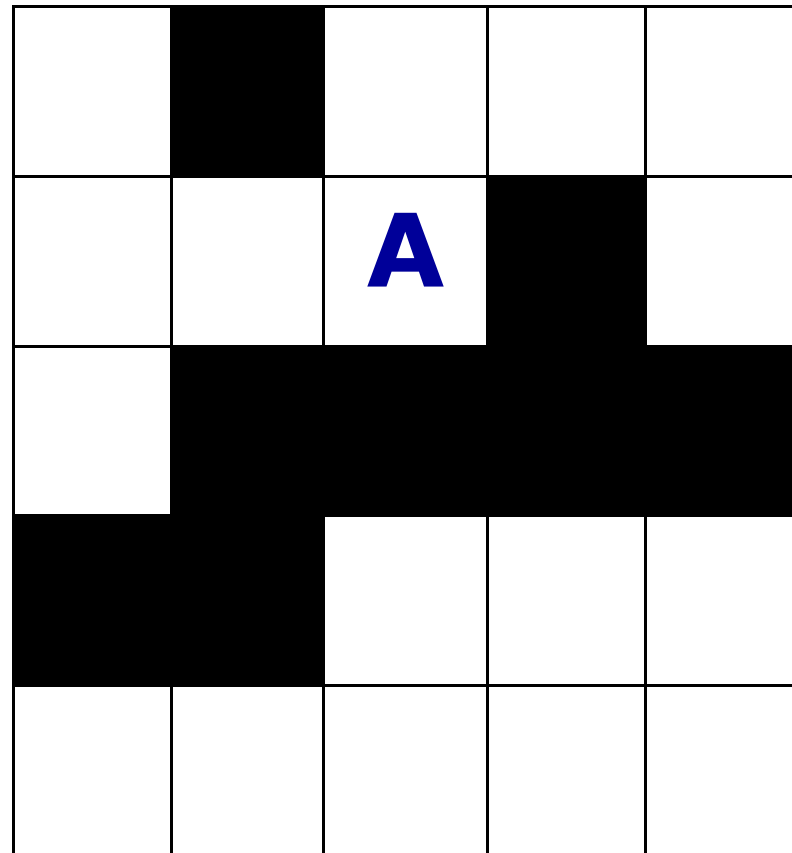
- Cari Edge Tree, Back Edge dan gambarkan dfs tree
- Labeli setiap edge/node berurutan abjad
- Buatlah matrik keterhubungannya



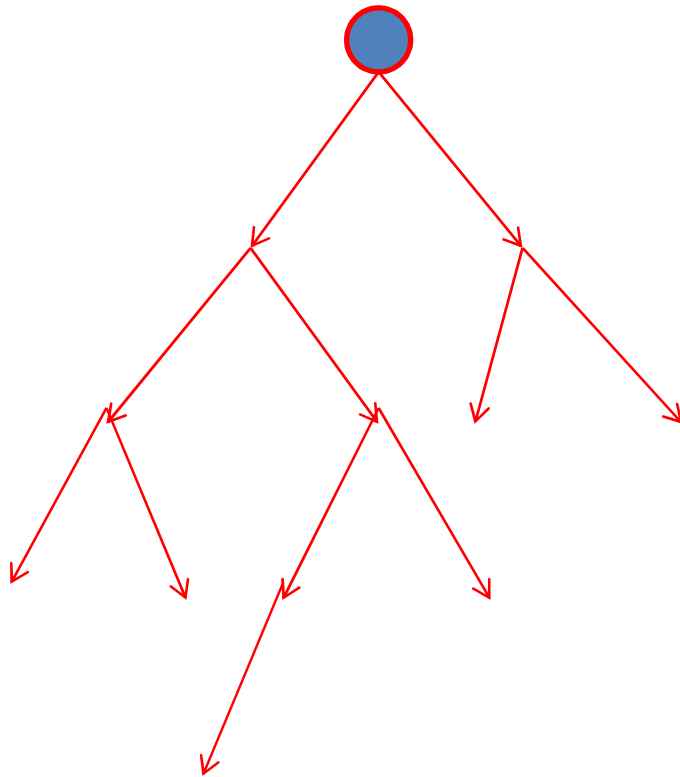
[solusi](#)

# Problem: Finding Area

- Cari area yang dapat ditemukan dari A.



# Aplikasi dfs pd directed graph strongly connected



1. If  $\text{dfs}(v)$  visit all vertek dalam  $G$ , then  $\exists$  path dari  $v$  ke setiap vertek dalam  $G$

2.  $\exists$  path dari setiap vertek dalam  $G$  ke  $v$

Misal ini benar

Jadi  $1+2 = \text{strongly connected}$ , kenapa ?

Karena jika ada vertek  $x, y$ , maka dari  $x \rightarrow v \rightarrow y$

(2)

(1) **Jad yang di perlukan adalah meyakinkan Ada suatu PATH dari  $v$  ke setiap vertek di  $G$**

# Bagaimana melihat bahwa ada path dari setiap vertek dari $v$ dalam $G$

$G$   $\xrightarrow{\text{Reverse edge}}$   $G^R$

Lakukan dfs( $v$ )

If seluruh vertek telah di kunjungi maka  
Mengakibatkan di dalam  $G$  ada path  
Dari setiap vertek ke  $v$

Procedurennya :

Ambil vertek dalam  $G$

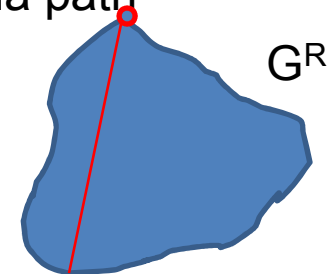
Lakukan dfs ( $v$ )

Reverse  $G$

Lakukan dfs( $v$ ) dalam  $G^R$

If seluruh vertek telah dikunjungi dengan dfs dalam  $G$  dan  $G^R$

Maka  $G$  adalah **strongly connected** else  $G$  bukan **strongly connected**



# Cara lain melihat strongly connected pada G

```
dfsSC(v) {
  a[v]=time++;visited[v]=1;
  mini=a[v];
  for all w adj to v do
    if !visited[w] then
      mini=min(mini,dfsSc(w))
    else mini=min(mini,a[w])
  if mini==a[v] then STOP {not strongly connected}
}
```

# Aplikasi dfs dan bfs dalam G

- strongly connected (dfs dir.)
- Memeriksa acyclic dalam G (dfs dir.)
- Topologi SORT (dfs dir.)
- 2 edge connectivity (dfs undir.)
- Connected component (bfs undir)
- Bipartite graph (bfs undir)
- Semuanya berada dalam order linear

# Tugas Simulasi

- Buat Simulasi Algoritma DFS dan BFS
- Hitung Order Fungsi dan kompleksitasnya
- Presentasikan