

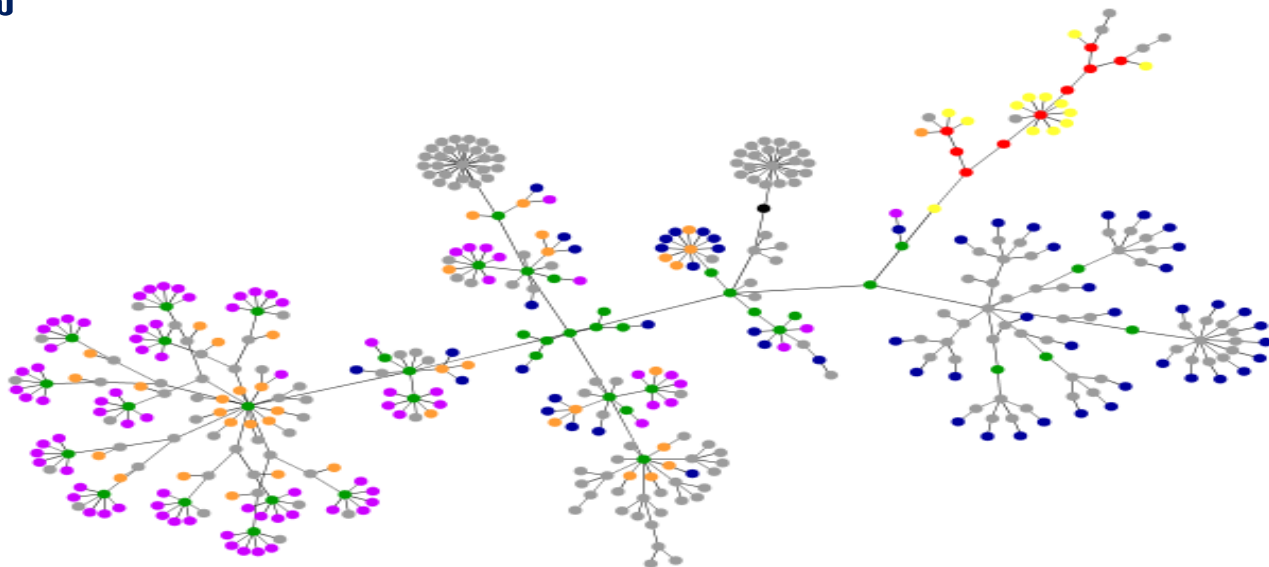


กราฟ - Graph

ค่ายโอลิมปิกวิชาการ สาขาวิชาคอมพิวเตอร์

ค่ายที่ 2 วันที่ 4

วิทยากร - ผู้ช่วยศาสตราจารย์ ดร. อัครา ประโยชน์



Content

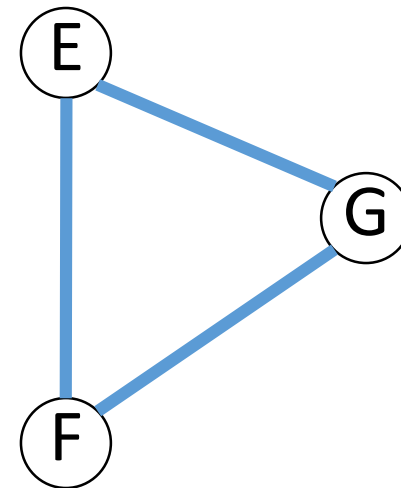
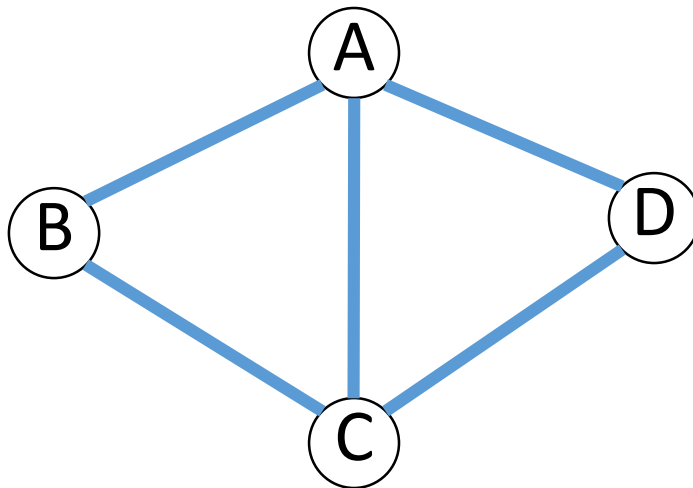
1. Introduction to Graph
2. Graph Representation
3. Traversal
4. Topological sorting

1. Introduction

- ปัญหาหลาย ๆ ปัญหาอยู่ในรูปแบบของ objects และ ความสัมพันธ์ระหว่าง objects เหล่านั้น เช่น
 - ระบบสายการบิน การจัดตารางงาน การเชื่อมต่อวงจร
- กราฟ $G = (V, E)$ คือ กลุ่มของโหนดที่เรียกว่าเวอร์เท็กซ์ (vertex/vertices) และกลุ่มของเอดจ์ (edge) ที่ใช้เชื่อมโยงเวอร์เท็กซ์
 - เวอร์เท็กซ์ คือ object ที่มีชื่อและ properties
 - เอดจ์ คือ ความสัมพันธ์ระหว่างเวอร์เท็กซ์

ตัวอย่างกราฟ

- กราฟ G ประกอบด้วย เวอร์เทกซ์ A, B, C, D, E, F, G และ เอดจ์ AB, AC, AD, BC, CD, EF, GE, FG

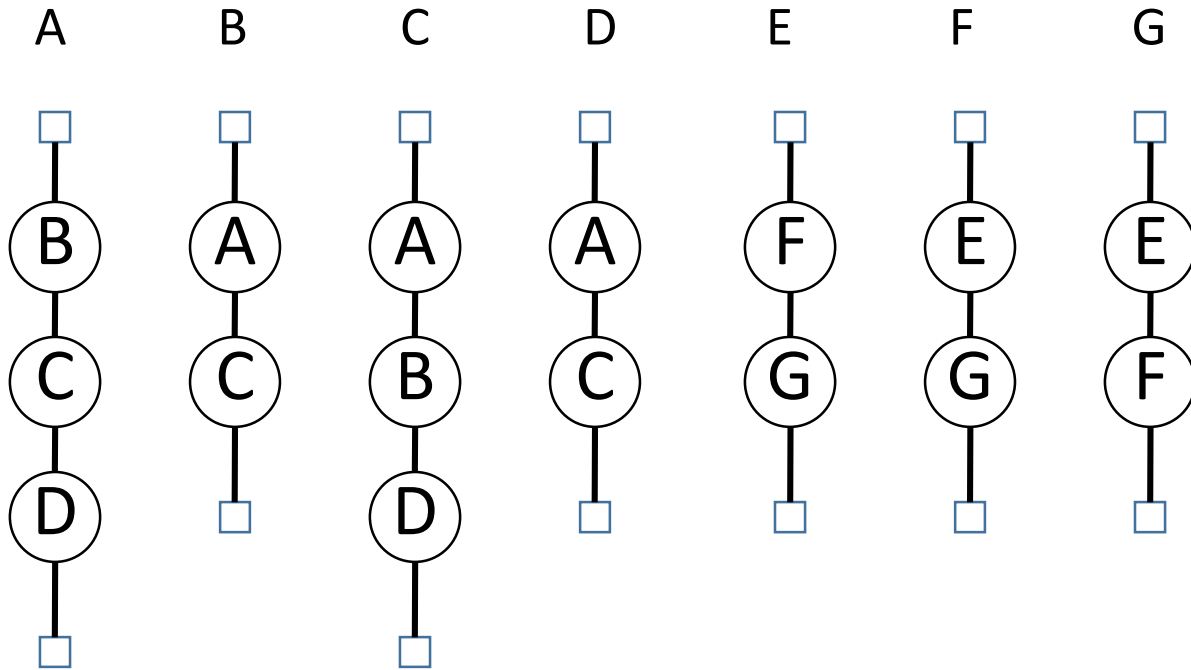


Representation

- Adjacency Matrix

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

- Adjacency linked list



```
#include <iostream>
using namespace std;
// stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};
```

```
class DiaGraph{
    // insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode*
head) {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;

        newNode->next = head; // point new node to current
head
        return newNode;
    }
    int N; // number of nodes in the graph
public:
    adjNode **head;           //adjacency list as array of pointers
```


// Constructor

```
DiaGraph(graphEdge edges[], int n, int N) {
```

```
    // allocate new node
```

```
    head = new adjNode*[N]();
```

```
    this->N = N;
```

```
    // initialize head pointer for all vertices
```

```
    for (int i = 0; i < N; ++i)
```

```
        head[i] = nullptr;
```

```
    // construct directed graph by adding edges to it
```

```
    for (unsigned i = 0; i < n; i++) {
```

```
        int start_ver = edges[i].start_ver;
```

```
        int end_ver = edges[i].end_ver;
```

```
        int weight = edges[i].weight;
```

```
        // insert in the beginning
```

```
        adjNode* newNode = getAdjListNode(end_ver, weight,
```

```
head[start_ver]);
```

```
            // point head pointer to new node
```

```
            head[start_ver] = newNode;
```

```
        }
```

```
    }
```

```
// Destructor
~DiaGraph() {
    for (int i = 0; i < N; i++)
        delete[] head[i];
    delete[] head;
}
};
```

```
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != nullptr) {
        cout << "(" << i << ", " << ptr->val
            << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}
```

```

int main()// graph implementation
{
    // graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
    };
    int N = 6;    // Number of vertices in the graph
    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);
    // construct graph
    DiaGraph diagraph(edges, n, N);
    // print adjacency list representation of graph
    cout<<"Graph adjacency list "<<endl<<"(start_vertex, end_vertex,
weight):"<<endl;
    for (int i = 0; i < N; i++)
    {
        // display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }
    return 0;
}

```

LAB 01: GRAPH INPUT

จงเขียนโปรแกรมเพื่ออ่านข้อมูลกราฟเข้าสู่ Adjacency matrix

1 เมื่อ input ถูกกำหนดให้อยู่ในรูปแบบ row-column

- เช่น 3x3

0	1	0
1	0	1
0	1	0

2 เมื่อ input ถูกกำหนดให้อยู่ในรูปแบบ Adjacency linked list

- เช่น 3 vertices 2

1	3
2	



LAB 01: GRAPH INPUT

3 เมื่อ input ถูกกำหนดให้อยู่ในรูปแบบ edges

เช่น 3 edges 1 2

2 3

1 3

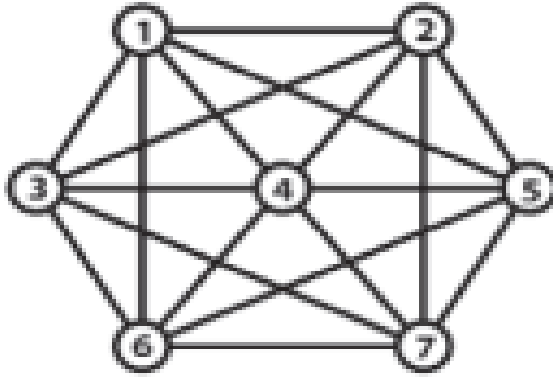
4 เมื่อ input ถูกกำหนดให้ตัวอักษรแทนชื่อ vertex และอยู่ในรูปแบบ edges

เช่น 3 edges A B

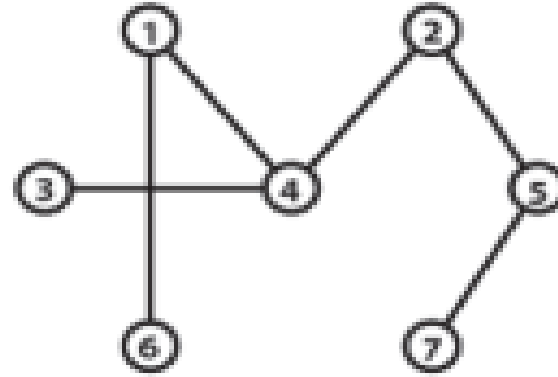
B C

A C

เลือกใช้โครงสร้างข้อมูลแบบไหนดี?



Dense



Sparse

- **dense graph** : จำนวนของ edges มีค่าใกล้เคียงกับจำนวนสูงสุดในกราฟ คือทุกคู่เวอร์เท็กซ์มี edge เชื่อมต่อ $= |V| * (|V| - 1)$
- **sparse graph** : จำนวนของ edges มีค่าน้อย เมื่อเทียบกับจำนวนสูงสุด
- Dense graph ใช้ matrix
- Sparse graph ใช้ list

เมื่อกราฟมีขนาดใหญ่มาก ๆ ทำอย่างไร?

- allocate an array dynamically (the limit is large)

```
int* a1 = new int[SIZE]; // SIZE limited only by OS/Hardware
```

- การใช้ vector <https://www.programiz.com/cpp-programming/vectors>
- การใช้ vector of vector

```
vector<vector<int>> vec{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9, 4 } };
```

- การใช้ map <https://www.programiz.com/cpp-programming/map>

เมื่อกราฟมีขนาดใหญ่มาก ๆ ทำอย่างไร?

- การใช้ map of vector ในกรณีที่โหนดแต่ละโหนดมีจำนวน edge ไม่มาก

`map<int, vector<Link>> AdjList;`

- การใช้ map of map ในกรณีของ weighted graph

map แรกหมายถึง outgoing nodes

ส่วน map ที่สองหมายถึง target node และ weight

`map<int, map<int, int>> Oulala;`



LAB 02: GRAPH Representation

1 จงเขียนโปรแกรมเพื่ออ่านข้อมูลกราฟเข้าสู่ Array of Vector เมื่อ input ถูกกำหนดให้อยู่ในรูปแบบ Adjacency list

- เช่น 3 vertices 2

1 3

2

2 จงเขียนโปรแกรมเพื่ออ่านข้อมูลกราฟเข้าสู่ Map of Vector เมื่อ input ถูกกำหนดให้อยู่ในรูปแบบ edges

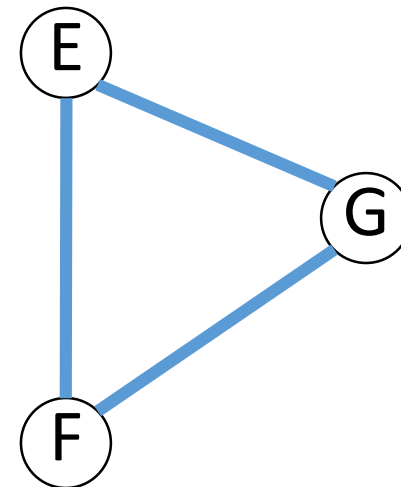
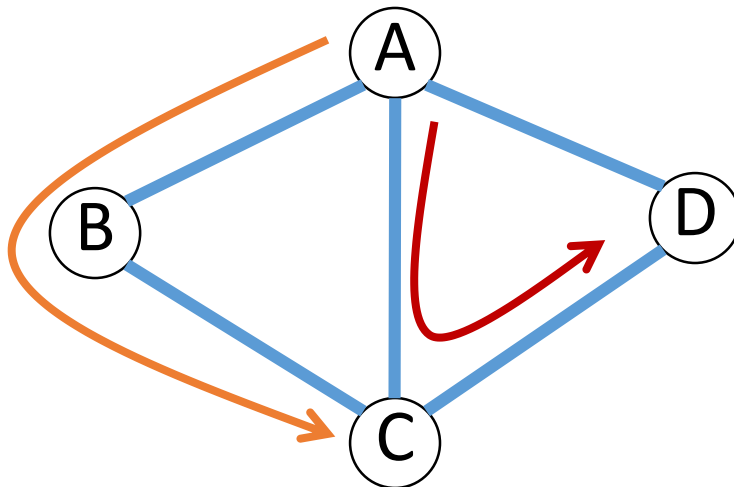
เช่น 3 edges 1 2

2 3

1 3

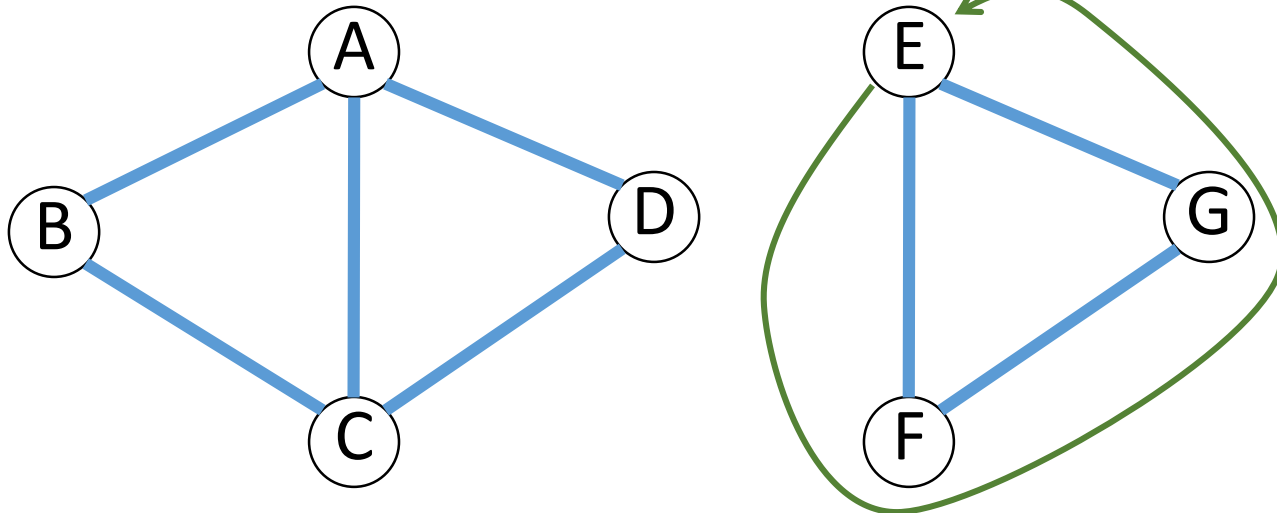
Path

- A path คือ ลิสต์ของเวอร์เท็กซ์ที่แต่ละเวอร์เท็กซ์มีเอดจ์เชื่อมต่อไปยังเวอร์เท็กซ์ลำดับถัดไป
- ABCD คือ path จาก A ไป D
- ACD คือ อีกหนึ่ง path จาก A ไป D



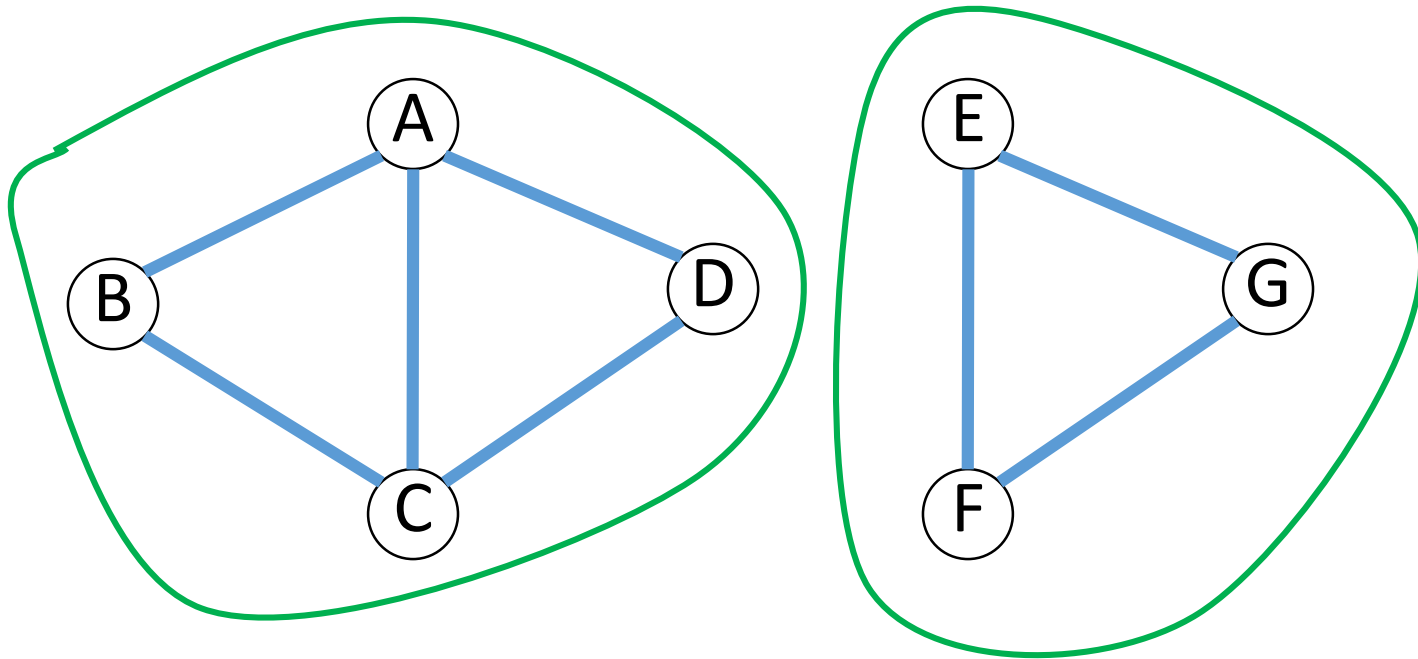
Cycle

- Path จะจัดเป็น simple path ถ้าใน path นั้นไม่มีเวอร์เท็กซ์ซ้ำ
- Cycle คือ simple path ยกเว้นเฉพาะโหนดแรกและโหนดสุดท้าย เช่น EFGE



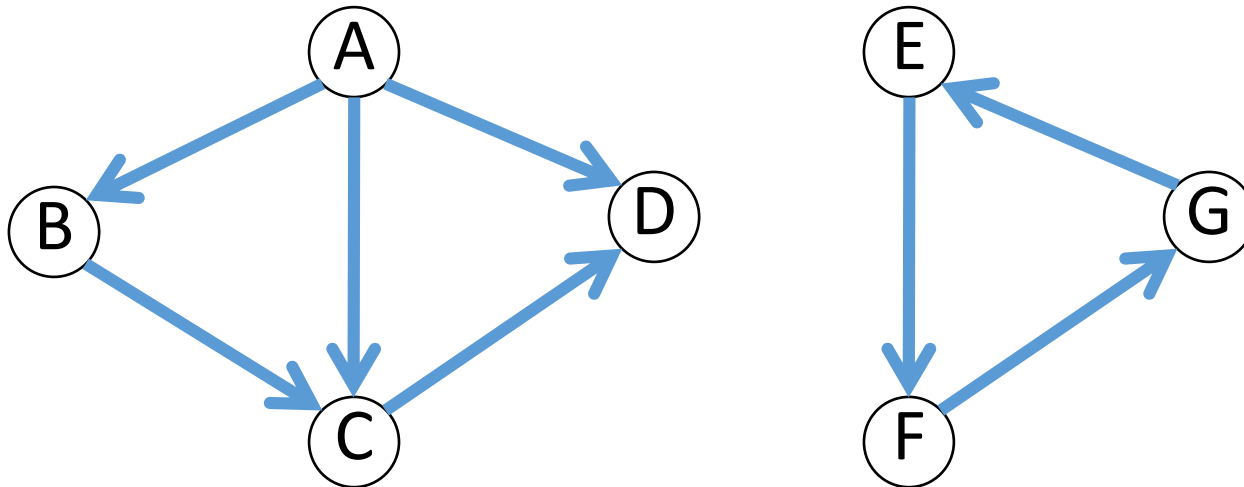
Connected

- ถ้ามี path จากทุก ๆ โหนด ไปยังโหนดต่าง ๆ ที่เหลือทั้งหมด
- Graph G ไม่ connected แต่ประกอบด้วย connected components 2 components



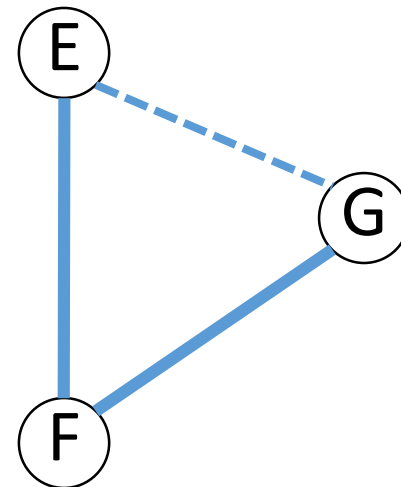
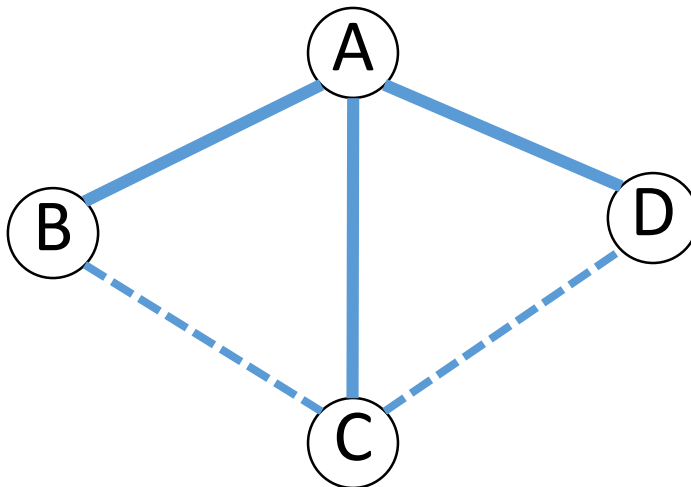
Directed Graph

- กราฟ G เป็น Directed Graph ประกอบด้วย เวอร์เท็กซ์ A, B, C, D, E, F, G และ เอ็ดจ์ AB, AC, AD, BC, CD, EF, GE, FG



Tree

- กราฟที่ไม่มี cycle คือ Tree
- กลุ่มของ trees ที่ไม่เชื่อมต่อกัน เรียกว่า forest
- Spanning tree คือ subgraph ที่มีครบทุกเวอร์เท็กซ์ แต่มีเอดจ์เฉพาะที่สามารถสร้างเป็น tree ได้ เช่น AB, AC, AD, EF, FG



2. Graph Traversal

- Depth First Search
- Breadth First Search

Depth-First Search

- จาก vertex เริ่มต้น ให้ visit vertex ที่อยู่ถัดไปและถัดไป จนกระทั่งไม่สามารถไปได้อีกแล้ว ก็จะทำการย้อนกลับมายัง vertex ที่ visited ไปแล้วก่อนหน้านี้ และหา vertex ถัดไปอีก เป็นเช่นนี้เรื่อยไป จนสุดท้าย algorithm จะสิ้นสุดการทำงานเมื่อทุก ๆ vertex ใน graph ถูกพิจารณาจนหมด
- ในการ traverse graph นั้นเราจะ visit เฉพาะ vertex ที่ยังไม่เคย visited มาก่อน ด้วยเหตุนี้เราจึงมักทำเครื่องหมายอะไรสักอย่างเพื่อให้รู้ว่า vertex นี้ได้ถูกพิจารณาแล้ว
- ในกรณีที่มี vertex ที่อยู่ถัดไปมากกว่า 1 ก็ต้องกำหนดวิธีการเลือกกว่าจะทำตามลำดับใด ให้มีรูปแบบที่แน่นอน เช่นตามลำดับตัวอักษร เป็นต้น

Depth-First Search

ALGORITHM $DFS(G)$

//Implements a depth-first search traversal of a given graph

//Input: Graph $G = (V, E)$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they've been first encountered by the DFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

$count \leftarrow 0$

for each vertex v in V **do**

if v is marked with 0

$dfs(v)$

dfs(v)

//visits recursively all the unvisited vertices connected to vertex *v* and

// assigns them the numbers in the order they are encountered via

// global variable *count*

count \leftarrow *count* + 1; mark *v* with *count*

for each vertex *w* in *V* adjacent to *v* **do**

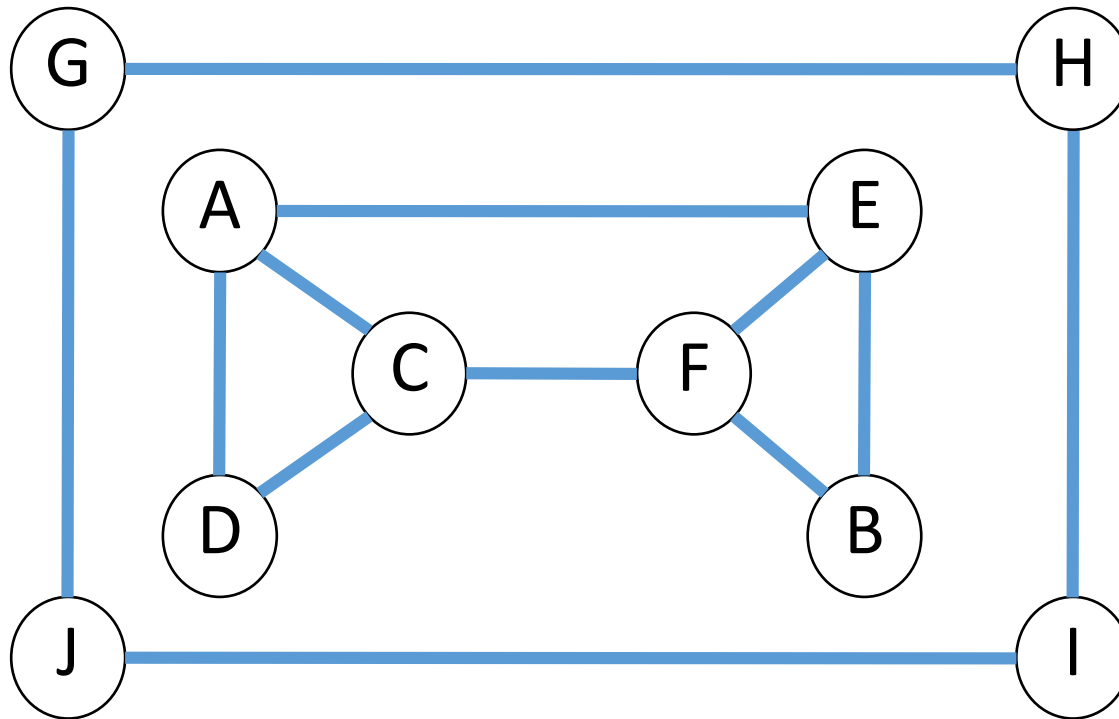
if *w* is marked with 0

dfs(w)

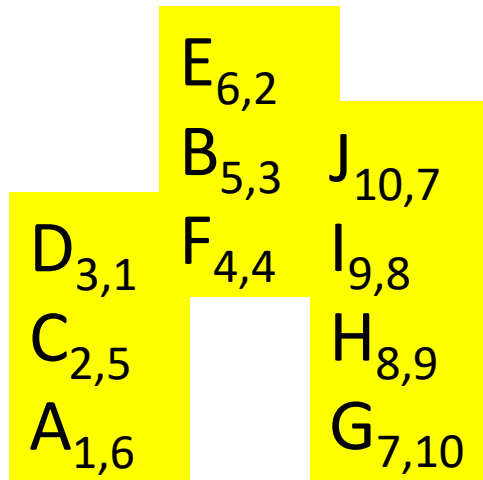
Depth-First Search

- การสร้าง *depth-first search forest* จะช่วยให้การทำ DFS สะดวกขึ้น (อย่างไร?)
- ให้ vertex ตั้งต้นเป็น root ของ tree
- เมื่อพบ vertex ที่ยังไม่ถูกพิจารณา ก็จะทำให้การเชื่อมต่อให้เป็นลูกของ vertex ที่มันถูกอ้างอิงมา โดยเราจะเรียก edge ที่เชื่อมต่อว่า *tree edge*
- Algorithm อาจตรวจพบ edge ที่ก่อให้เกิดการย้อนกลับไปยัง vertex ที่ได้รับการพิจารณาไปแล้วก่อนหน้านี้ นอกเหนือไปจาก parent ของมันเอง เราจะเรียก edge เหล่านี้ว่า *back edge* เพราะมันเชื่อมต่อ vertex กับ ancestor ของมัน
- Data structure ที่ใช้สร้าง DFS forest?

Example

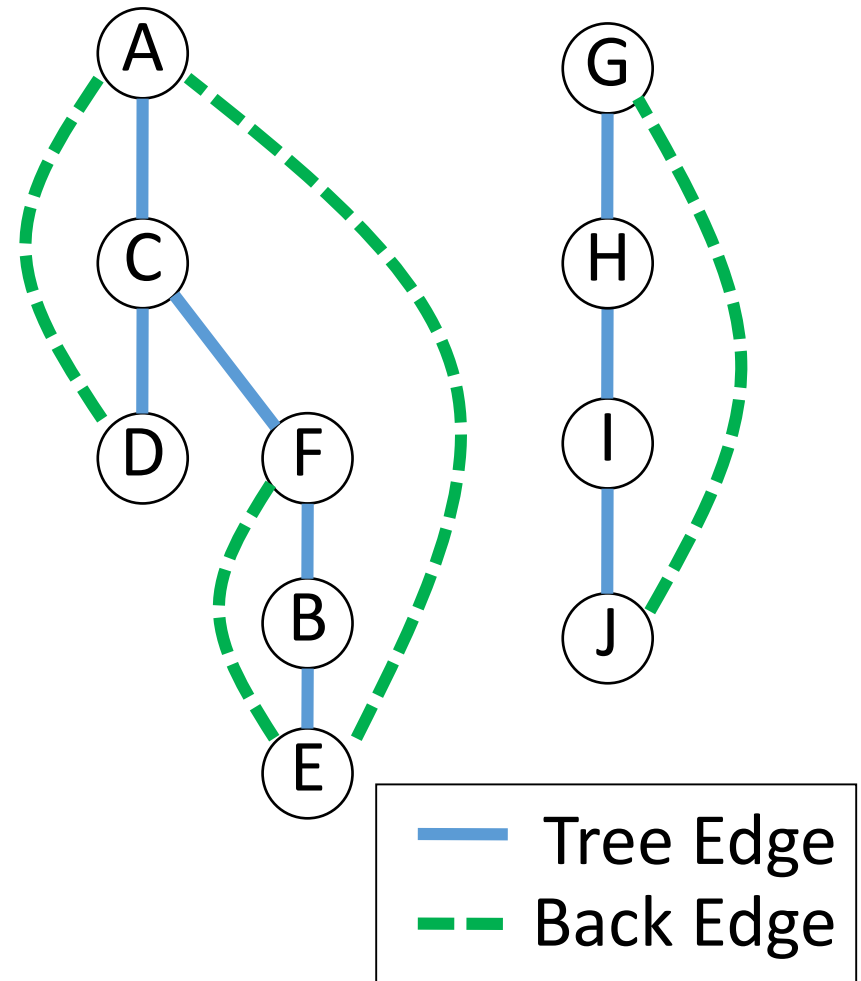


Example: DFS



Traversal's stack

- เลขตัวแรกหมายถึงลำดับที่การ **visit** เวอร์เท็กซ์ หรือลำดับการถูก **push** ลงสู่ **stack**
- เลขตัวที่สองคือ ลำดับที่ **vertex** เป็น **dead-end** หรือ ถูก **pop** ออกจาก **stack**



Analysis

- เวลาที่ใช้ในการ traverse tree จะขึ้นกับ data structure ที่ใช้ในการ represent graph
- ถ้าใช้ adjacency matrix representation, the traversal's time efficiency is in $\Theta(|V|^2)$
- ถ้าใช้ adjacency linked list representation, it is in $\Theta(|V| + |E|)$
- $|V|$ คือจำนวน vertex ใน graph
- $|E|$ คือจำนวน edge ใน graph

Application

- **checking connectivity of a graph:** เนื่องจาก DFS สิ้นสุดการทำงานเมื่อทุก ๆ vertex ที่เชื่อมต่อกับ vertex เริ่มต้นได้รับการพิจารณา การตรวจสอบ connectivity สามารถทำได้โดยใช้ DFS traversal ซึ่งเริ่มจาก vertex ใดก็ได้ เมื่อ algorithm สิ้นสุดการทำงานให้ตรวจสอบว่า ทุก ๆ vertex ใน graph ได้รับการพิจารณาแล้วหรือไม่ graph จะถือว่าเป็น connected ถ้าทุก ๆ vertex ถูก visited
- **checking acyclicity of a graph :** เราสามารถใช้ประโยชน์จาก DFS forest ได้ นั่นคือ ถ้า DFS forest ไม่มี back edges เลย graph ก็จะเป็นแบบ acyclic



LAB 03: Graph Traversal

1 จงเขียนโปรแกรมเพื่อสำรวจกราฟแบบ DFS

Breadth-First Search

- จาก vertex เริ่มต้น เราจะทำการ visit ทุก ๆ vertex ที่อยู่ติดกันให้หมดก่อนจะพิจารณา vertex ที่อยู่ไกลออกไป
- นั่นคือ เราพิจารณาตามระยะห่างจาก vertex เริ่มต้นนั่นเอง
- การ traverse จะสิ้นสุดเมื่อทุก ๆ vertex ในการได้รับการพิจารณา

Breadth-First Search

Algorithm BFS(G)

//Implements a breadth-first search traversal of a given graph

//Input: Graph $G = (V, E)$

//Output: Graph G with its vertices marked with consecutive integers

//in the order they've been visited by the BFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

$count \leftarrow 0$

for each vertex v in V **do**

if v is marked with 0

$bfs(v)$

Breadth-First Search

bfs(v)

//visits all the unvisited vertices connected to vertex v

// and assigns them the numbers in the order they are

// visited via global variable count

count \leftarrow *count* + 1;

mark *v* with *count* and initialize a queue with *v*

while the queue is not empty **do**

for vertex *w* in *V* adjacent to the front vertex **do**

if *w* is marked with 0

count \leftarrow *count* + 1;

 mark *w* with *count*;

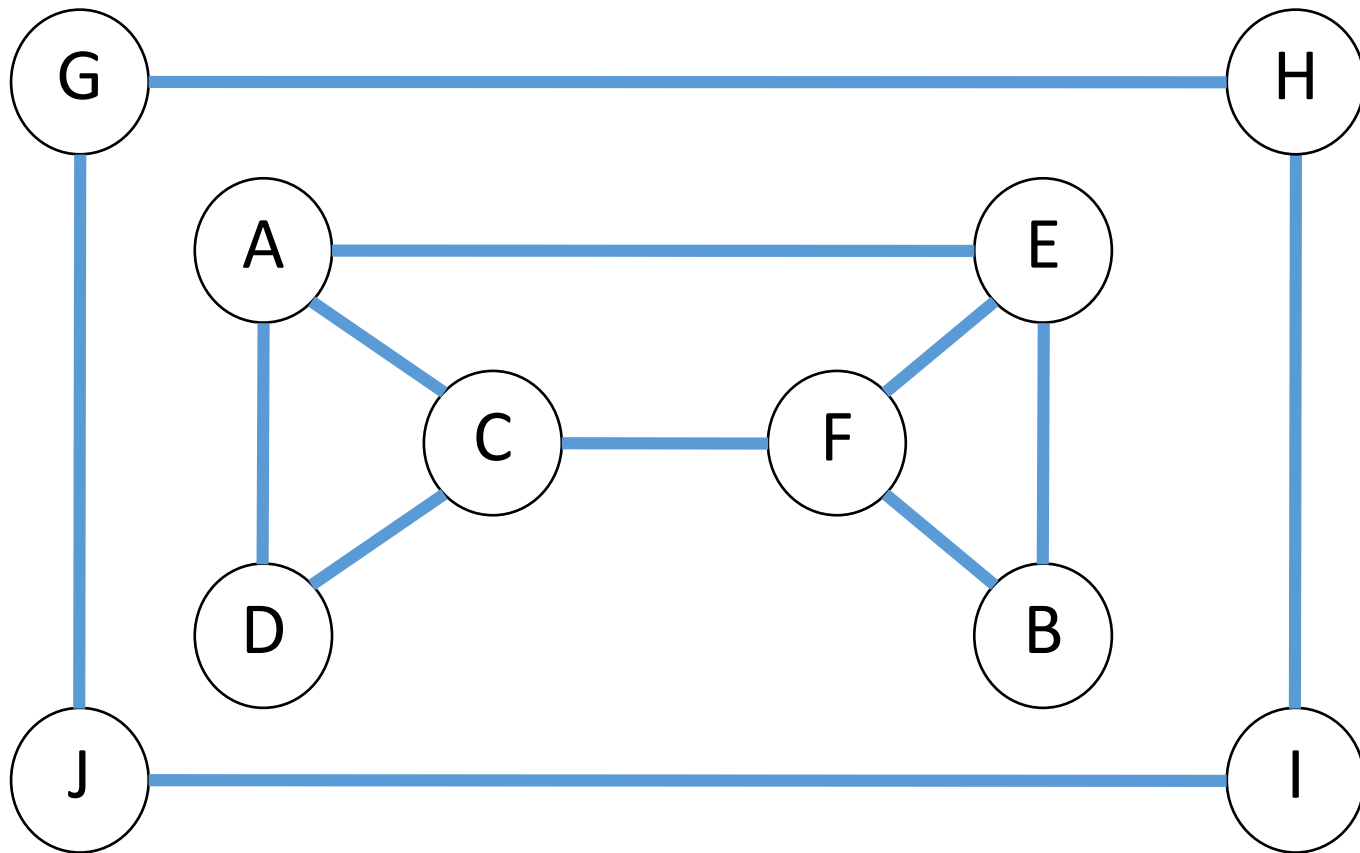
add w to the queue

remove the front vertex from the queue

Breadth-First Search

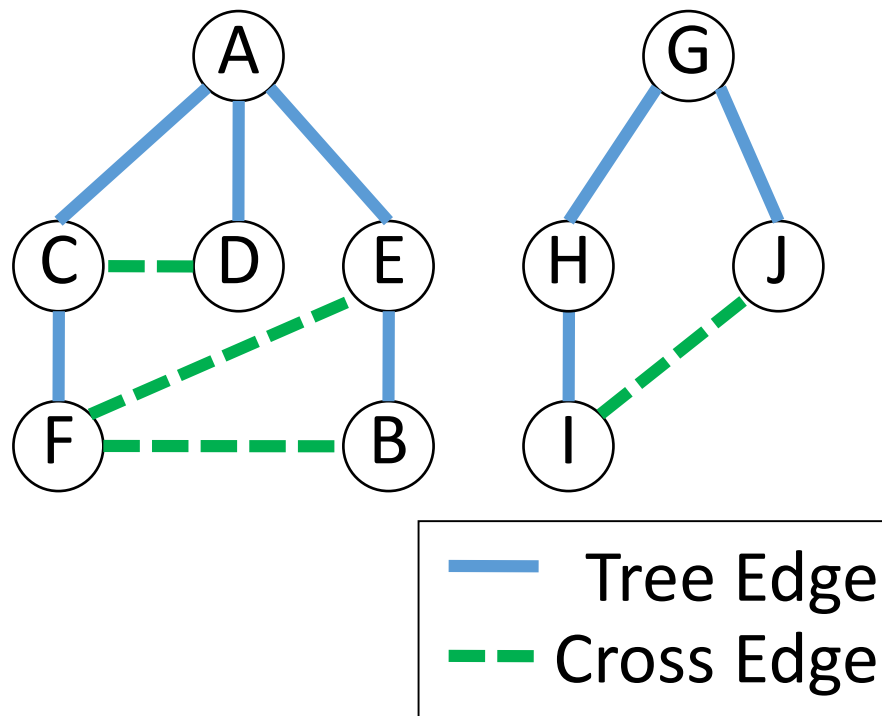
- เช่นเดียวกับ DFS traversal เรานิยมสร้าง *breadth-first search forest*.
- Root ของ tree ก็คือ vertex ตั้งต้น
- เมื่อพบ vertex ที่ยังไม่ถูกพิจารณา ก็จะเชื่อมต่อเข้าเป็นลูกของ vertex ที่มันถูกอ้างอิงผ่านมา และก็จะเรียก edge นั้นว่า *tree edge*.
- ถ้าพบ edge ที่เชื่อมต่อไปยัง vertex ที่เคย visited มาแล้วนอกเหนือจาก parent หรือ immediate predecessor ของตัวเอง เราจะเรียก edge นั้นว่า *cross edge*.
- Data structure ในการสร้าง BFS forest?

Example



Example : BFS

A_1 C_2 D_3 E_4 F_5 B_6 G_7 H_8 J_9 I_{10}



Traversal's queue

ตัวเลขหมายถึงลำดับของการ visit เวอร์เท็กซ์ ทั้งการเพิ่มและลบจากคิว

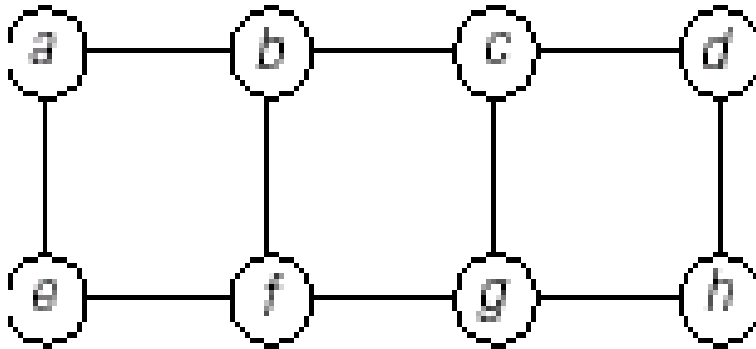
Analysis

- BFS มีค่าประสิทธิภาพเช่นเดียวกับ DFS
- $\Theta(|V|^2)$ ถ้าใช้ adjacency matrix representation
- $\Theta(|V| + |E|)$ ถ้าใช้ adjacency linked list representation.
- แต่ที่ต่างจาก DFS ก็คือ BFS มีลำดับของ vertex เพียงแบบเดียว เนื่องจากใช้ queue หรือ FIFO (first-in first-out)

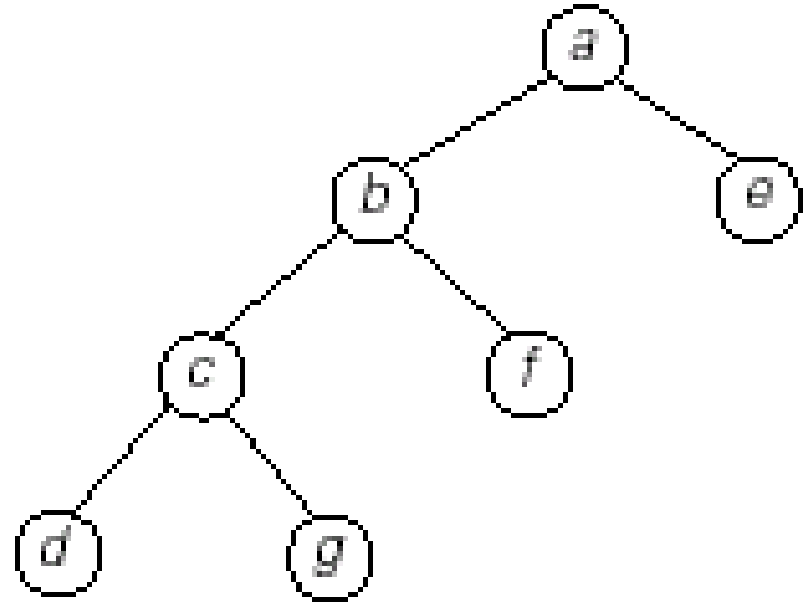
Application

- Check Connectivity of a graph
- Checking acyclicity of a graph
- Finding minimum-edge path

Finding Minimum-edge Path




Graph.



Part of its BFS tree that identifies the minimum-edge path from *a* to *g*.

Summary

	DFS	BFS
Data Structure	stack	Queue
No. of vertex orderings	2 ordering	1 ordering
Edge types (Undirected graph)	Tree and back edges	Tree and cross edges
Applications	Connectivity, acyclicity, articulation points	Connectivity, acyclicity, minimum edge paths
Efficiency for adjacency matrix	$\Theta(V ^2)$	$\Theta(V ^2)$
Efficiency for adjacency linked list	$\Theta(V + E)$	$\Theta(V + E)$



LAB 04: Graph Traversal: BFS

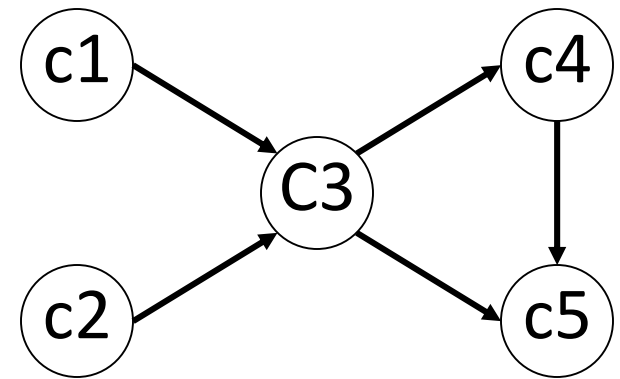
1 จงเขียนโปรแกรมเพื่อสำรวจกราฟแบบ BFS

3. Topological Sorting

- ให้อาลำดับของการพิจารณาหรือ visit vertex สอดคล้องกับทิศทางของ edge ใน directed graph
- นั่นคือ สำหรับทุก ๆ edge ใน digraph, vertex ที่เป็นจุดเริ่มต้นของ edge ต้องถูก visit ก่อน vertex ที่เป็นจุดสุดท้ายของ edge

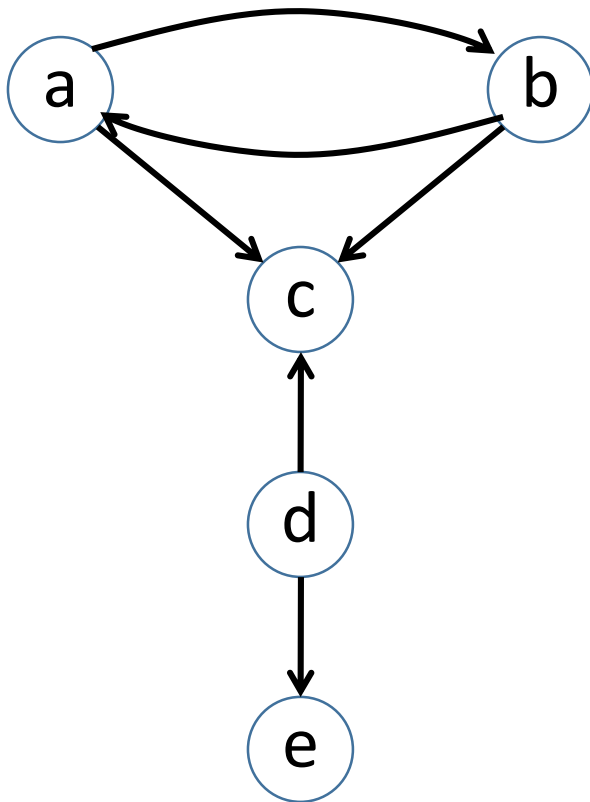
Topological Sorting- Example

- สมมติมีวิชาบังคับ 5 วิชา {C1, C2, C3, C4, C5} ซึ่งนักศึกษาสามารถเลือกลงทะเบียนอย่างไรก็ได้ ถ้าไม่ขัดกับความสัมพันธ์วิชาบังคับก่อนเรียน นั่นคือ
 - C1 และ C2 ไม่มีวิชาบังคับก่อน
 - C3 มีวิชาบังคับก่อน 2 วิชา คือ C1 และ C2
 - C4 มีวิชาบังคับก่อนคือ C3
 - C5 มีวิชาบังคับก่อน 2 วิชา คือ C3 และ C4.



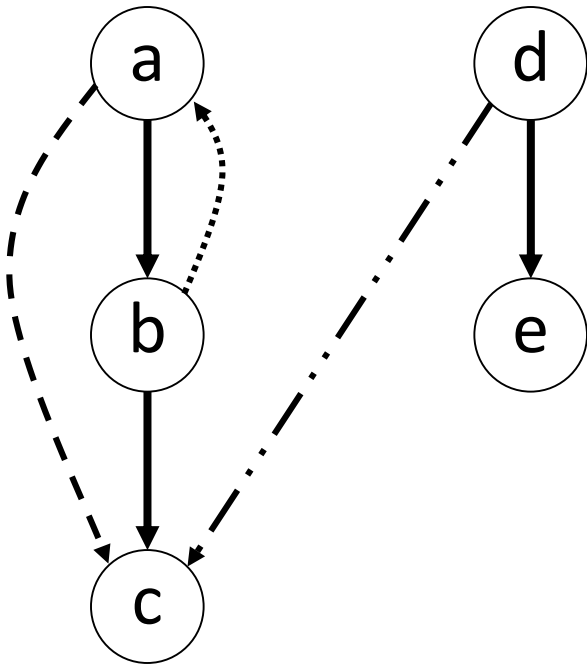
- ถ้านักศึกษาสามารถลงทะเบียนวิชาเหล่านี้ได้เพียง 1 วิชาต่อภาคการศึกษา นักศึกษาควรเลือกลำดับการลงทะเบียนวิชาเหล่านี้อย่างไร

Digraph



- A *directed graph*, or *digraph*, is a graph with directions specified for all its edges
- ความแตกต่างในการ represent undirected และ directed graphs มี 2 ข้อ คือ
- (1) adjacency matrix ของ directed graph ไม่จำเป็นต้องสมมาตร symmetric;
- (2) ในการใช้ adjacency linked lists, edge ใน directed graph 1 edge ใช้สร้าง node เพียง 1 node เท่านั้น ไม่ใช่ 2 nodes

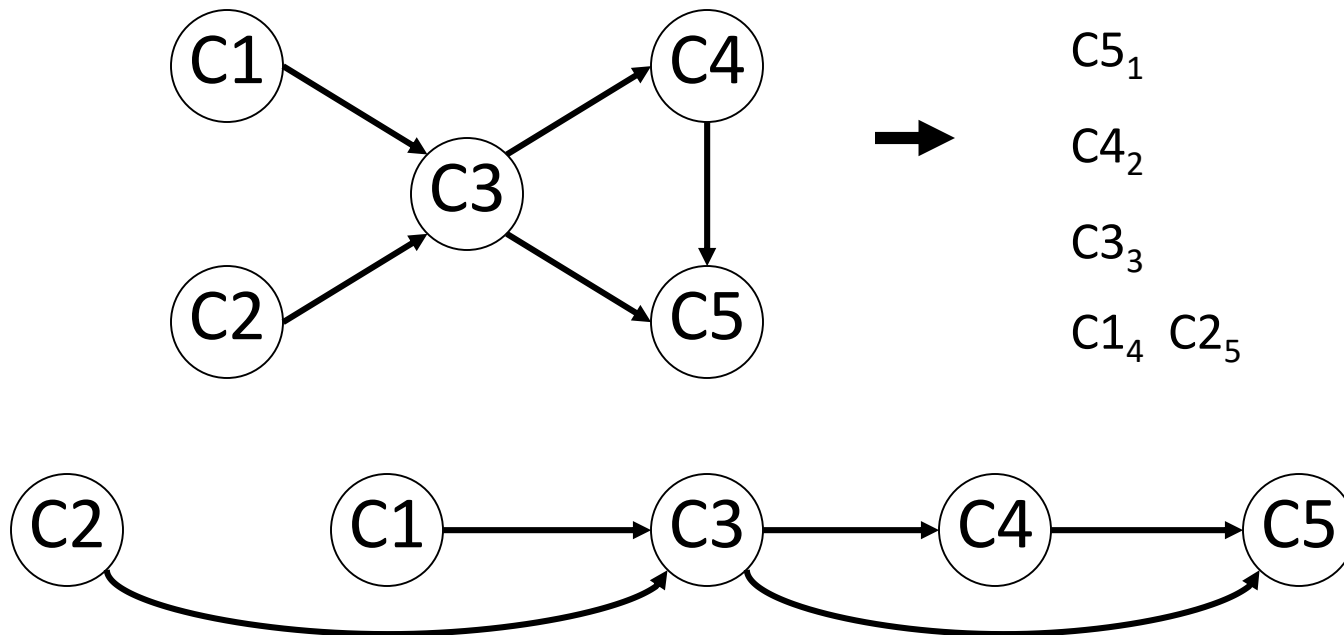
DFS Forest



- Tree ที่ได้จะประกอบด้วย edge 4 ประเภท
 - **tree edges** (ab, bc, de),
 - **back edges** (ba) from vertices to their ancestors,
 - **forward edges** (ac) from vertices to their descendants in the tree other than their children,
 - **cross edges** (dc), which are none of the aforementioned types.
- ถ้า DFS forest ของ digraph ไม่มี back edges, digraph นั้นจะเรียกว่า **directed acyclic graph** หรือ **dag**
- **Dag จำเป็นหรือไม่ในการแก้ปัญหา Topological Sorting ?**

Solving Topological Sorting Using DFS

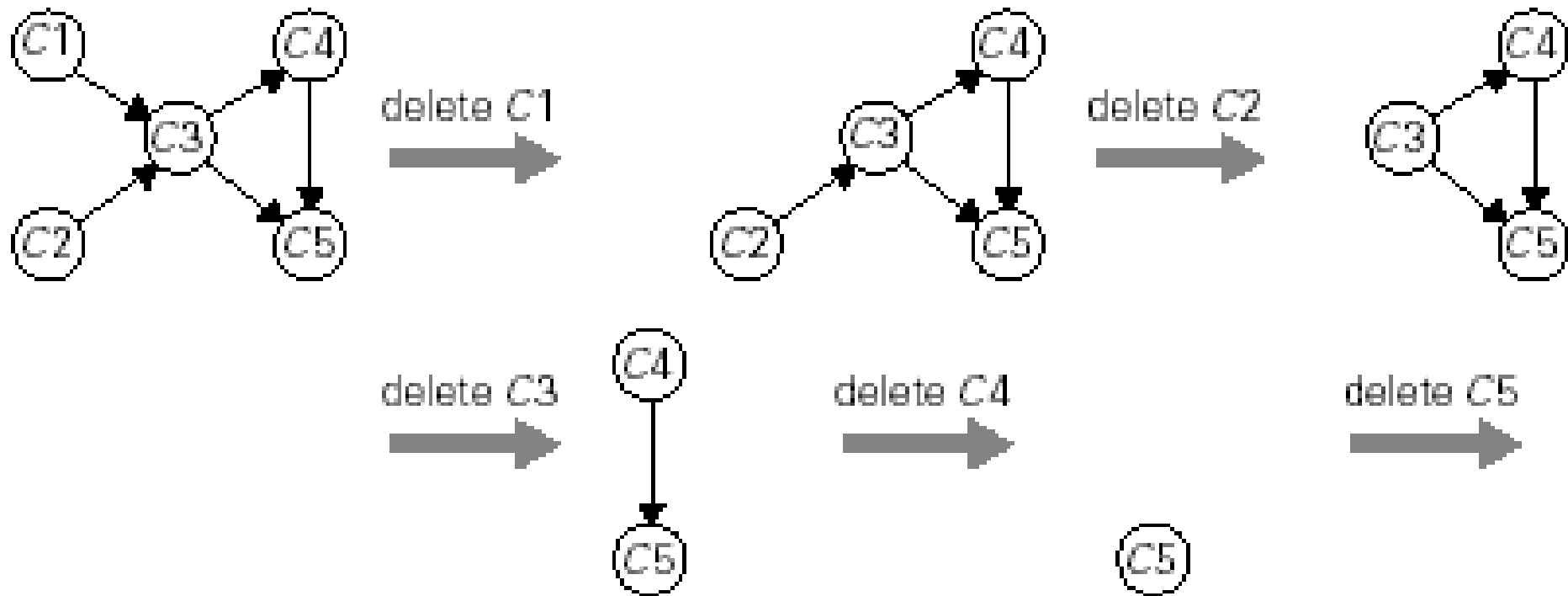
- Traverse DAG ด้วย DFS
- ให้พิจารณาลำดับของ vertex ที่กลายเป็น dead end หรือลำดับของการ pop ออกจาก stack นั้นเอง เมื่อทำมาเรียงย้อนลำดับ ก็จะได้ผลลัพธ์ของ topological sorting



Solving Topological Sorting Using Source

- A direct implementation of the decrease (by one)-and-conquer technique:
- source คือ vertex ที่ไม่มี incoming edge
- ให้หา source ใน digraph แล้ว delete source นั้นพร้อมทั้ง edge ทั้งหมดที่ออกจาก source นั้นทิ้ง เราก็จะได้ digraph ที่มีจำนวน vertex และ edge น้อยลง
- ทำกระบวนการนี้ซ้ำๆ ไปจนกว่าจะไม่เหลือ vertex ใน digraph

Solving Topological Sorting Using Source



The solution obtained is $C1, C2, C3, C4, C5$

Application

- ตรวจสอบว่า prerequisites ของ tasks ใน project ไม่มี contradiction ก่อนการทำ scheduling tasks ซึ่งอาจต้องใช้ algorithm อื่นๆ เข้ามาช่วยในการทำ scheduling



LAB 05: Topological Sorting

1 จงเขียนโปรแกรมเพื่อหาคำตอบ topological sorting