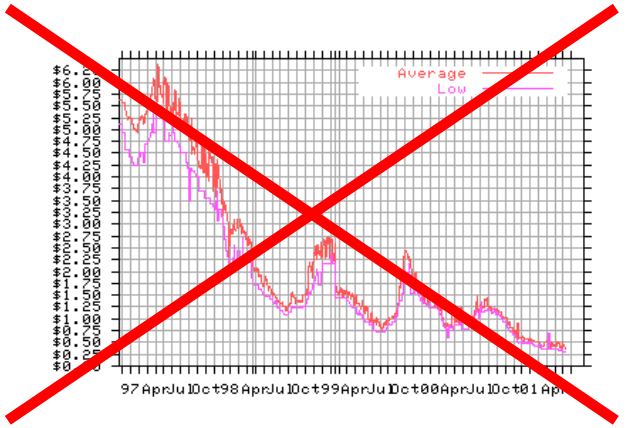
Design and Analysis of Data Structures and Algorithms :: Graph part 1

อ.ดร.วรินทร์ วัฒนพรพรหม

โครงสร้างข้อมูลกราฟ (Graph)

กราฟ (Graph) เป็นโครงสร้างข้อมูลแบบไม่ใช่เชิงเส้นอีกชนิดหนึ่ง กราฟเป็นโครงสร้างข้อมูลที่มีการนำไปใช้ในงานที่เกี่ยวข้องกับการแก้ปัญหาที่ค่อนข้างซับซ้อน เช่น การวางข่ายงานคอมพิวเตอร์ การวิเคราะห์เส้นทางวิกฤติ และปัญหาเส้นทางที่สั้นที่สุด เป็นต้น



โครงสร้างข้อมูลกราฟ (Graph)

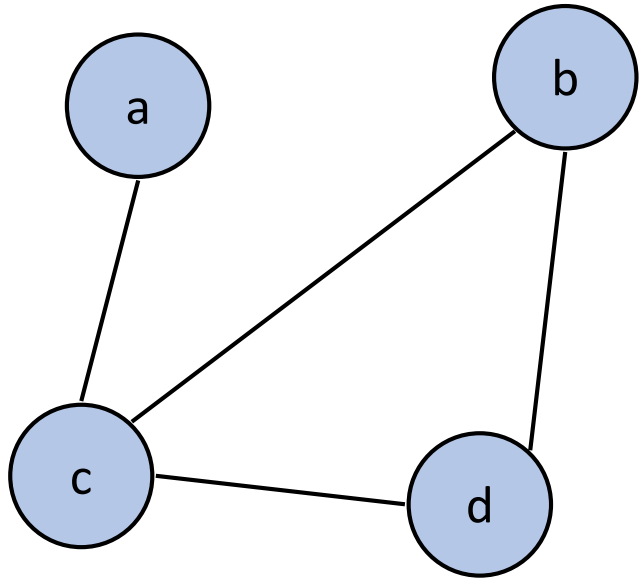
ศัพท์ที่เกี่ยวข้อง

1. เวอร์เทก (Vertex) หมายถึง โหนด (Node)
2. เอดจ์ (Edge) หมายถึง เส้นเชื่อมของโหนด
3. ดีกรี (Degree) หมายถึง จำนวนเส้นเข้าและออก ของโหนดแต่ละโหนด
4. แอดจาเซนท์โหนด (Adjacent Node) หมายถึง โหนดที่มีการเชื่อมโยงกัน

นิยามกราฟ

- กราฟ เป็นโครงสร้างที่นำมาใช้เพื่อแสดงความสัมพันธ์ระหว่างวัตถุ โดยแทนวัตถุด้วยเวอร์เท็กซ์ (Vertex) หรือโหนด (Node) และเชื่อมโยงความสัมพันธ์ด้วยเอดจ์ (Edge)
- เขียนในรูปของสัญลักษณ์ได้เป็น $G = (V,E)$
- $V(G)$ คือ เซตของเวอร์เทกซ์ที่ไม่ใช่เซ็ตว่าง และมีจำนวนจำกัด
- $E(G)$ คือ เซตของเอดจ์ ซึ่งเขียนด้วยคู่ของเวอร์เท็กซ์

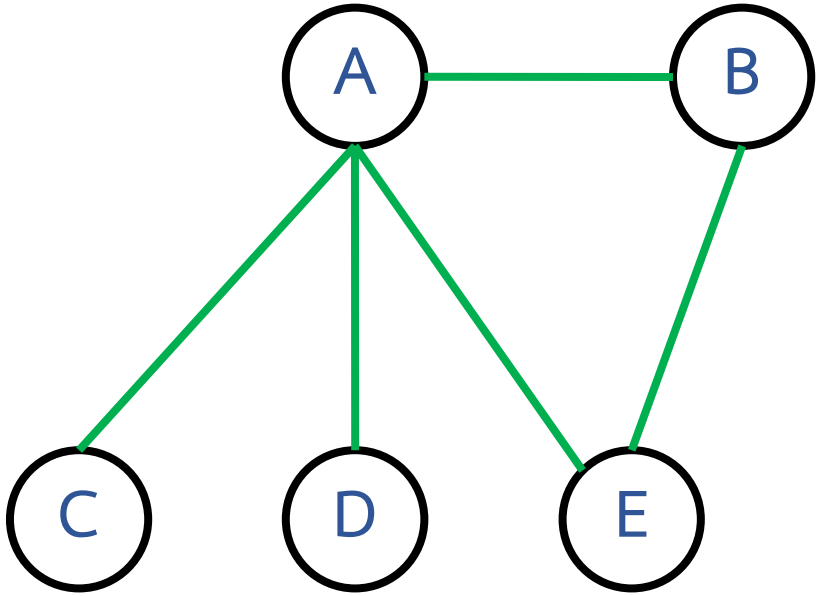
องค์ประกอบโครงสร้างข้อมูลแบบกราฟ



V = {a, b, c, d}

E = {(a, c), (b, c), (b, d), (c, d)}

ตัวอย่าง



เวอร์เท็กซ์ V(G) ได้แก่ A B C D E

เอดจ์ E(G) ได้แก่

เส้นที่เชื่อมโยงจาก A ไป B

เส้นที่เชื่อมโยงจาก A ไป C

เส้นที่เชื่อมโยงจาก A ไป D

เส้นที่เชื่อมโยงจาก A ไป E

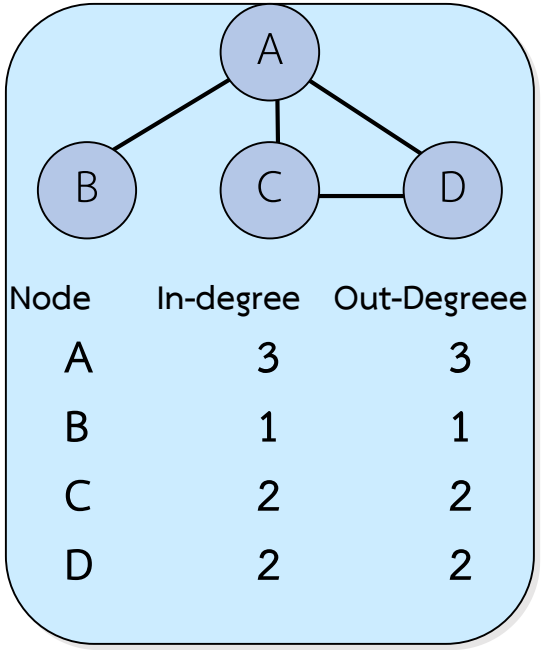
เส้นที่เชื่อมโยงจาก B ไป E

V(G) = {A, B, C, D, E}

E(G) = {(A,B), (A,C), (A,D), (A,E), (B,E)}

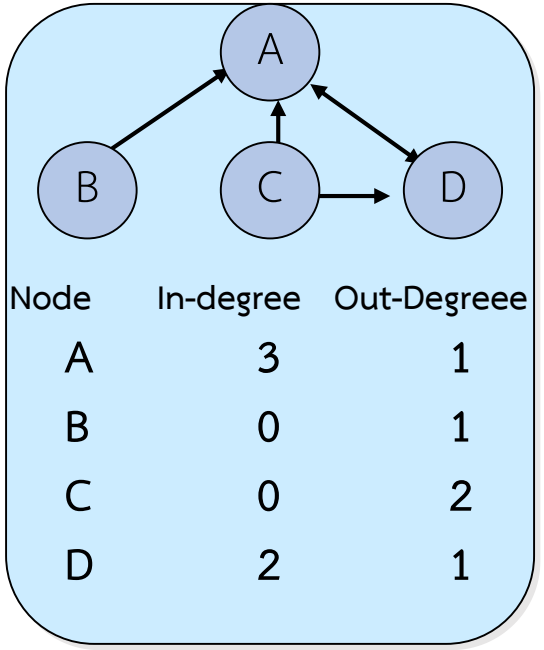
อินดีกรีและเอาท์ดีกรี

- แต่ละโหนดจะมีจำนวนเส้นเชื่อมระหว่างโหนดไม่เท่ากัน
- In-degree แสดงจำนวนเส้นเชื่อมที่เข้ามายังโหนดนั้นๆ
- Out-degree แสดงจำนวนเส้นเชื่อมที่ออกจากโหนดนั้น
- ใน Undirected Graph จำนวน In-degree และ Out-degree จะเท่ากัน



Node	In-degree	Out-Degreee
A	3	3
B	1	1
C	2	2
D	2	2

Undirected Graph

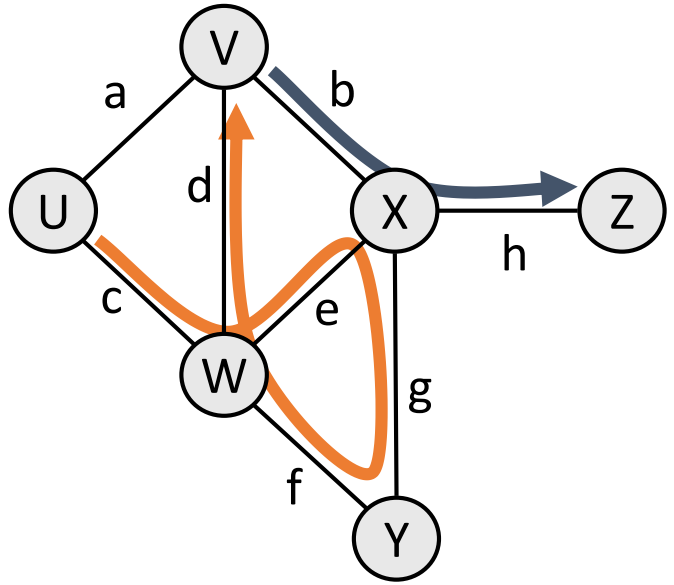


Node	In-degree	Out-Degreee
A	3	1
B	0	1
C	0	2
D	2	1

Directed Graph

โครงสร้างข้อมูลแบบกราฟ (Graph)

- แอดจาเซนท์โหนด (Adjacent Node) หรือ เพื่อนบ้าน (neighbor)

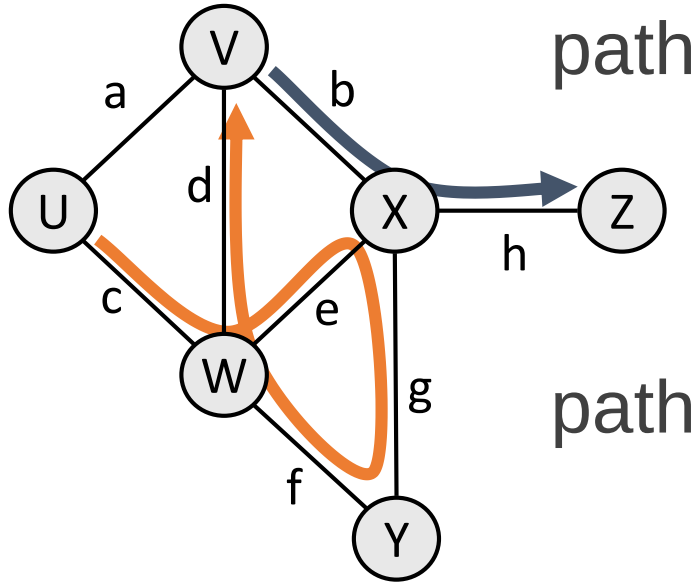


U กับ V ใช่

W กับ Z ไม่ใช่

โครงสร้างข้อมูลแบบกราฟ (Graph)

- **เส้นทาง (Path)** ใช้เรียกลำดับของ เวอร์เทก (Vertex) ที่เชื่อมต่อกันจากจุดหนึ่งไปยังอีกจุดหนึ่ง



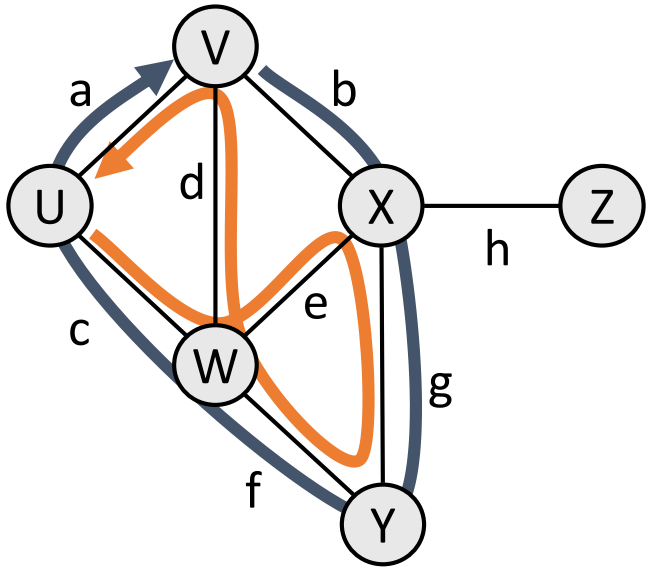
path from *V* to *Z*: {b, h} or {V, X, Z}

path from *U* to *V*: ????

path length: Number of vertices or edges contained in the path.

โครงสร้างข้อมูลแบบกราฟ (Graph)

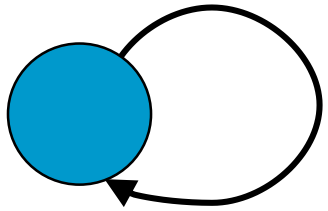
- **Cycle** คือ Path ที่ประกอบด้วยอย่างน้อย 3 Vertex และมีจุดเริ่มต้นและสิ้นสุดเดียวกัน



example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
example: {c, d, a} or {U, W, V, U}.

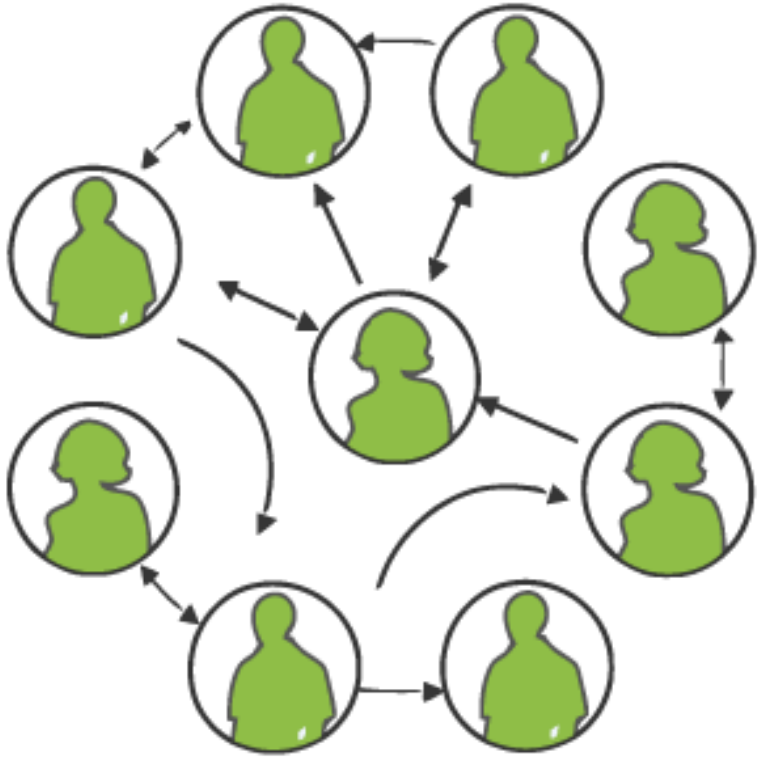
โครงสร้างข้อมูลแบบกราฟ (Graph)

- **ลูป (Loop)** มีเพียง Edge เดียวและมีจุดเริ่มต้นและสิ้นสุดเดียวกัน

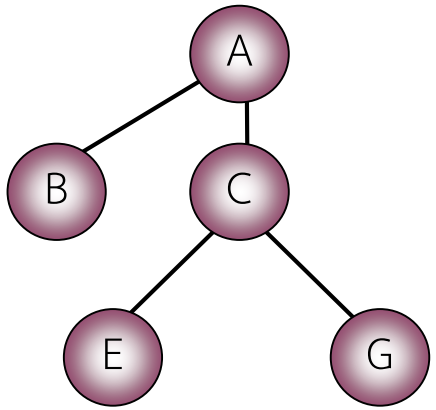


ตัวอย่างของกราฟ

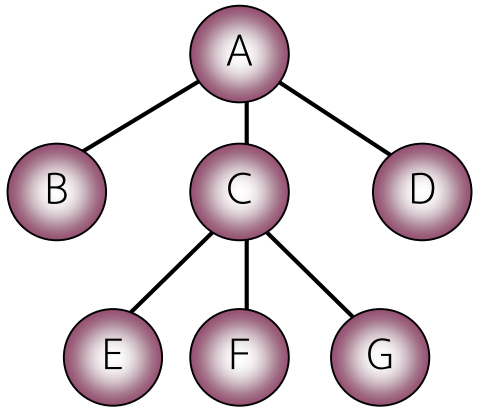
- Web pages with links
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Facebook friends
- Course pre-requisites
- Family trees
- Paths through a maze



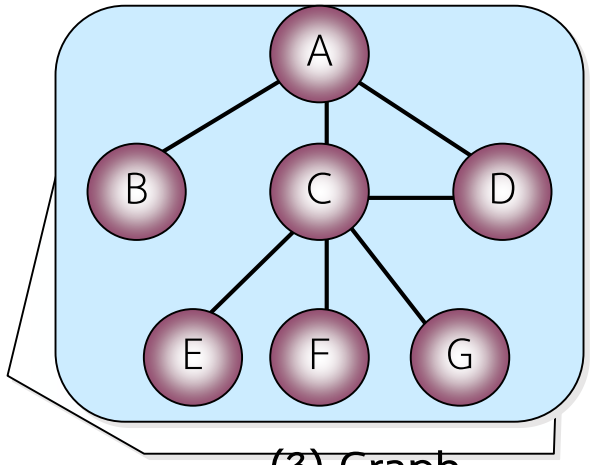
Graph VS. Tree



(1) Binary tree

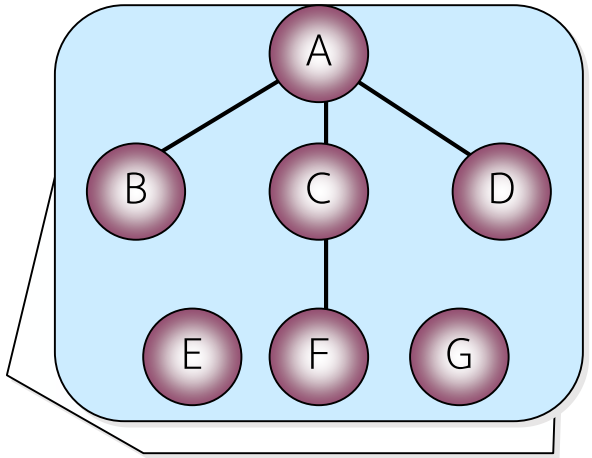


(2) Non binary tree



(3) Graph

โหนด C,D มี Parent โหนดมากกว่า 1



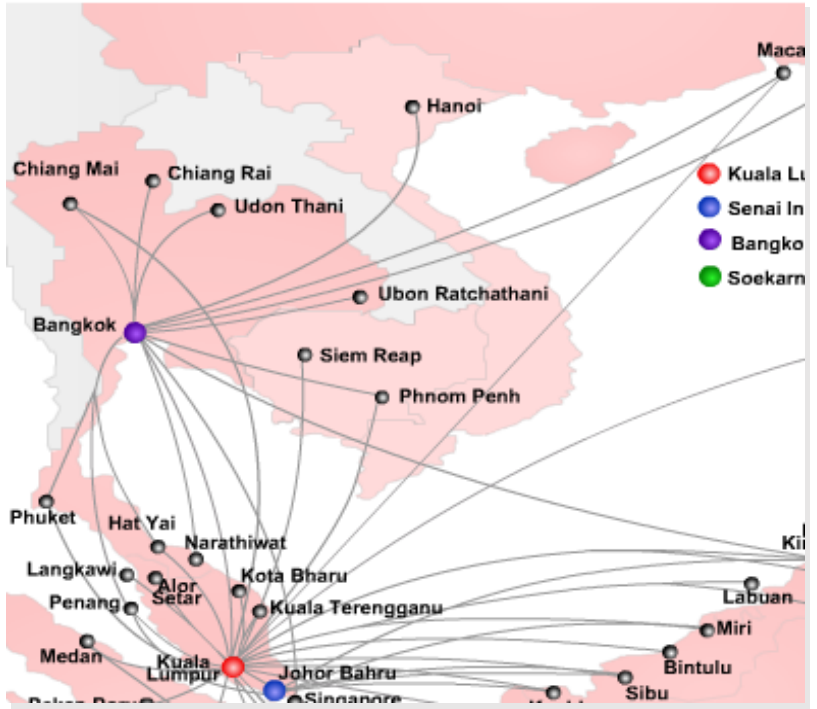
(4) Graph

โหนด E,G ไม่มีเส้นเชื่อม

- กราฟเป็น Super Set ของต้นไม้
- Tree ต้องมี Parent Node เพียงโหนดเดียว, แต่ Graph ไม่จำเป็น
- บางโหนดอาจไม่มีเส้นเชื่อมได้ เช่น บางเมืองไม่มีสายการบิน

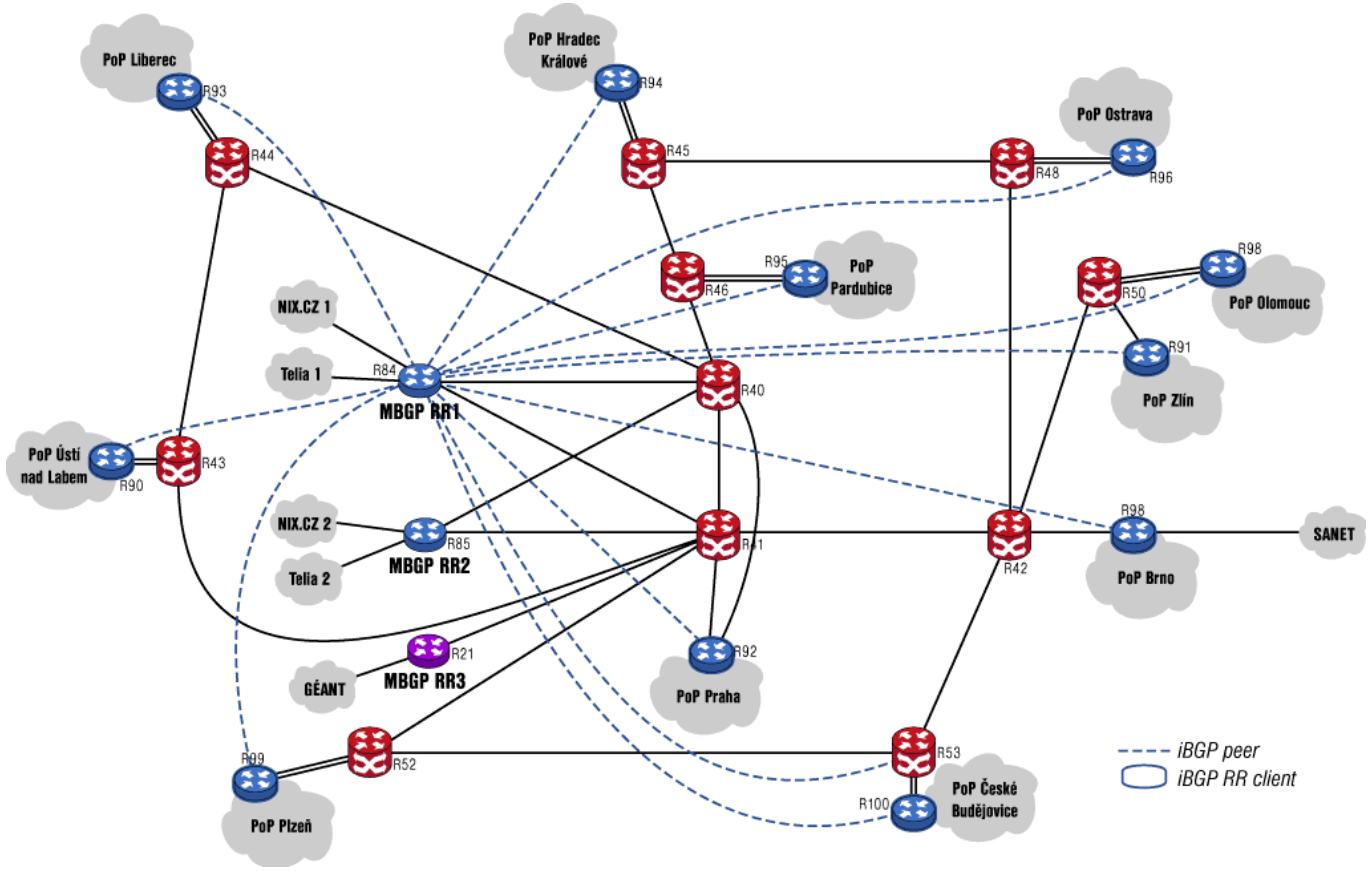
ประโยชน์ของกราฟ (Routing การหาเส้นทาง)

สายการบิน (การเชื่อมต่อของสายการบิน ตารางบิน)



ประโยชน์ของกราฟ (Routing การหาเส้นทาง)

- Network (การเชื่อมต่อของอุปกรณ์ Router)
- เพื่อใช้ในการรับส่งข้อมูลในเครือข่าย



ประโยชน์ของกราฟ (Algorithm Design)

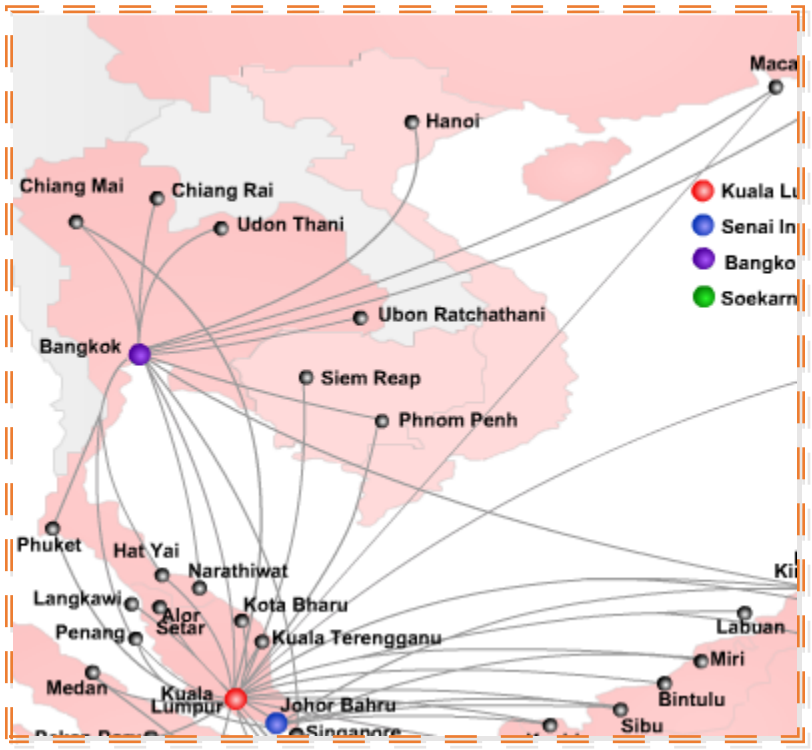
- Map Coloring คือวิธีการระบายสีในแผนที่โดยใช้สีน้อยที่สุด
- พื้นที่ติดกันห้ามใช้สีเดียวกัน

ชนิดของกราฟ - ทิศทาง และ น้ำหนัก

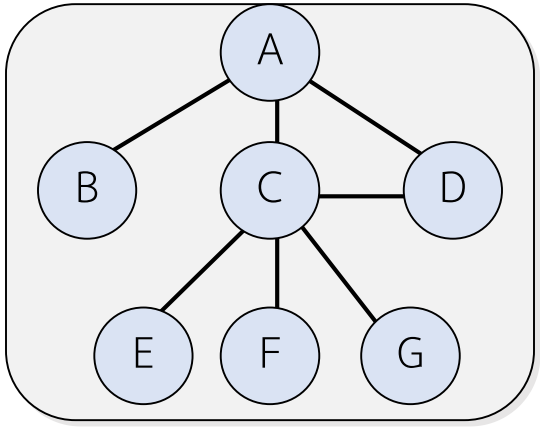
- กราฟแบบไม่มีทิศทาง (Undirected Graph)
- กราฟแบบมีทิศทาง (Directed Graph) หรือ Digraph
- กราฟไม่มีน้ำหนัก (Unweighted Graph)
- กราฟมีน้ำหนัก (Weighted Graph)

Directed & Undirected Graph

- Undirected Graph คือกราฟที่เส้นเชื่อมไม่มีลูกศรกำกับทิศทาง
- หมายถึงความสัมพันธ์ของ 2 โหนดแบบไปและกลับ



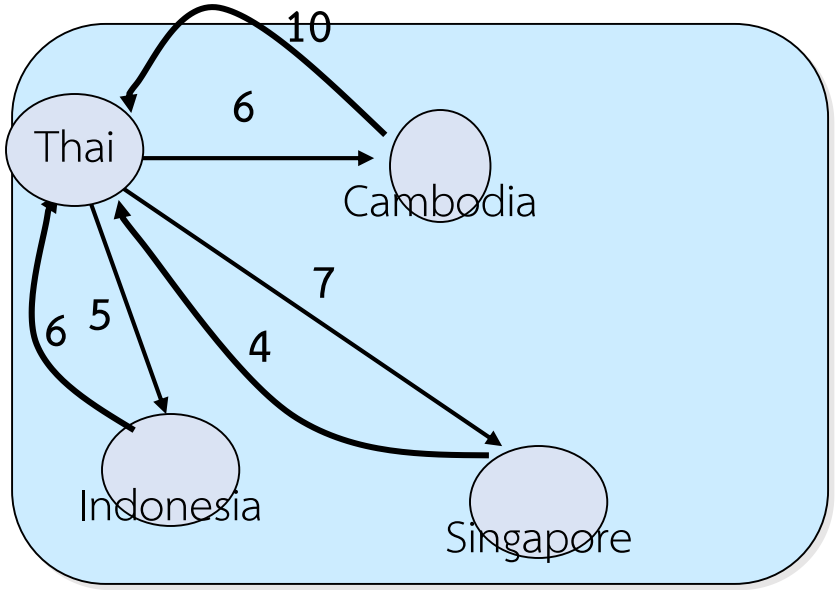
Undirected Graph แสดงสายการบินของ Air Asia



Undirected Graph แสดงโหนดและเส้นเชื่อมของกราฟรูปหนึ่ง

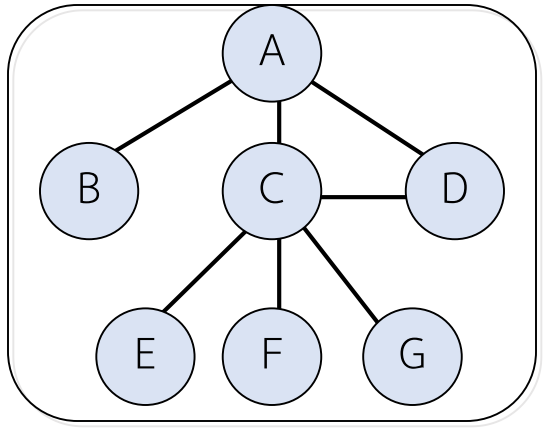
Directed & Undirected Graph

- Directed Graph คือกราฟที่เส้นเชื่อมมีลูกศรกำกับทิศทาง
- เช่น อาจมีสายการบินจาก กรุงเทพ-เสียมเลียบ กัมพูชา แต่ไม่มีสายการบินจาก เสียมเลียบ-กรุงเทพ
- หรือ Edge แสดงค่าโดยสารที่มีราคา ไป-กลับไม่เท่ากัน
- หรือ ค่าโทรศัพท์ไทยไปสิงคโปร์ แพงกว่าสิงคโปร์โทรหาไทย

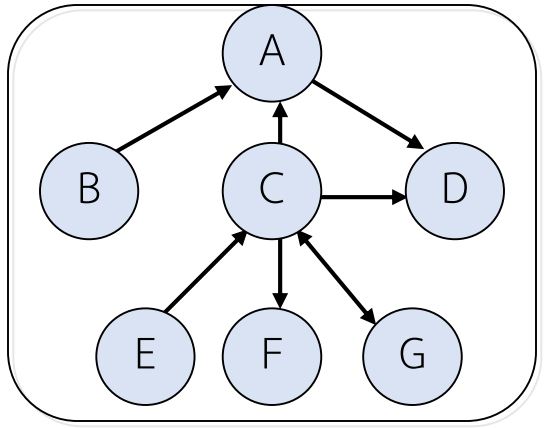


Unweighted Graph

- กราฟแบบนี้เส้นเชื่อมแสดงถึงความสัมพันธ์ของ 2 โหนด (เหมือนกราฟทั่วไป)
- เส้นเชื่อมแสดงถึงความหมายบางอย่าง (เหมือนกับกราฟทั่วไป)
- แต่ไม่ระบุข้อมูลหรือค่าบางอย่าง (แตกต่างจากสิ่งอื่นๆ)
    - เช่น ถนนที่เชื่อมเมือง 2 เมืองแต่ไม่ระบุระยะทาง
    - ผังรถไฟฟ้าใต้ดิน แต่ไม่ระบุราคาค่าโดยสารระหว่างสถานี
    - หรือมองว่าค่าข้อมูลเหล่านั้นมีค่าเท่ากันหมด
- อาจเป็น Directed หรือ Undirected Graph ก็ได้



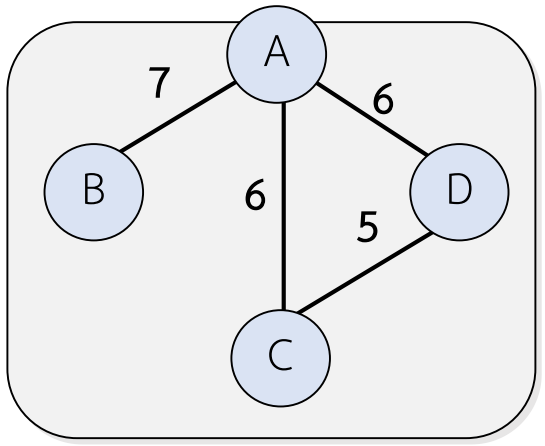
Unweighted & Undirected Graph



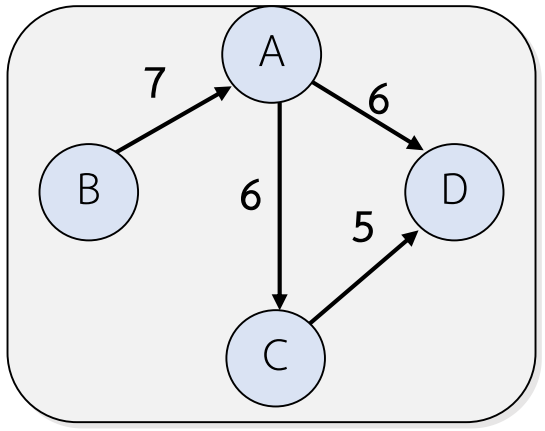
Unweighted & directed Graph

Weighted Graph

- กราฟแบบนี้เส้นเชื่อมแสดงถึงความสัมพันธ์ของ 2 โหนด (เหมือนกราฟทั่วไป)
- เส้นเชื่อมแสดงถึงความหมายบางอย่าง (เหมือนกับกราฟทั่วไป)
- เส้นเชื่อมระบุข้อมูลหรือค่าบางอย่างที่ต้องการบ่งชี้
  - เช่น ถนนที่เชื่อมเมือง 2 เมืองพร้อมระบุระยะทางระหว่างเมือง
  - อาจเป็น Directed หรือ Undirected Graph ก็ได้



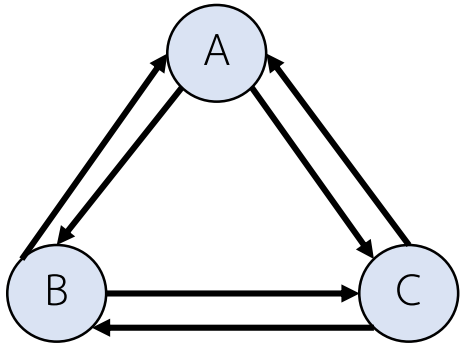
Weighted & Undirected Graph



Weighted & directed Graph

Complete Graph (กราฟสมบูรณ์)

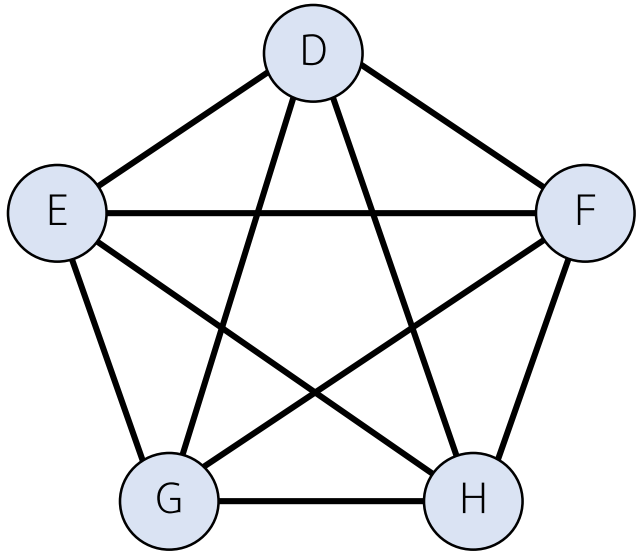
- กราฟที่ทุกโหนดมีเส้นเชื่อมถึงโหนดอื่นๆ ทั้งหมด



กราฟมีทิศทาง

จำนวน Edge = N*(N-1)

เช่น 3*(3-1) =6



กราฟไม่มีทิศทาง

$$\text{จำนวน Edge} = \frac{N*(N-1)}{2}$$

เช่น 5*(5-1) / 2 = 10

Graph Storage Structure

การแทนที่กราฟในหน่วยความจำสามารถทำได้ 2 แบบ ดังนี้

1. Adjacency Matrix : ใช้อาร์เรย์เก็บข้อมูล

2. Adjacency List : ใช้ลิงค์ลิสต์เก็บข้อมูล

Adjacency Matrix

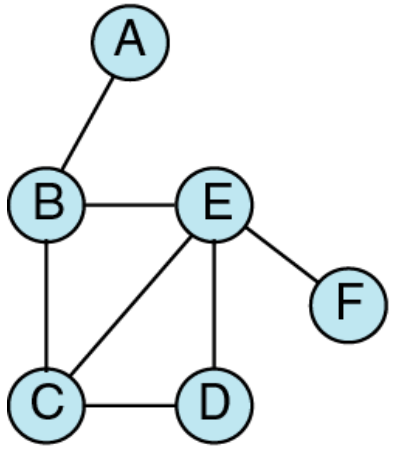
ใช้อาร์เรย์หนึ่งมิติเพื่อเก็บVertex และใช้เมตริกซ์ (อาร์เรย์สองมิติ) เพื่อเก็บ Edge

- เป็นโครงสร้างที่ประกอบไปด้วยโหนดและเส้นเชื่อมต่อที่บอกถึงเส้นทางของการเดินทาง หรือความสัมพันธ์ในทิศทางซึ่งสามารถนำมาแทนความสัมพันธ์นั้นด้วยการกำหนดเมตริกซ์ $n \times n$
- M^k เป็นเมทริกซ์ของกราฟใด ๆ k คือทางเดินที่มีความยาว k จากโหนดหนึ่งไปอีกโหนดหนึ่ง

Adjacency Matrix

0 : ไม่เป็นแอดจาเซนซีกัน

1 : เป็นแอดจาเซนซีกัน



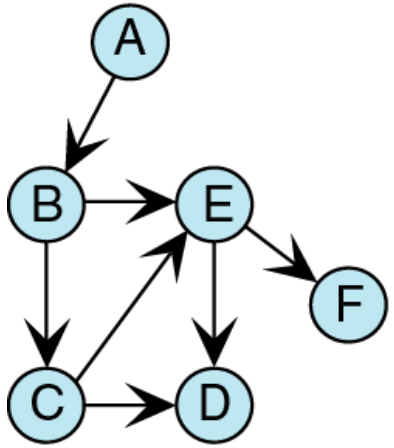
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

(a) Adjacency matrix for directed graph

Simple Graph with AdjMatrix

```
class Graph{
private:
    int** matrix;
    int row,column;
    int maxSize;
public:
    Graph(int inputSize){
        row=column=maxSize=inputSize;
        matrix = new int*[row];              //O(1)
        for(int i=0;i<row;i++)
            matrix[i]=new int[column];       //O(V)
    }

    void Init(){
        for(int i=0;i<row;i++)               //O(V^2)
            for(int j=0;j<column;j++)                  //O(V)
                matrix[i][j]=0;
    }
```

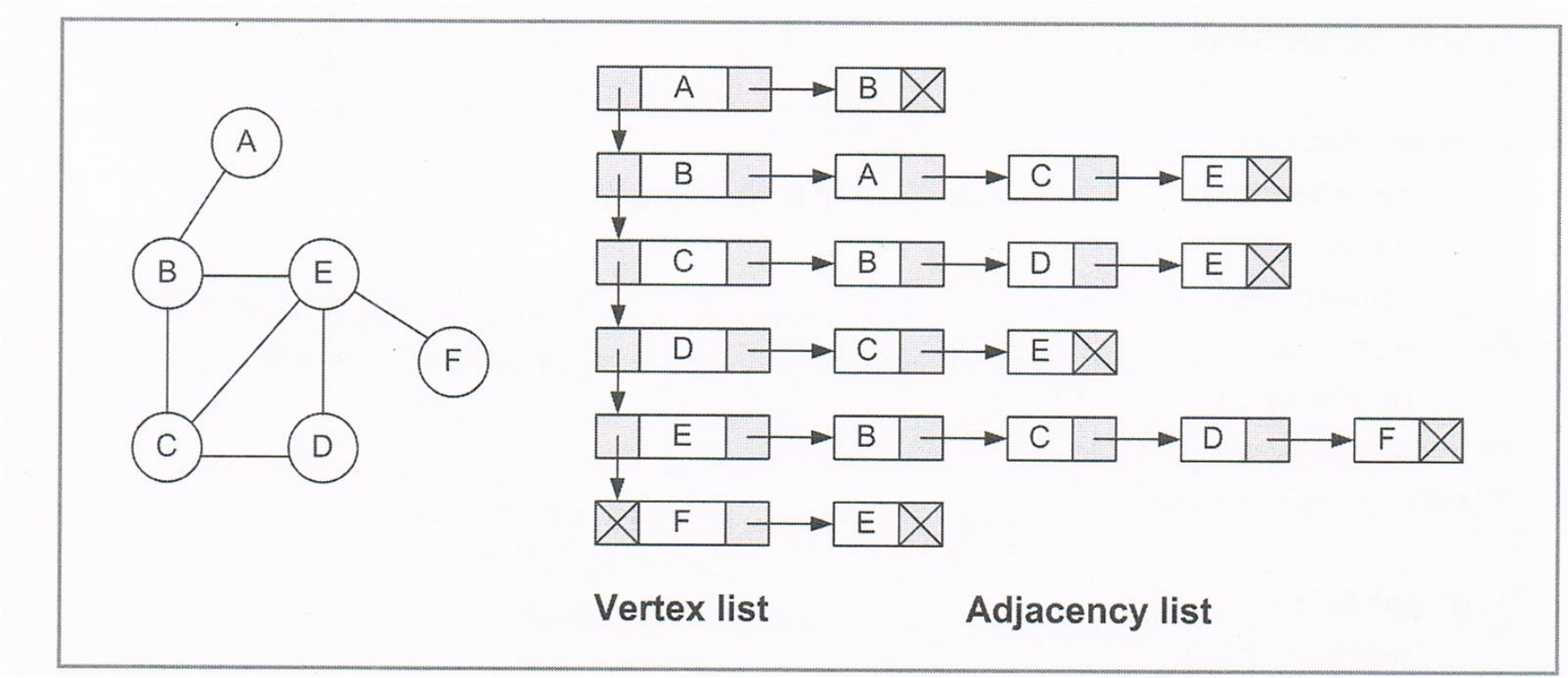
Simple Graph with AdjMatrix

```
int adjacent(int i,int j){
        if(matrix[i][j]>0)                          //O(1)
            return 1;
       return 0;
    }

    int addEdge(int i,int j,int value){
        matrix[i][j]=value;                         //O(1)
        matrix[j][i]=value; // non-directed;
    }

    int removeEdge(int i,int j){
        matrix[i][j]=0;                                     //O(1)
        matrix[j][i]=0; // non-directed;
    }
```

Simple Graph with AdjMatrix

```
int neighbors(int i){
        for(int j=0;j<row;j++)                          //O(v)
            if(matrix[i][j]>0)
            cout << j << " ";
    }


void PrintMatrix(){
        for(int i=0;i<row;i++){                         //O(v^2)
            for(int j=0;j<column;j++)           //O(v)
               cout<<matrix[i][j] << " ";
            cout << endl;
        }

    }
};
```

การแทนด้วย Adjacency List



Vector in C++ and Java

Note that in below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of linked list. The vector implementation has advantages of cache friendliness.

```
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);              //O(v)
    adj[v].push_back(u);
}
```

Vector in C++ and Java

```
// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)        //O(v^2)
    {
        cout << "\n Adjacency list of vertex "
             << v << "\n head ";
        for (auto x : adj[v])          //O(v)
           cout << "-> " << x;
        printf("\n");
    }
}
```

Vector in C++ and Java

```
// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
    return 0;
}
```

Vector in C++ and Java

```
Adjacency list of vertex 0
 head -> 1-> 4

 Adjacency list of vertex 1
 head -> 0-> 2-> 3-> 4

 Adjacency list of vertex 2
 head -> 1-> 3

 Adjacency list of vertex 3
 head -> 1-> 2-> 4

 Adjacency list of vertex 4
 head -> 0-> 1-> 3
```

การท่องไปในกราฟ (Graph Traversal)

- หลักในการทำงานท่องไปในกราฟ คือ แต่ละโหนดจะถูกเยือนเพียงครั้งเดียว ดังนั้นจึงต้องมีค่าที่บอกว่าโหนดใดได้ถูกเยือนไปแล้ว
- เทคนิคการท่องไปในกราฟ มี 2 แบบ คือ
  - Depth-First Traversal
  - Breadth-First Traversal

Depth-First Traversal

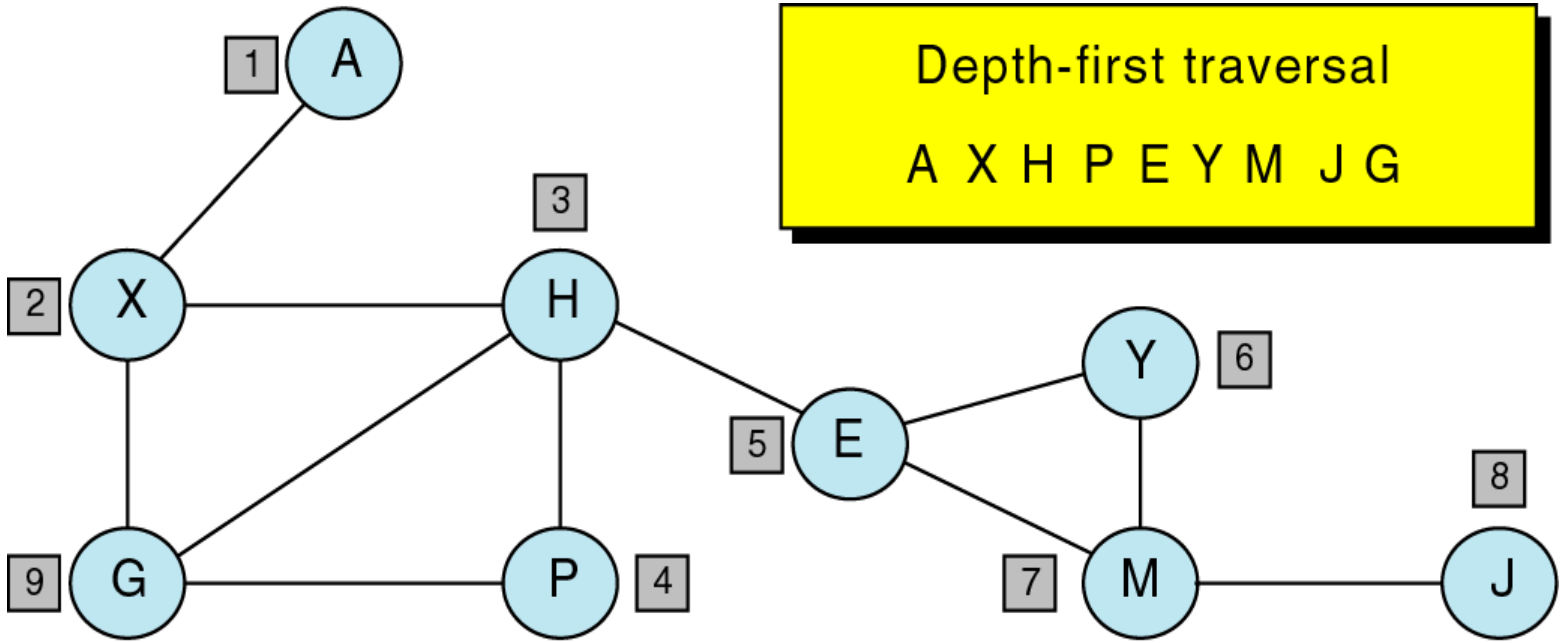
- **การท่องแบบลึก (Depth-First Traversal)** การทำงานคล้ายกับการท่อง

  ทรีโดยกำหนดเริ่มต้นที่โหนดแรก และเยือนโหนดถัดไปตามแนววิถีนั้นจนกระทั่งนำไปสู่ปลายวิถีนั้น จากนั้นย้อนกลับ **(Backtrack)** ตามแนววิถีเดิมนั้น จนกระทั่งสามารถดำเนินการต่อเนื่องเข้าสู่แนววิถีอื่นๆ เพื่อเยือนโหนดอื่นๆ ต่อไปจนครบทุกโหนด

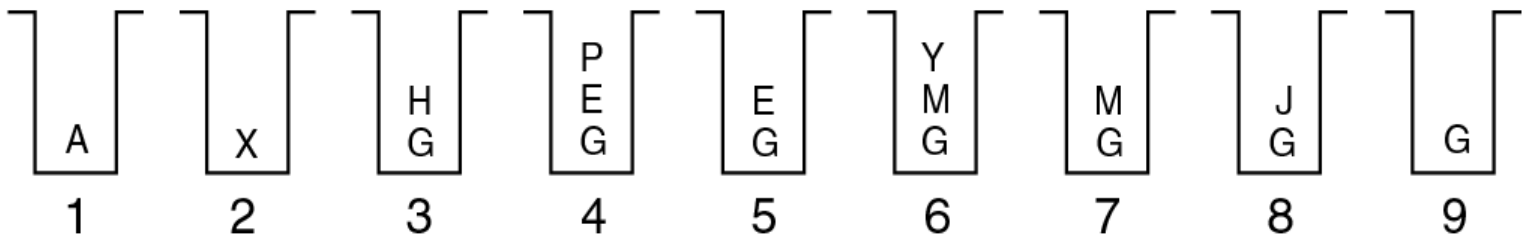
Depth-First Traversal

- **ขั้นตอนการทำงานของ Depth-First Traversal**

  1. ทำให้ทุกโหนดบนกราฟมีสถานะเป็น 1
  2. Push โหนดเริ่มต้นลงใน Stack แล้วเปลี่ยนสถานะเป็น 2
  3. Pop โหนดใน Stack ออกมาทำการเยี่ยมแล้วเปลี่ยนสถานะเป็น 3
  4. Push โหนดข้างเคียงของโหนดในหัวข้อ 3 ที่มีสถานะเป็น 1 ลง Stack แล้วเปลี่ยนสถานะเป็น 2
  5. กลับไปทำข้อ 3 จนกระทั่งไม่มีโหนดใน Stack



(a) The graph



(b) Stack contents

Breadth-First Traversal

- **การท่องแบบกว้าง (Breadth-First Traversal)** วิธีนี้ทำโดยเลือกโหนดที่เป็นจุดเริ่มต้น ต่อมาให้เยือนโหนดอื่นที่อยู่ใกล้กันกับโหนดเริ่มต้นทีละระดับจนกระทั่งเยือนหมดทุกโหนดในกราฟ

Breadth-First Traversal

- **ขั้นตอนการทำงานของ Breadth-First Traversal**

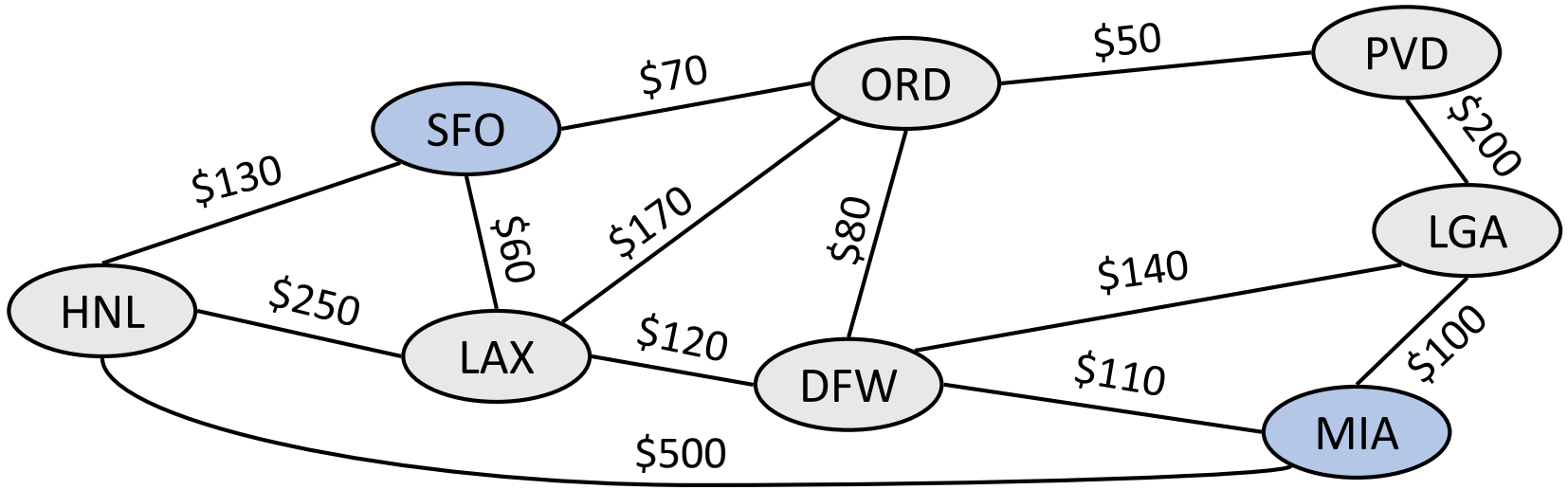
  1. ทำให้ทุกโหนดบนกราฟมีสถานะเป็น 1
  2. ให้เอาโหนดเริ่มต้นมาใส่ใน Queue แล้วเปลี่ยนสถานะเป็น 2
  3. ให้นำโหนดแรกสุดของ Queue ออกมาเยือนแล้วเปลี่ยนสภานะเป็น 3
  4. สำรวจตัวข้างเคียงตัวของโหนดที่นำออกมาตามข้อ 3

     ถ้าโหนดใดมีสถานะเป็น 1 ให้นำใส่ Queue

     ถ้าสถานะเป็นอย่างอื่นก็ไม่ต้องทำอะไรกับโหนดเหล่านั้น
  5. กลับไปทำข้อ 3 จนกระทั่งไม่มีโหนดอยู่ใน Queue

การค้นหาเส้นทาง (Path Searching)

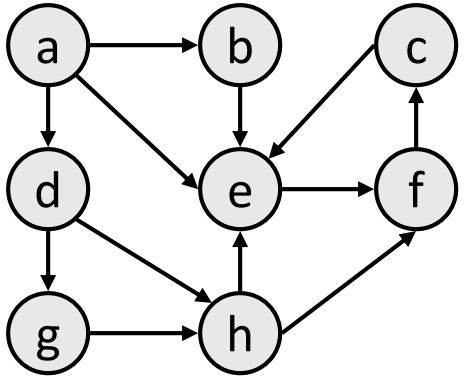
- ค้นหาเส้นทางจากเวอร์เท็กซ์หนึ่งไปอีกจุดหนึ่ง :
  - บางครั้งเราต้องการเส้นทางใด ๆ (หรือต้องการทราบว่าแค่มีเส้นทาง)
  - บางครั้งเราต้องการลดความยาวของเส้นทาง (# เอดจ์)
  - เราต้องการหาเส้นทางที่สั้นที่สุด (ผลรวมของน้ำหนักเอดจ์)
- เส้นทางที่สั้นที่สุดจาก MIA ถึง SFO คืออะไร?
- เส้นทางไหนมีต้นทุนต่ำที่สุด?



Depth-first search

- **depth-first search** (DFS): ค้นหาเส้นทางระหว่างจุดยอดสองจุดโดยสำรวจแต่ละเส้นทางที่เป็นไปได้ให้ไกลที่สุดก่อนที่จะย้อนรอย (backtracking).
  - มักเขียนด้วย recursive
  - อัลกอริทึมสำหรับกราฟจำนวนมากเกี่ยวข้องกับการท่องการเข้าถึงหรือทำเครื่องหมาย vertices ว่าเดินทางผ่านมาแล้ว

- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



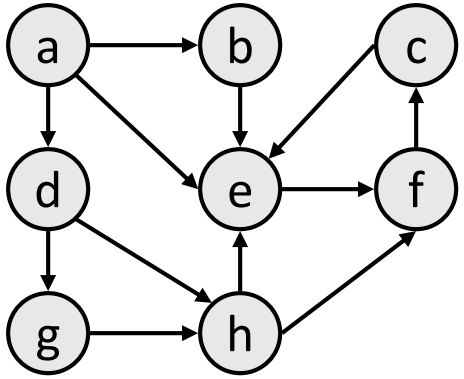
DFS pseudocode

method **dfs**(v_1, v_2):
  dfs(v_1, v_2, { }).

method **dfs**(v_1, v_2, *path*):
  path += v_1.
  mark v_1 as visited.
  if v_1 is v_2:
    a path is found!

  for each unvisited neighbor *n* of v_1:
    if **dfs**(*n*, v_2, *path*) finds a path: a path is found!

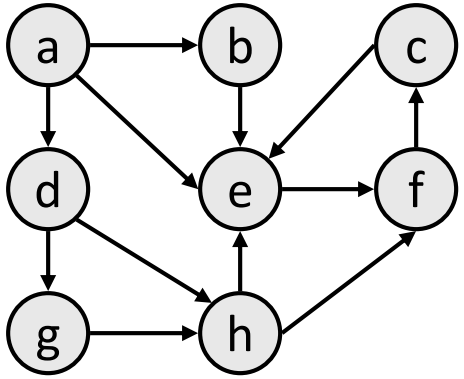
  path -= v_1. // path is not found.



- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

DFS observations

- *discovery*: DFS is guaranteed to find *a* path if one exists.



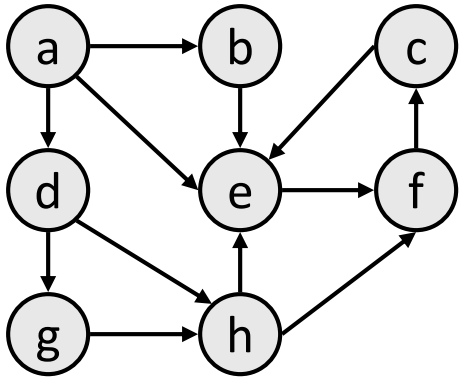
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it

- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, f) returns {a, d, c, f} rather than {a, d, f}.

Breadth-first search

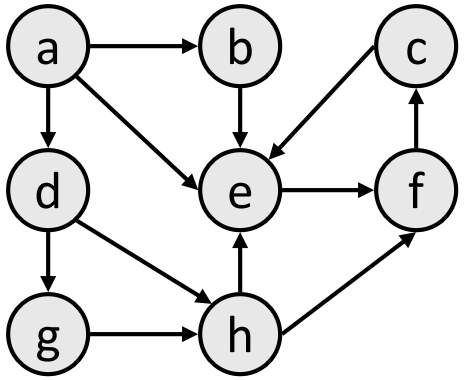
- **breadth-first search** (BFS): ค้นหาเส้นทางระหว่างโหนดสองโหนดโดยก้าวลงจากทุกเส้นทางหนึ่งก้าวแล้วย้อนกลับ (backtracking) ทันที
- มักจะดำเนินการโดยการจัดคิวของเวอร์เท็กซ์ที่จะเข้าถึง

- BFS จะส่งคืนเส้นทางที่สั้นที่สุดเสมอ (เส้นทางที่มีเอดจ์น้อยที่สุด) ระหว่างเวอร์เท็กซ์เริ่มต้นและสิ้นสุด.
  - to b: {a, b}
  - to c: **{a, e, f, c}**
  - to d: {a, d}
  - to e: **{a, e}**
  - to f: **{a, e, f}**
  - to g: {a, d, g}
  - to h: **{a, d, h}**



BFS pseudocode

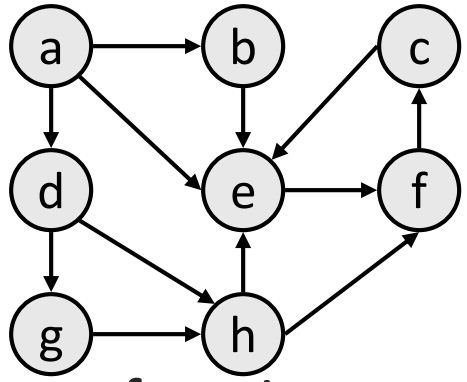
```
method bfs(v1, v2):
    queue := {v1}.
    mark v1 as visited.

    while queue is not empty:
        v := queue.removeFirst().
        if v is v2:
            a path is found!

        for each unvisited neighbor n of v:
            mark n as visited.
            queue.addLast(n).

    // path is not found.
```



- Trace bfs(*a*, *f*) in the above graph.

BFS observations

- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.



- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).

- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

DFS, BFS runtime

- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?

- Answer: $O(|V| + |E|)$
  - where $|V|$ = number of vertices, $|E|$ = number of edges
  - Must potentially visit every node and/or examine every edge once.

  - why not $O(|V| * |E|)$?

- What is the space complexity of each algorithm?
  - (How much memory does each algorithm require?)

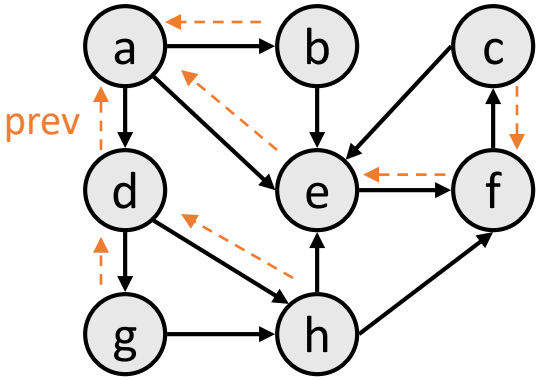
BFS that finds path

method **bfs**(v_1, v_2):
  queue := {v_1}.
  mark v_1 as visited.

  while *queue* is not empty:
    v := *queue*.removeFirst().
    if *v* is v_2:
      a path is found! *(reconstruct it by following .prev back to v_1.)*

    for each unvisited neighbor *n* of *v*:
      mark *n* as visited. *(set n.prev = v.)*
      queue.addLast(*n*).

  // path is not found.



- By storing some kind of "previous" reference associated with each vertex, you can reconstruct your path back once you find v_2.