โครงสร้างข้อมูลกราฟและการสำรวจ

โดย

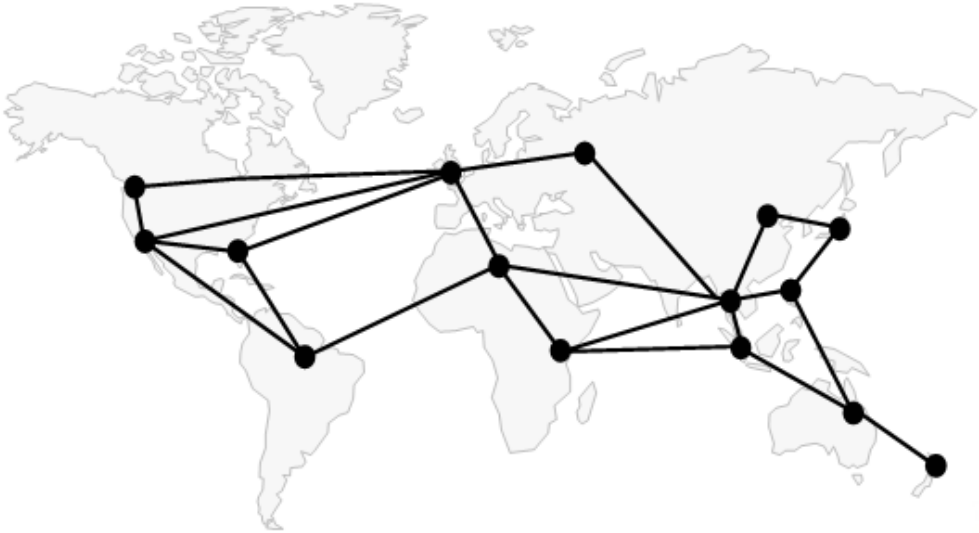
อ.ดร.ลือพล พิพานเมฆาภรณ์

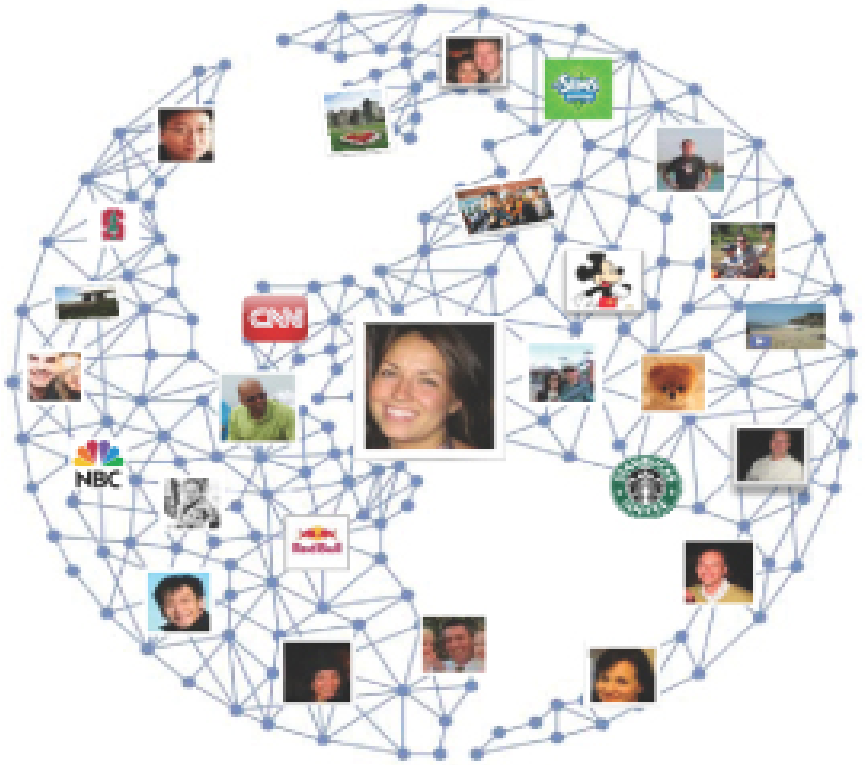
กราฟ (Graph)

เป็นโครงสร้างข้อมูลที่มีความสำคัญมากในทางคอมพิวเตอร์ เนื่องจากกราฟจะถูกใช้แสดงความสัมพันธ์ (relationship) ที่น่าสนใจระหว่างข้อมูลแล้ว กราฟยังสามารถที่จะโมเดลปัญหาที่มีขนาดใหญ่และซับซ้อน เพื่อแก้ปัญหาได้ง่ายขึ้น เช่น

1) ในระบบขนส่งมวลชน (transportation systems) เราอาจโมเดลการเดินทางของยานพาหนะ (vehicle) เพื่อหาเวลาในการเดินทางที่สั้นที่สุด หรือเส้นทางที่สั้นที่สุด โดยใช้กราฟ
2) ในอินเทอร์เน็ตและโซเชี่ยลเน็ตเวิร์ค (Social Network) กราฟสามารถแสดงความสัมพันธ์ระหว่างเว็บเพจหรือผู้ใช้คนอื่นๆ ได้

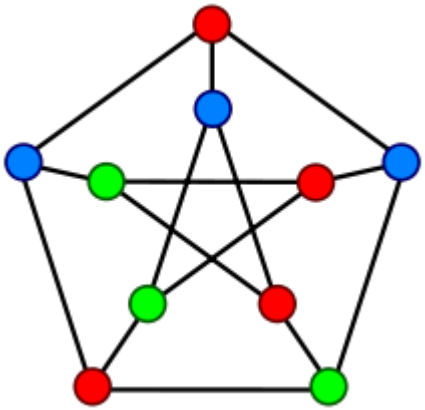
ตัวอย่างกราฟ

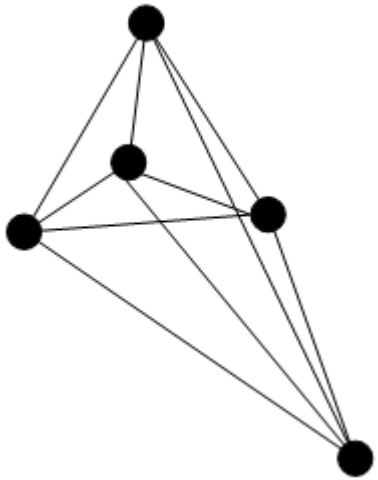




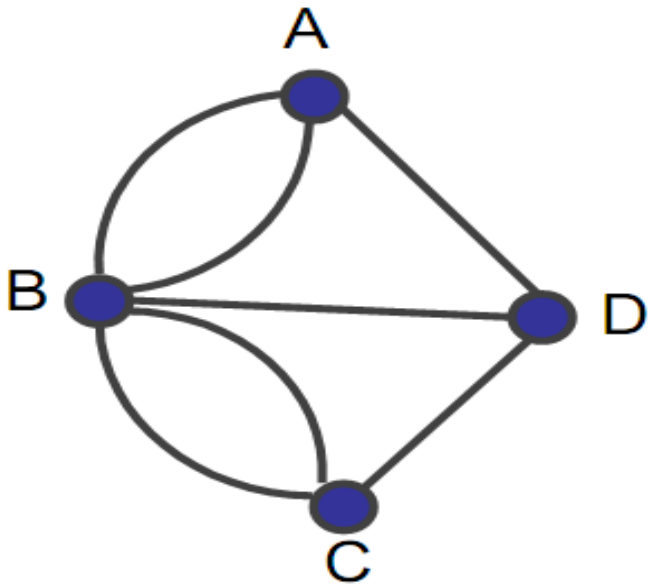
นิยามของกราฟ

- กราฟ G = (V, E) ถูกนิยามว่าเป็นเซตของเวอร์เท็กซ์ (vertex/vertices) และเอดจ์ (edge) โดยที่
  - เวอร์เท็กซ์ ทำหน้าที่แทน Object
  - เอดจ์ คือความสัมพันธ์ระหว่างเวอร์เท็กซ์



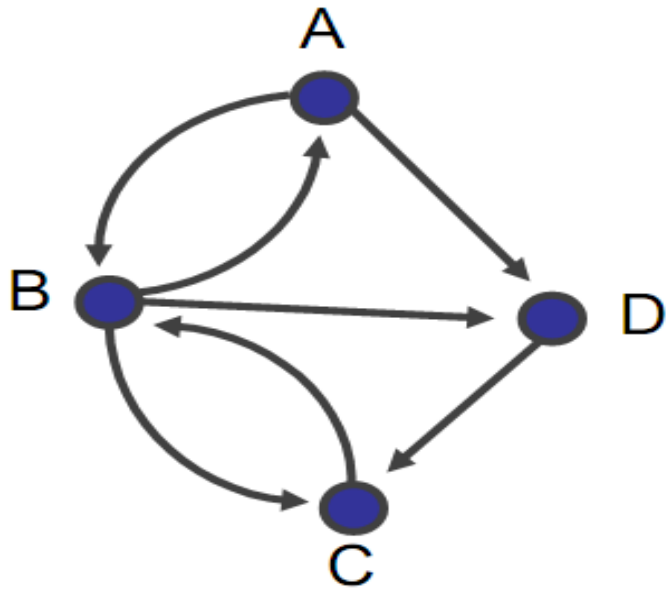


กราฟแบบมีทิศทาง และไม่มีทิศทาง (Undirected & Directed Graph)



เวอร์เท็กซ์: {A, B, C, D}

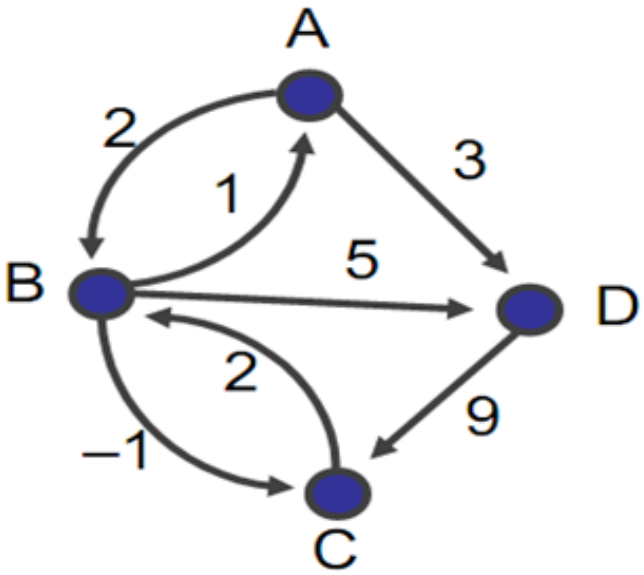
เอดจ์: {(A,B), (B,A), (B,D), (D,B), (C,D), (D,C)
(B,C), (C,B), (A,D), (D,A)}



เวอร์เท็กซ์: {A, B, C, D }

เอดจ์: {(A,B), (B,A), (A,D), (B,D), (D,C), (B,D),
(B,C), (C,B)}

กราฟแบบมีค่าน้ำหนัก (Weighted Graph)

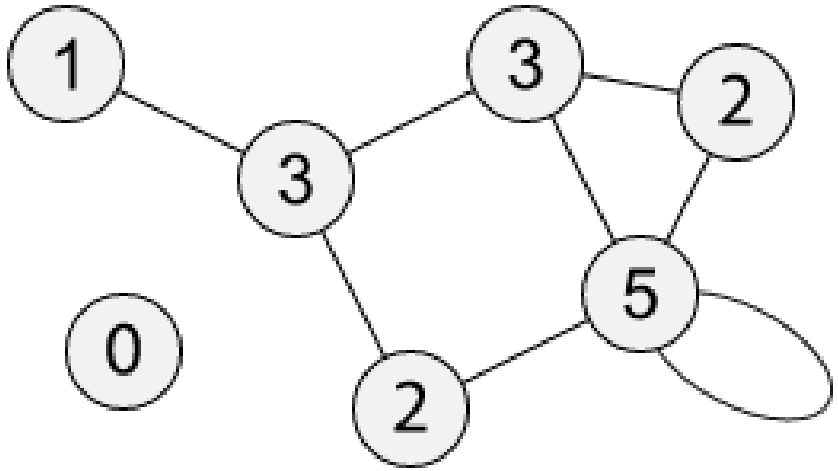


เวอร์เท็กซ์: {A, B, C, D }

เอดจ์: { (A, B, 2), (B, A, 1), (A, D, 3), (D, C, 9),
(B, D, 5), (B, C, -1), (C, B, 2) }

ดีกรี (Degree)

ดีกรีของเวอร์เทกซ์ v (degree(v)) คือจำนวนเอจด์ซึ่งร่วมกันกับ v



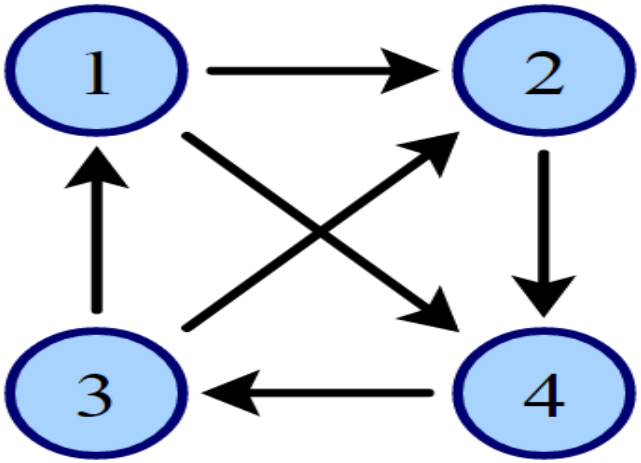
ดีกรีของเวอร์เท็กซ์ 0 degree(0) = 0

ดีกรีของเวอร์เท็กซ์ 1 degree(1) = 1

ดีกรีของเวอร์เท็กซ์ 5 degree(5) = 5

ดีกรีเข้าและดีกรีออก (In & Out Degree)

สำหรับกราฟแบบมีทิศทาง จะมีดีกรีอยู่ 2 ชนิด คือ In-degree และ Out-degree



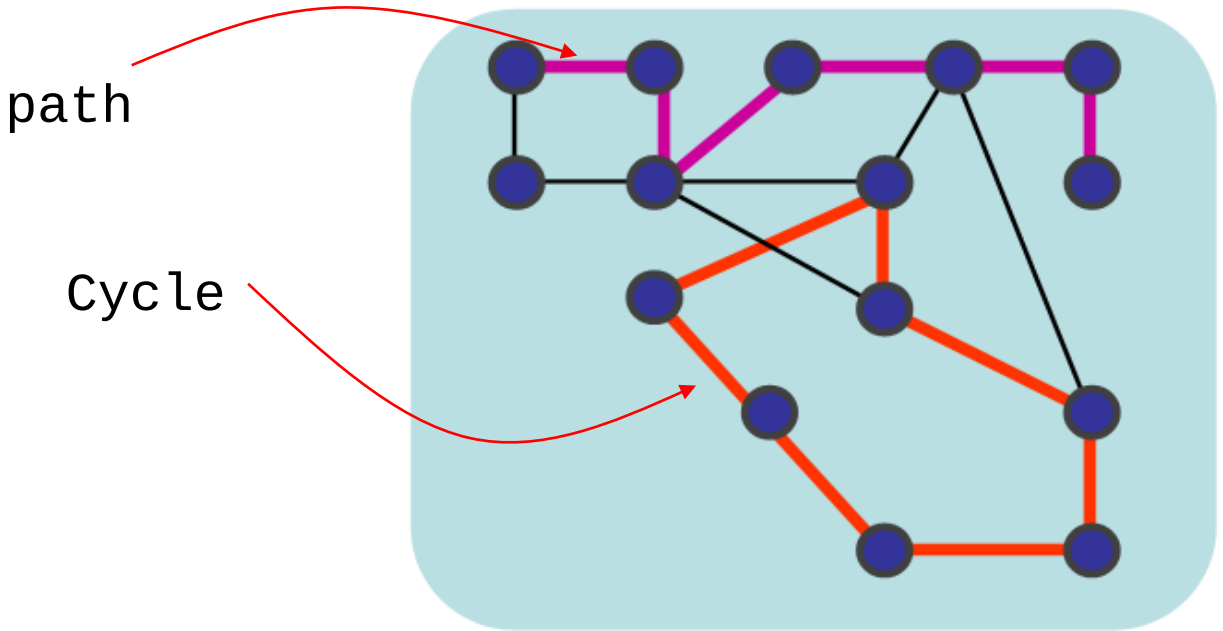
เวอร์เท็กซ์ 1 มี **in-degree** เท่ากับ 1 แต่มี **out-degree** เท่ากับ 2

เวอร์เท็กซ์ 2 มี **in-degree** เท่ากับ 2 แต่มี **out-degree** เท่ากับ 1

$$\deg(v) = \deg^{-}(v) + \deg^{+}(v)$$

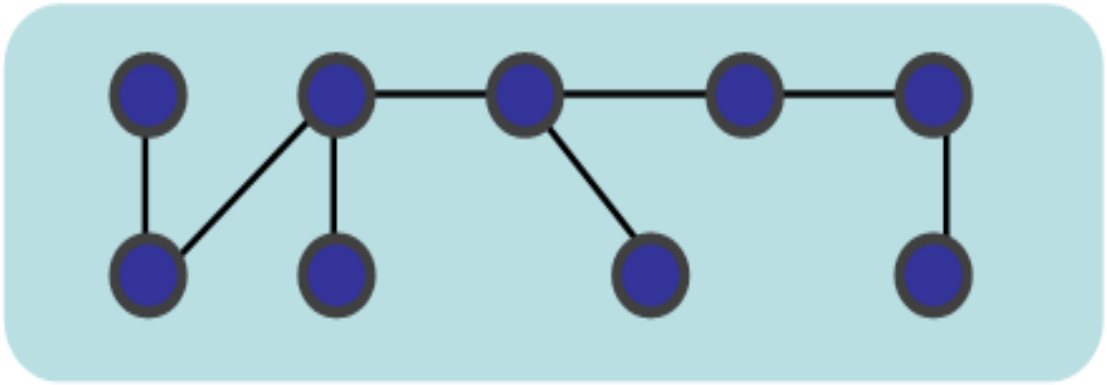
พาสและวงวน (Path & Cycle)

- Path คือลำดับ (Sequence) ของ vertices ที่เชื่อมด้วยเอดจ์เดียวกัน
- Cycle คือ path ลากผ่าน vertices โดยมีโหนดแรกและโหนดสุดท้ายเป็นโหนดเดียวกัน

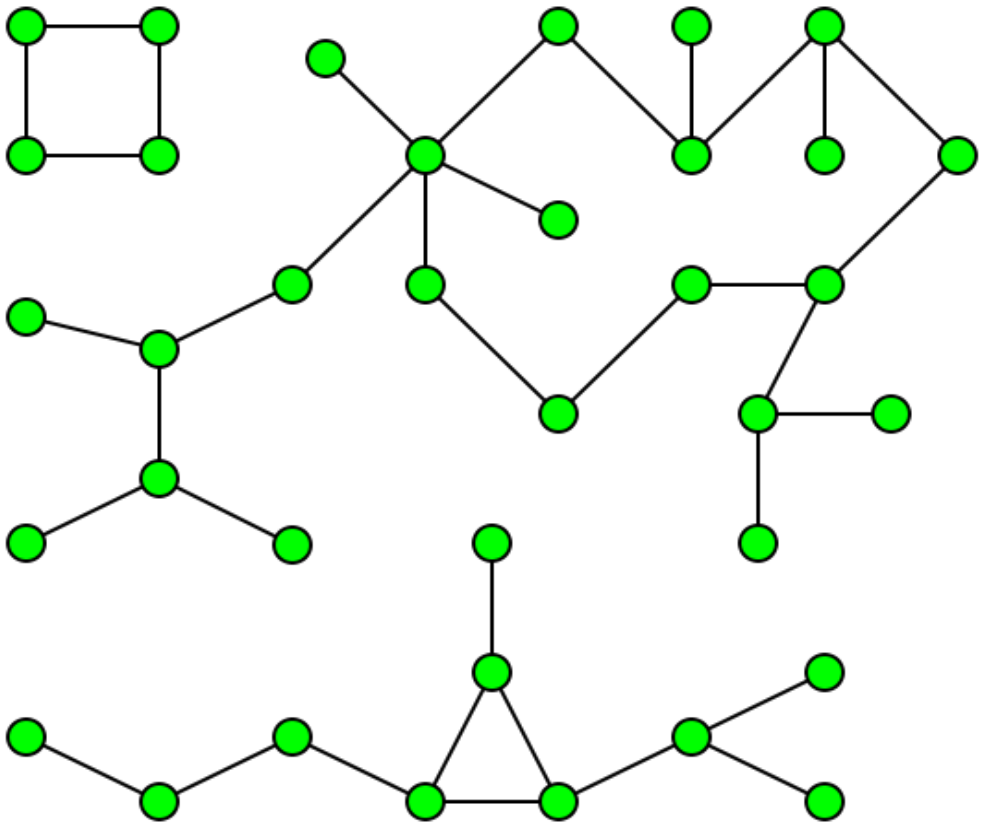


Acyclic Graph

- คือกราฟที่ไม่มีวงวน (cycle) ใดๆ ปรากฏขึ้น บางครั้งอาจเรียกว่าเป็น spanning tree
- Spanning tree คือกราฟย่อยที่มี edge เชื่อมกันกับทุก vertex ในกราฟ



Connected components in a graph



หมายถึงเซตย่อยของเวอร์เท็กซ์ในกราฟ ซึ่งเชื่อมต่อซึ่งกันและกัน

บางครั้งอาจเรียกว่าเป็นกราฟย่อย (sub-graph)

- กราฟโดยทั่วไปจะมีเพียง 1 component ซึ่งหมายความว่าเราสามารถเข้าถึงทุกๆ เวอร์เท็กซ์ในกราฟได้โดยผ่านการหาพาสในกราฟ

โครงสร้างข้อมูลสำหรับกราฟ

- เมตริกซ์ประชิด (Adjacency Matrix)
  - มักใช้อาร์เรย์ 2 มิติ แทนเวอร์เท็กซ์ที่เชื่อมต่อกันด้วยเอดจ์
  - ข้อดีคือจัดการง่ายและทำงานได้รวดเร็ว
  - ข้อเสียคือสิ้นเปลืองหน่วยความจำ โดยเฉพาะอย่างยิ่ง sparse graph

- ลิสต์ประชิด (Adjacency List)
  - ใช้ลิสต์ร่วมกับอาร์เรย์ขนาด 1 มิติเพื่อเก็บเอจด์ระหว่างเวอร์เท็กซ์
  - ข้อดีคือประหยัดหน่วยความจำที่ใช้ในการแสดงกราฟขนาดใหญ่
  - ข้อเสียคือยุ่งยากในการจัดการกว่าเมตริกซ์ประชิด

เมตริกประชิดสำหรับกราฟ

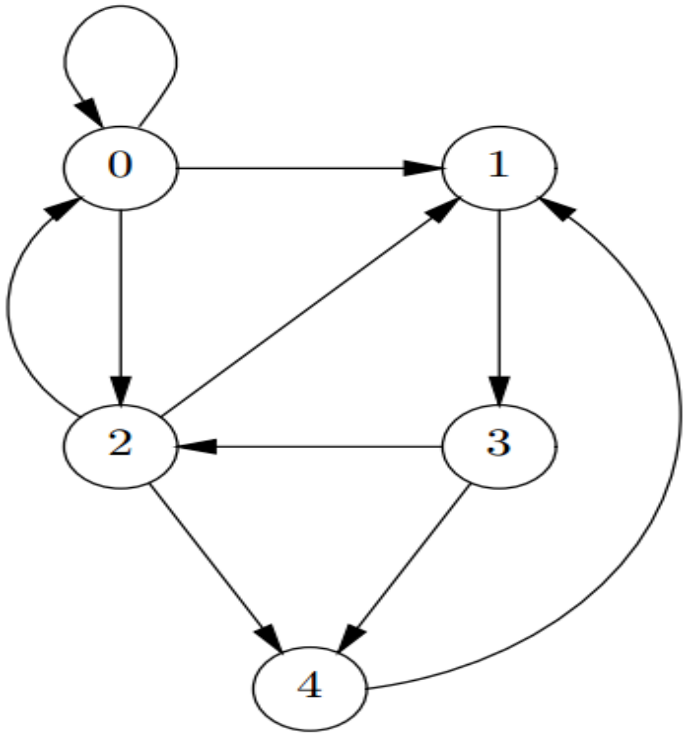
```
#define MAXV 5

typedef struct  {
   int edges[MAXV][MAXV];
   int degree[MAV];
   int nvertices;
   int nedges;
 } graph;
```

เวอร์เท็กซ์

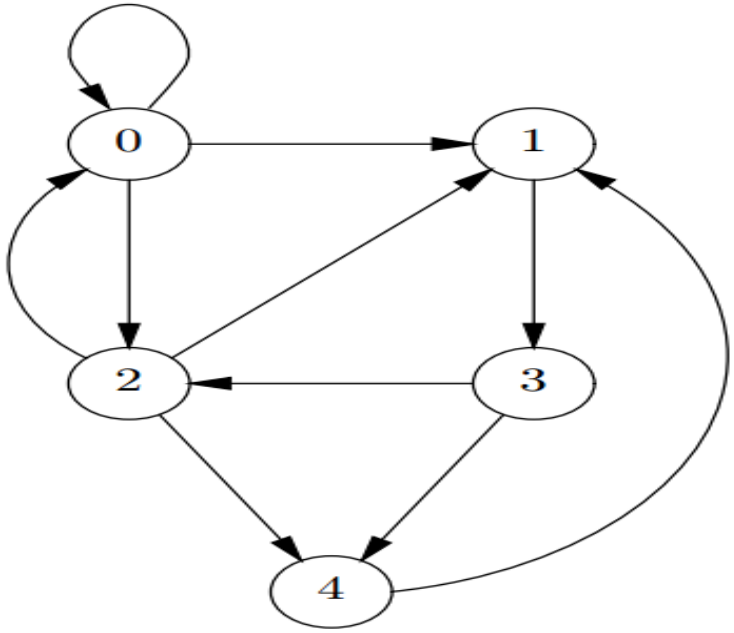
	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

เวอร์เท็กซ์

edges[][]



```
#include<stdio.h>
#define MAXV 5
typedef struct  {
   int edges[MAXV][MAXV];
   int degree[MAV];
   int nvertices, nedges;
 } graph;

main()
{  graph g;
   g.edges[0][0] = 1;    g.edges[0][1] = 1;
   g.edges[0][2] = 1;    g.edges[0][3] = 0;
   g.edges[0][4] = 0;

   g.edges[1][0] = 0;    g.edges[1][1] = 0;
   g.edges[1][2] = 0;    g.edges[1][3] = 1;
   g.edges[1][4] = 0;
                 ……..
}
```



```
initial_graph(graph *g)
{  g->nvertrices = 0;
   g->nedges = 0;
   for(int i=0;i< MAXV;i++)
      g->degree[i] = 0;
}
```

การกำหนดค่าเริ่มต้นให้กับกราฟ

```
read_graph(graph *g)
{  int i, m, x, y;
   initial_graph(g);

   scanf(“%d %d”,&(g->nvertrices),&m);
   for(i=0; i< m ; i++)
    { scanf(“%d %d”,&x, &y);
      g->edges[x][y] = 1;
      g->nedges++;
      g->degree[x]++;
      g->degree[y]++;
    }
}
```

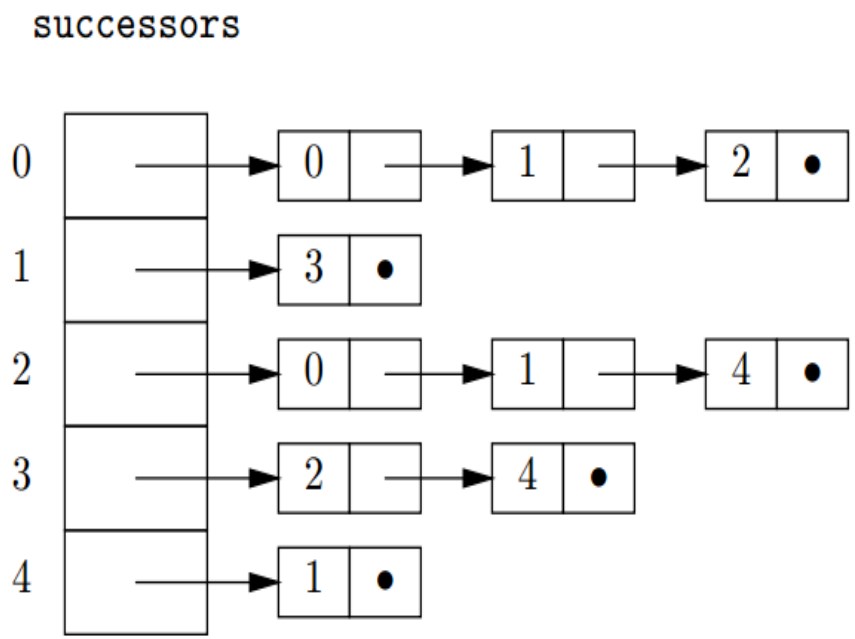
การอ่านข้อมูลเพื่อสร้างกราฟ โดยเริ่มต้นจากรับค่าจำนวนเวอร์เท็กซ์และจำนวนเอดจ์ จากนั้นอ่านเข้าทีละเอดจ์ เพื่อสร้าง adjacency matrix

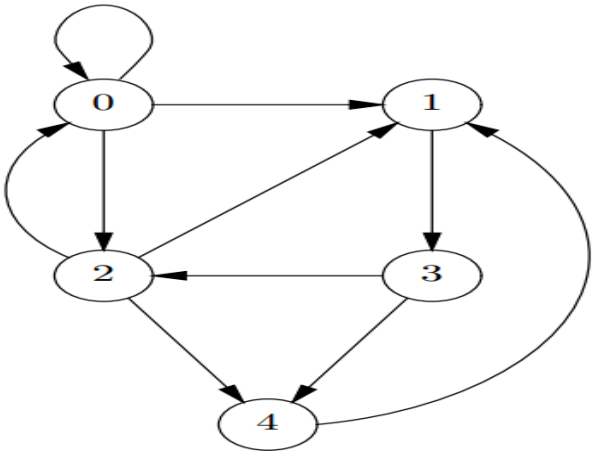
การเข้าถึงข้อมูลในกราฟโดยเมตริกซ์ประชิด

```
print_graph(graph *g)
{ int i, j;
  for(i=0; i < g->nvertrices; i++)
  {  printf("%d :", i);
     for(j=0; j<= g->nvertrices; j++)
        if(g->edges[i][j] == 1)
           printf(" %d", j);
     printf("\n");
  }
}
```

```
0 : 0 1 2
1 : 3
2 : 0 1 4
3 : 2 4
4 : 1
```

ลิสต์ประชิด






```
struct node {
int item;
struct node* next;
};

struct adj_list{
 struct node *head;
};

struct graph {
   int nvertices;
   struct adj_list array[MAXV];
}
```

An implementation of adjacency list

```
struct Graph* createGraph(int n)
{ struct Graph* graph = (struct Graph *)
   malloc(sizeof(struct Graph));

    for(int i = 0; i < n; ++i)
        graph->array[i].head = NULL;
  return graph;
}
```

```
void addEdge(struct Graph* graph, int src, int dest)
{ struct Node* newnode = CreateNode(dest);
   newnode->next = graph->array[src].head;
   graph->array[src].head = newnode;
}
```

An implementation of adjacency list

```
struct Node* CreateNode(int dest)
{ struct Node *newnode = (struct Node*)malloc(sizeof(struct
   Node));
    newnode->item = dest;
    newnode->next = NULL;
return newnode;
}
```

การเข้าถึงข้อมูลใน adjacency list

```
void printGraph(struct Graph *g)
{ int v;
  for(v=0; v < graph->nverterces ++v) {
    struct Node* p = graph->array[v].head;
   while(p)
    {  print(“-> %d”, p->item);
       p = p->next;
    }
    printf(“\n”);
  }
}
```

การเข้าถึงข้อมูลใน adjacency list

```
int main()
{ int V = 5;

  struct Graph *g = createGraph(V);
  addEdge(g, 0,1);
  addEdge(g, 0,4);
  addEdge(g, 1,2);
  addEdge(g, 2,3);
  addEdge(g, 3,4);

  printGraph(g);
}
```

การสำรวจกราฟ (Graph Traversal)

- ปัญหาพื้นฐานที่สำคัญที่สุดของกราฟ คือทำอย่างไรจึงจะเข้าถึงแต่ละโหนดในกราฟ หรือที่รู้จักกันว่าเป็นการเยี่ยมโหนด

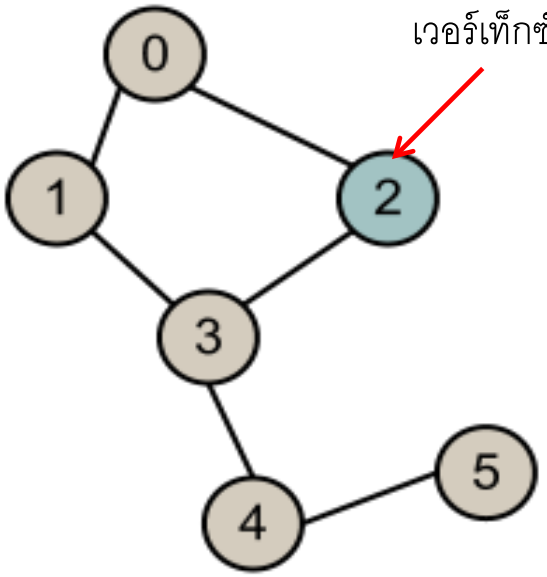
- 2 เทคนิคที่นิยมใช้ในการสำรวจกราฟ
  - การค้นหาแนวกว้าง (Breadth First Search : BFS)
  - การค้นหาแนวลึก (Depth First Search :DFS)

Breadth-First Search (BFS)

- กำหนดเวอร์เท็กซ์เริ่มต้น (source vertex)
- สร้างคิว (queue) เพื่อสำรวจกราฟ โดยจะเก็บ neighbor nodes ของ source vertex จากนั้นค่อยๆ ขยายเส้นทางสำรวจจากเวอร์เท็กซ์ซึ่งถูกเยี่ยมแล้ว (visited vertices) นำไปใส่ไว้ในคิว
- BFS จะจบการทำงานเมื่อทุกโหนดในกราฟถูกเยี่ยม หรือไม่มีเวอร์เท็กซ์ในคิว

ตัวอย่าง BFS

Breadth First Search




```
1. q = {}
2. q = {2}
3. q = {0, 3}
4. q = {1}
5. q = {4}
6. q = {5}
```

Using a queue

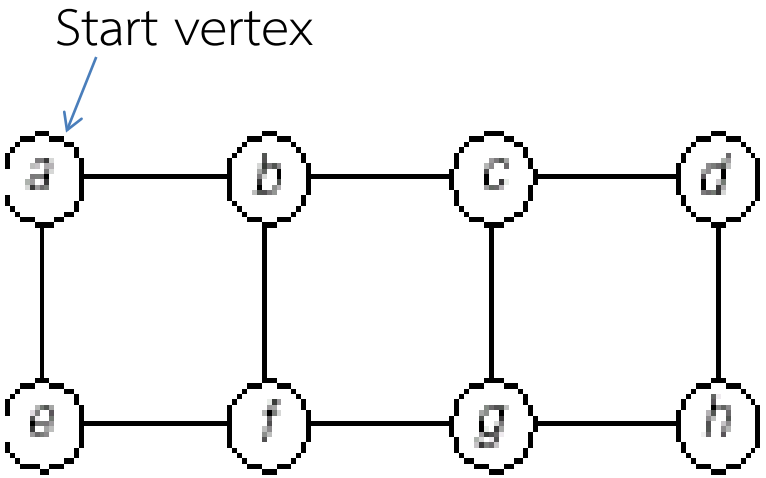
BFS algorithm

```
void BFS (graph *g, int start)
 {   queue Q;                              // queue data structure
      initial_queue(&Q);                     // make empty queue
      enqueue(&Q, start);                  // enqueue with start node
    while(empty(&Q)== FALSE)
     {  v = dequeue(&Q);                    // call dequeue
        printf("visit node : %d", v);     // print visit node
        visited[v] = TRUE;                  // mark visited v
        for(i=0;i < g->nvertrices; i++)     // explore neighbors
          { if(visited[i] == FALSE)
            { if(g->edges[i][v]== 1 || g->edges[v][i] == 1)
                  enqueue(&Q, i);           // next explore
            }
          }
     }
 }
```

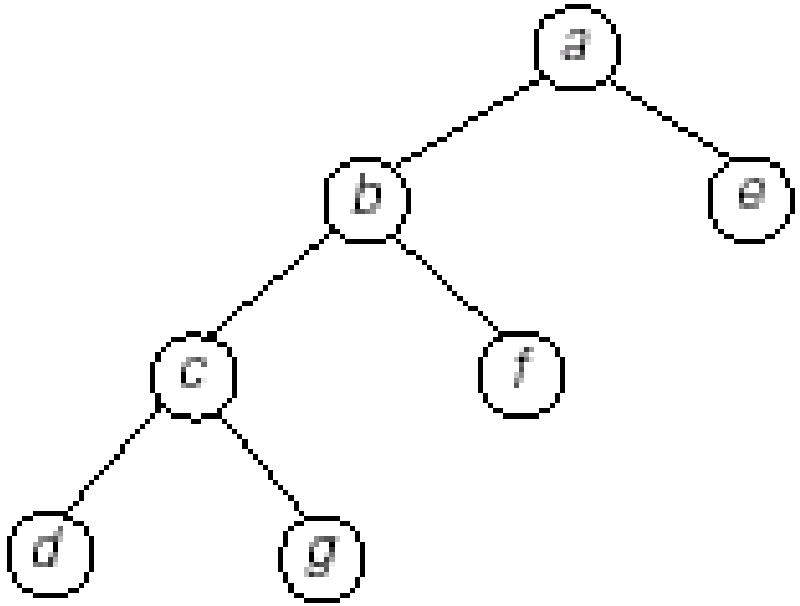
แอพพลิเคชันของ BFS

- ตรวจสอบ Connectivity ของกราฟ :
  - เนื่องจาก BFS สิ้นสุดการทำงานเมื่อทุกๆ vertex ที่เชื่อมต่อกับ vertex ตั้งต้นได้รับการพิจารณา การตรวจสอบ connectivity สามารถทำได้โดยใช้ BFS traversal ซึ่งเริ่มจาก vertex ใดก็ได้ เมื่อ algorithm สิ้นสุดการทำงานให้ตรวจสอบว่า ทุกๆ vertex ใน graph ได้รับการพิจารณาแล้วหรือไม่ graph จะถือว่า connected ถ้าทุกๆ vertex ถูก visited
- ค้นหาพาสที่สั้นที่สุด
  - สำรวจต้นไม้ BFS จะทำให้ได้ Path ที่มีความยาวที่สั้นที่สุดจาก source vertex ไปยังทุกๆ vertex

Finding Minimum-edge Path



กราฟ G

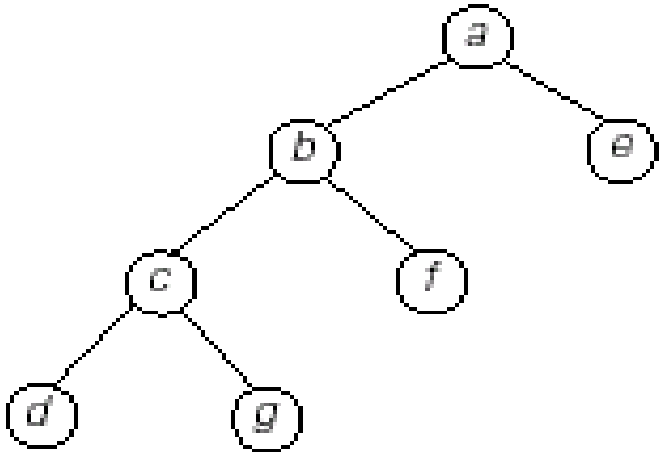


ต้นไม้ BFS

Shortest Path a->b = 1, a->e = 1, a->c=2, a->d=3, a->g=3

ทำอย่างไรจึงจะจัดเก็บต้นไม้ BFS

- ต้นไม้ BFS ที่เกิดจากลำดับในการสำรวจกราฟ มีความสำคัญเช่น เช่น การค้นหา path ที่สั้นที่สุดจากเวอร์เท็กซ์เริ่มต้นในกราฟ เราจะมีวิธีการจัดเก็บมันได้อย่างไร



	0	1	2	3	4	5	6
node[]	a	b	c	d	e	f	g

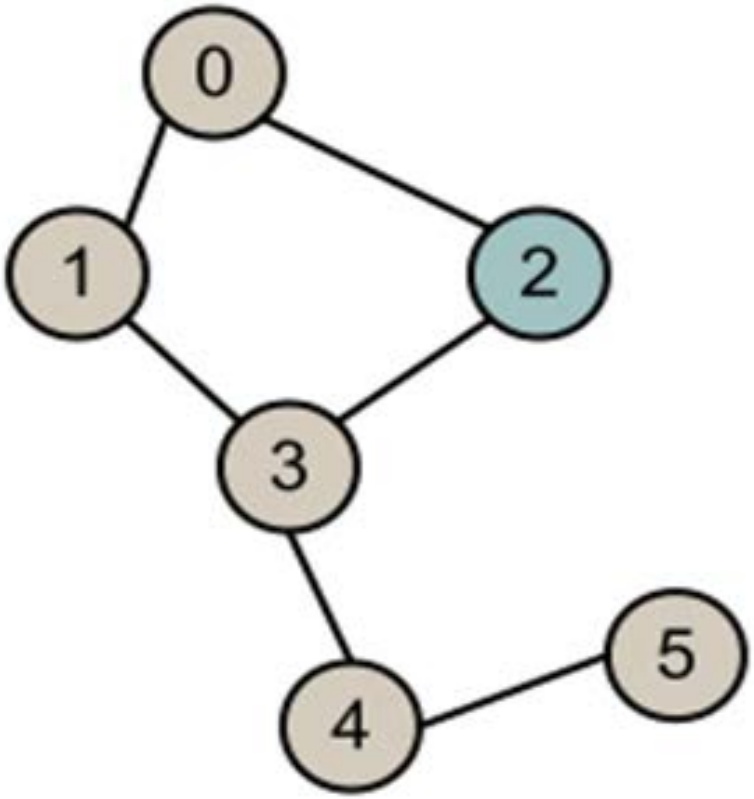
	0	1	2	3	4	5	6
parent[]	-1	0	1	2	0	1	2

```
void BFS (graph *g, int start)
 {   queue Q;                              // queue data structure
      initial_queue(&Q);                     // make empty queue
      enqueue(&Q, start);                  // enqueue with start node
    while(empty(&Q)== FALSE)
     {  v = dequeue(&Q);                    // call dequeue
        printf("visit node : %d", v);    // print visit node
        visited[v] = TRUE;                  // mark visited v
        for(i=0;i < g->nvertrices; i++)    // explore neighbors
          { if(visited[i] == FALSE)
            { if(g->edges[i][v]== 1 || g->edges[v][i] == 1)
                    { enqueue(&Q, i);       // next explore
                      parent[i] = v;
                    }
            }
          }
     }
 }
```

Depth First Search

- เริ่มต้นที่ start vertex ในกราฟ
- ขยาย Path จาก start vertex ไปทีละ path
- เมื่อไม่สามารถทำต่อไปได้ DFS จะทำ back edge เพื่อเปลี่ยนเส้นทางสำรวจไปยัง path อื่น จนกระทั่งทุก vertex ถูกสำรวจ
- ข้อดีของ DFS คือ
  - Save หน่วยความจำเมื่อสำรวจกราฟขนาดใหญ่
  - พัฒนาง่าย และทำงานได้รวดเร็วกว่า BFS

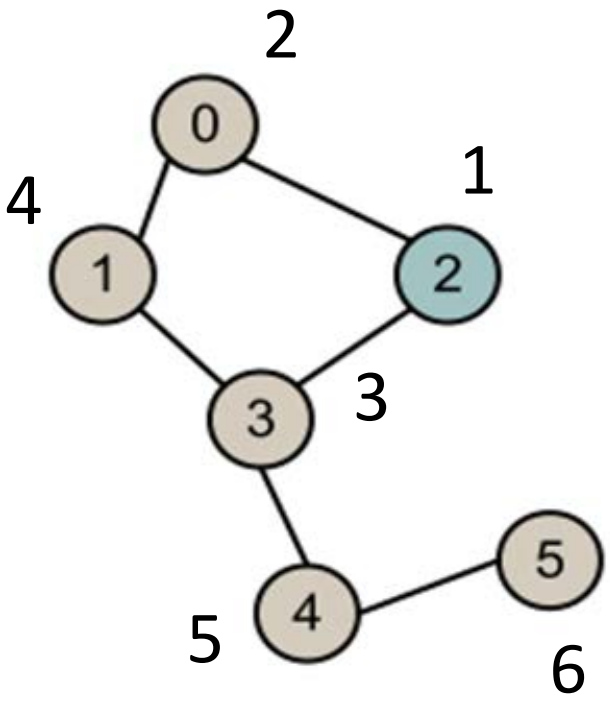
ตัวอย่าง DFS



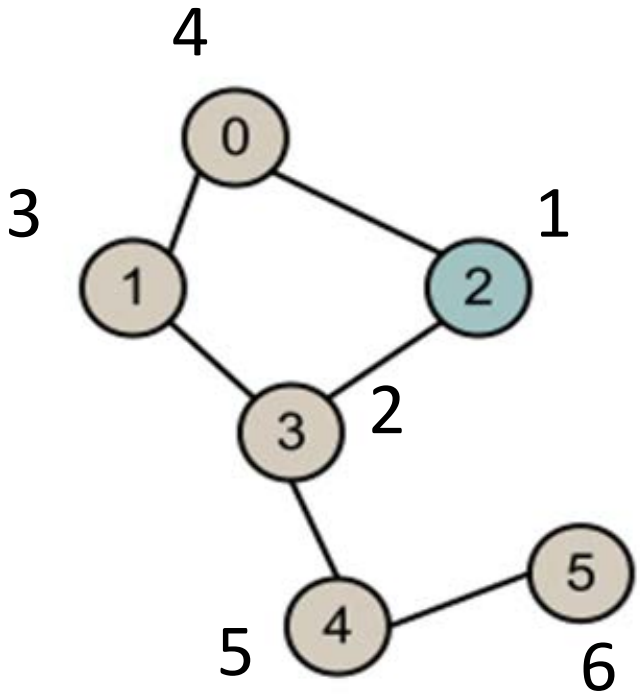
stack{}
stack{2}
stack{3, 2}
stack{1,3,2}
stack{0,1,3,2}

stack{1,3,2}
stack{3,2}
stack{4,3,2}
stack{5,4,3,2}
stack{4,3,2}
stack{3,2}
stack{2}
stack{0}

ลำดับในการเยี่ยมเวอร์เท็กซ์ของ BFS และ DFS



BFS



DFS

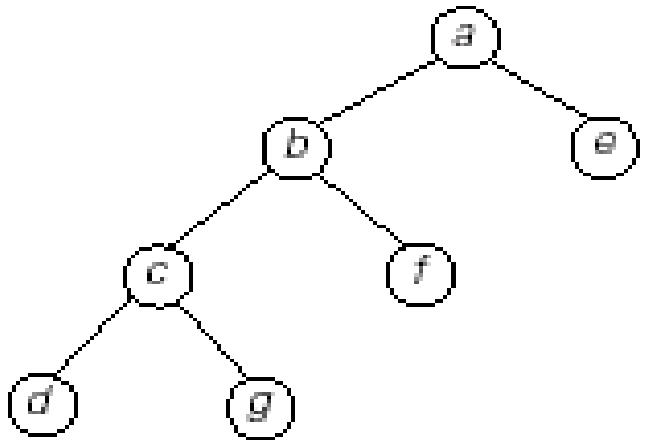
DFS algorithm

```
void DFS (graph *g, int v)
 { int i;
    printf("%d ", start);                    // process visit node
    visited[v] = TRUE;                  // mark visit
    for(i=0; i < g->nvertrices; i++)        // explore neighbors
      { if(g->edges[i][v]== 1 || g->edges[v][i] == 1)
          { if(visit[i] == FALSE)           // if not visit
              {  parent[i] = v;
                 DFS(g, i);
              }
       }
 }
```

การหา path ในกราฟ

```
int find_path(int start, int end, int parent[])
{ int p = 0;

    if((start == end || end == -1)
        return 0;
   else
         return find_path(start, parent[end], parent) + 1;
}
```

0	1	2	3	4	5	6
-1	0	1	2	0	1	2

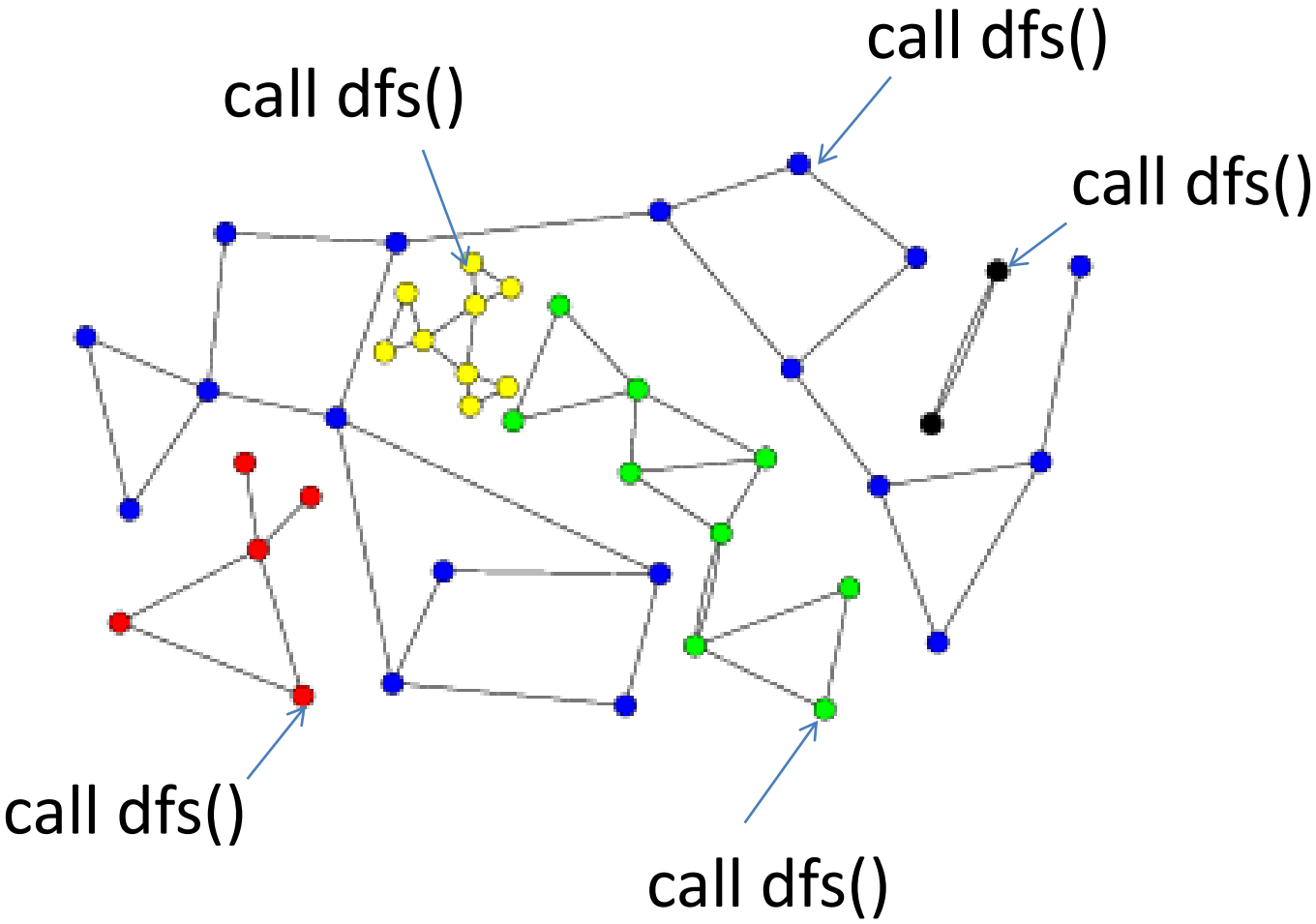


Connected Components

เนื่องจากนิยามของ connected components ซึ่งกล่าวว่า component หมายถึงเซตของเวอร์เท็กซ์ที่เชื่อมโยงซึ่งกันและกัน ดังนั้นเราจึงสามารถประยุกต์ใช้ DFS เพื่อค้นหาจำนวน component ในกราฟได้อย่างไม่ยากนัก

วิธีการก็คือ ทุกครั้งที่มีการเรียก DFS โดยส่งเวอร์เท็กซ์เริ่มต้นไป เมื่อ DFS ทำงานเสร็จ เราก็จะ label ว่าเวอร์เท็กซ์ที่ถูกเยี่ยมเป็น component เดียวกัน ดังนั้นหากเวอร์เท็กซ์ใดที่ยังไม่ถูกเยี่ยมโดย DFS นั่นแปลว่ามี component อื่นๆ อยู่ในกราฟ เราก็เพียงแต่เรียก DFS อีกรอบโดยกำหนดเวอร์เท็กซ์เริ่มต้นไปยังเวอร์เท็กซ์ที่ยังไม่ได้ถูกเยี่ยม จนครบทุกเวอร์เท็กซ์ในกราฟ

5 connected components



Connected Component with DFS

```
Connected_component(graph *g)
{  int c;
   int i;

   c = 0;
   for(i=0; i< g-> nvertices; i++)
    { if(visited[i] = FALSE) {
         {   c = c +  1;
             printf("Component %d:", c);
             dfs(g, i);
             printf("\n");
         }
    }
}
```