Graph algorithms

- ให้ลองทำข้อ **Graph Connectivity** ก่อน

Finding Connected Component

- DFS และ BFS ไม่ได้มีประโยชน์สำหรับการท่องไปในกราฟเท่านั้น
- ทั้งสองอัลกอริทึมนี้ยังสามารถถูกใช้ในการแก้ปัญหาอื่นๆ ได้อีกด้วย
- ปัญหาแรกต่อไปนี้ สามารถแก้ได้ด้วย DFS หรือ BFS

- เราใช้ประโยชน์จากความจริงที่ว่า ในการเรียก dfs(u) หรือ bfs(u) หนึ่งครั้งนั้นจะไปเยี่ยมโหนดที่เชื่อมต่อกันกับ u นั่นทำให้เราเอาไว้หา หรือนับจำนวนของก้อนที่ติดกัน (connected component) ใน undirected graph ได้
- เราสามารถใช้ code ต่อไปนี้ เพื่อเริ่ม dfs หรือ bfs ใหม่จากโหนดที่เหลือที่ยังไม่ได้ visit และเวลาทำงานก็ยังเป็น O(V+E)

9

1 1 0

3 0 0 2 0 3 0

2 1 0 3 0

3 1 0 2 0 4 0

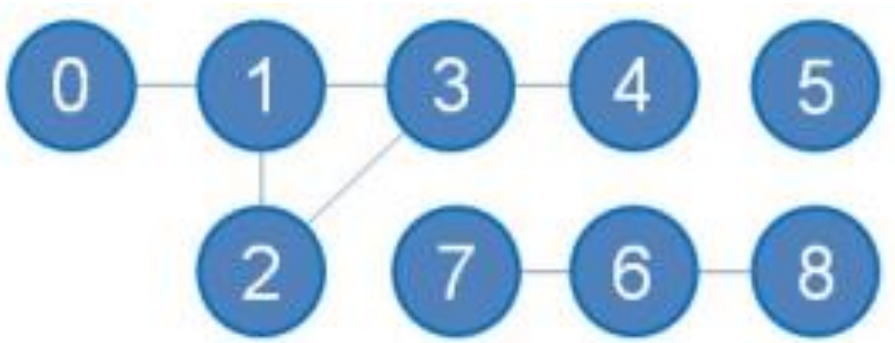
1 3 0

0

2 7 0 8 0

1 6 0

1 6 0



DFS(แบบที่ 2 Adjacency list แบบ weight)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
#define DFS_WHITE -1
#define DFS_BLACK 1
vector<vii> AdjList;
vi dfs_num;     // this variable has to be global, we cannot put it in recursion
int numCC;
void dfs(int u) {
  printf(" %d", u);                                    // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                          // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)             // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
  }
}
```

DFS(แบบที่ 2 ต่อ)

```
int main()
{
  int V, total_neighbors, id, weight;
  scanf("%d", &V);
  AdjList.assign(V, vii());
// assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < V; i++) {
    scanf("%d", &total_neighbors);
    for (int j = 0; j < total_neighbors; j++) {
      scanf("%d %d", &id, &weight);
      AdjList[i].push_back(ii(id, weight));
    }
  }
```

```
 numCC = 0;
  dfs_num.assign(V, DFS_WHITE);
// this sets all vertices' state to DFS_WHITE
  for (int i = 0; i < V; i++)
// for each vertex i in [0..V-1]
    if (dfs_num[i] == DFS_WHITE){
// if that vertex is not visited yet
      printf("Component %d:", ++numCC);
      dfs(i);
      printf("\n");
    }
  printf("There are %d connected components\n", numCC);
  return 0;
}
```

https://bit.ly/2SE3szF

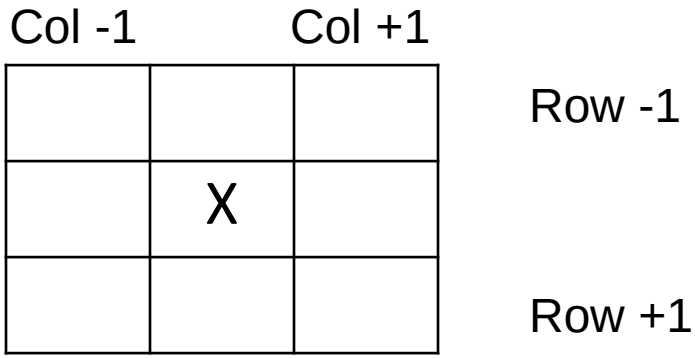
Flood Fill–labeling/ coloring the connected component

- DFS (หรือ BFS) สามารถถูกใช้ในจุดประสงค์อื่นนอกจากการหาหรือนับจำนวน connected component
- ต่อไปจะแสดงการปรับจูน O(V+E) dfs(u) (เราสามารถใช้ BFS ได้) เพื่อใช้ในการ label เป็นการกำหนดป้ายชื่อให้กับโหนดซึ่งใน CS มักจะเรียกว่าการทาสี ‘to color’ และนับขนาดของแต่ละ component
- ต่อไปเป็น code ที่นิยมเรียกว่า ‘flood fill’ และมันจะใช้กับ implicit graph เช่น 2D grid

- ให้ลองทำข้อมหานครกะลาแลนด์ก่อน

- ในข้อนี้ implicit graph เป็น 2D grid ที่โหนดคือช่องใน grid และเส้นเชื่อมคือการเชื่อมกันระหว่างช่องตตัวเองกับช่องในทิศ S/SE/E/NE/N/NW/W/SW
- ส่วน W แทนที่ดินน้ำท่วมรวดเร็ว และ L แทนที่ดินปกติ
- พื้นที่ที่น้ำท่วมอย่างรวดเร็วถูกนิยามด้วย ช่องที่ต่อกันด้วยอักขระ ‘W’
- เราสามารถระบุ(พร้อมทั้งนับ) พื้นที่ที่น้ำท่วมอย่างรวดเร็วได้โดยใช้ Floodfill
- เหมือนเราเปิดน้ำเริ่มจากจุดนั้นแล้วถามว่าน้ำไหลไปถึงใครได้บ้าง

- เราจะเริ่มจากตำแหน่ง x แล้วจะไปสำรวจตำแหน่งรอบๆ x ทำอย่างไร
- เราจะมีตัวแปรสองตัวมาช่วย dr, dc (r มาจากแถว c มากจากคอลัมน์)



- int dr[] = {1,1,0,−1,−1,−1, 0, 1}; // trick to explore an implicit 2D grid
  int dc[] = {0,1,1, 1, 0,−1,−1,−1}; // S,SE,E,NE,N,NW,W,SW neighbors
- พอเริ่มที่จุด x ก็ถามรอบตัว 8 ทิศว่าไปทางไหนได้บ้าง ตอนที่ถามตัวแรกก็ไปถามต่อ คล้ายๆ กับ DFS

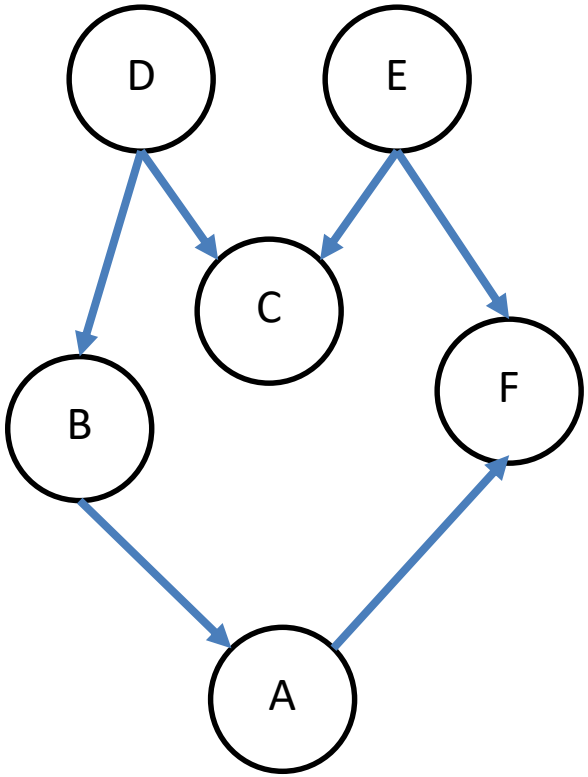
```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};
int dc[] = {0,1,1, 1, 0,-1,-1,-1};
int floodfill(int r, int c, char c1, char c2) {
// returns the size of CC
      if (r < 0 || r >= R || c < 0 || c >= C) return 0;
// outside grid
      if (grid[r][c] != c1) return 0;
// does not have color c1
      int ans = 1; // adds 1 to ans because vertex (r, c) has
c1 as its color
      grid[r][c] = c2; // now recolors vertex (r, c) to c2 to
avoid cycling!
      for (int d = 0; d < 8; d++)
            ans += floodfill(r + dr[d], c + dc[d], c1, c2);
      return ans; // the code is neat due to dr[] and dc[]
}
```

Topological sort (Directed Acyclic Graph, DAG)

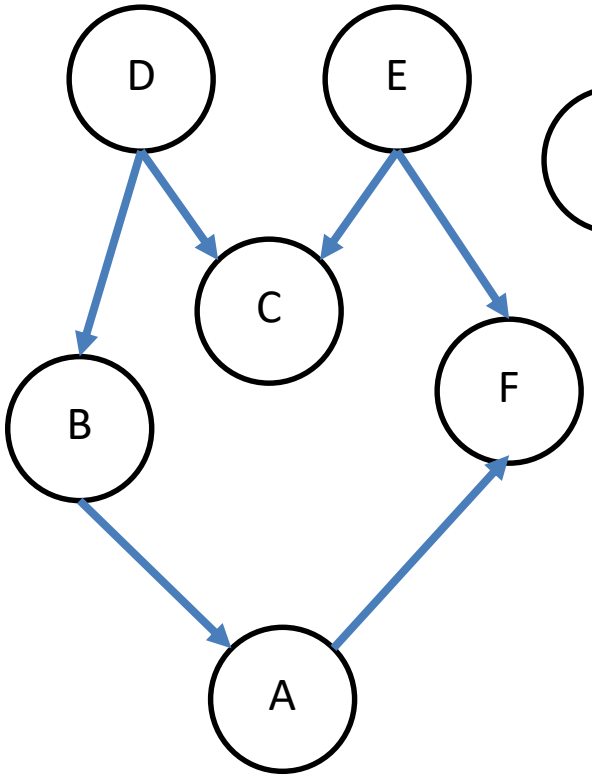
- Topological sort หรือ Topological ordering ของ Directed Acyclic Graph เป็นการเรียงลำดับเชิงเส้นตรงของโหนดใน Directed Acyclic Graph เพื่อให้ได้ว่าโหนด u มาก่อนโหนด v ถ้ามีเส้นเชื่อม (u->v) ใน DAG

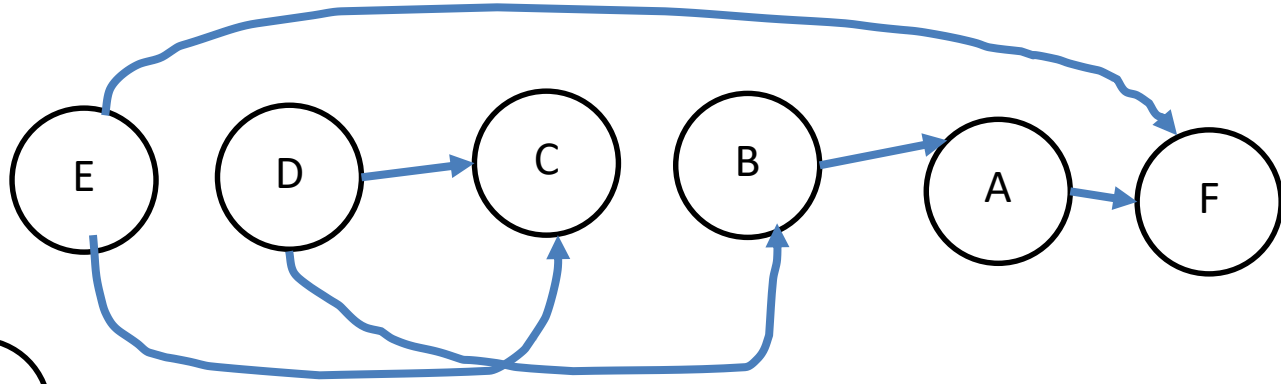
- ทั้งนี้ทุกๆ DAG จะมีอย่างน้อยหนึ่งหรืออาจจะมีมากกว่าหนึ่ง topological sort

- เราจะเรียงโหนดเป็นเส้นอย่างไรให้ไม่มีเส้นวิ่งย้อนกลับ



- เราจะเรียงโหนดเป็นเส้นอย่างไรให้ไม่มีเส้นวิ่งย้อนกลับ





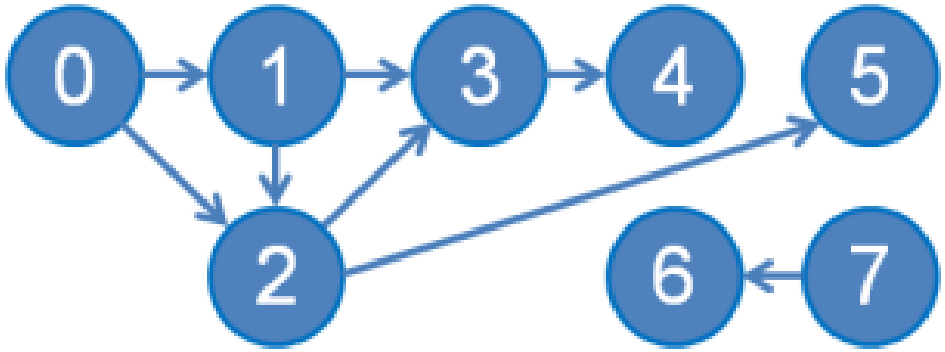
- application อย่างหนึ่งของ topological sorting คือการหาลำดับที่เป็นไปได้ของรายวิชาที่เรียนของนักศึกษาเพื่อที่จะทำให้จบตามเงื่อนไข แต่ละรายวิชานั้นมี pre-requisites ที่จะต้องผ่านก่อน ซึ่งตัว pre-requisites นี้จะต้องไม่เป็น cycle ดังนั้นเราสามารถมองได้ว่าเป็น DAG
- Topological sorting pre-requisites ของรายวิชาจะได้รายการเชิงเส้นของรายวิชาที่ต้องเรียนก่อนหลังที่ไม่ผิดเงื่อนไขของ รายวิชา pre-requisites
- จริงๆ แล้วมีอัลกอริทึมสำหรับ topological sorting หลายตัว วิธีหนึ่งที่ง่ายคือปรับ DFS

- หลักการคือ เราก็ท่องไปในกราฟ ถ้าโหนดไหนทำครบแล้วเราก็เก็บไว้ใน list
- เมื่อครบทุกโหนด เราก็ Print list ในลำดับย้อนกลับ

```
vi ts; // global vector to store the toposort in reverse order
void dfs2(int u) { // different function name compared to the
original dfs
        dfs_num[u] = VISITED;
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
                ii v = AdjList[u][j];
                if (dfs_num[v.first] == UNVISITED)
                        dfs2(v.first);
        }
        ts.push_back(u);
}
```

```
// inside int main()
        ts.clear();
        memset(dfs_num, UNVISITED, sizeof dfs_num);
        for (int i = 0; i < V; i++)
                if (dfs_num[i] == UNVISITED)
                        dfs2(i);
// alternative, call: reverse(ts.begin(), ts.end()); first
        for (int i = (int)ts.size() - 1; i >= 0; i--)
                printf(" %d", ts[i]);
        printf("\n");

// For the sample graph, the output is like this:
// 7 6 0 1 2 5 3 4 (remember that there can be >= 1 valid
toposort!!!!)
```



- ลองกับกราฟนี้ดู

- ทำไมการเอาโหนดไปไว้ด้านท้ายใน DFS นั่นเพียงพอในการหา topological sort ใน DAG

- ก่อนหน้านี้ Topological sort ใช้ DFS
- แต่ก็มี Kahn's algorithm ที่เป็นการ modify BFS

```
enqueue vertices with zero incoming degree into a (priority) queue Q;
while (Q is not empty) {
        vertex u = Q.dequeue();
        put vertex u into a topological sort list;
        remove this vertex u and all outgoing edges from this vertex;
        if such removal causes vertex v to have zero incoming degree
                Q.enqueue(v);
}
```