HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCTURES AND ALGORITHMS

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

WEEK 14 : GRAPH (PART 1)

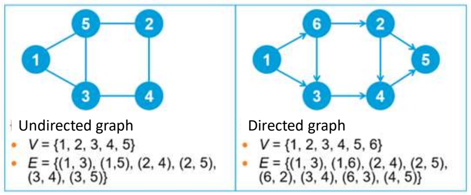
ONE LOVE. ONE FUTURE.

CONTENT

- Recall the basic concepts of graphs
- Graph representation data structure
- Graph traversal
- Find connected components
- Check bipartite graphs

RECALL THE BASIC CONCEPTS OF GRAPHS

- A graph is a structure consisting of nodes (also called vertices) and connections between nodes (also called edges or arcs).
- Graph notation: $G = (V, E)$, where V is a set of nodes and E is a set of edges (arcs)
- $(u,v) \in E$: *v is adjacent to u.*



RECALL THE BASIC CONCEPTS OF GRAPHS

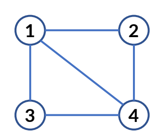
- Given $G = (V, E)$ is a graph
  - The path from vertex s to vertex t on the graph is a sequence of vertices $v_1, v_2, \ldots, v_k$, *where*
    - $s = v_1$, $t = v_k$
    - $(v_i, v_{i+1}) \in E$
  - A cycle is a path in which the first and last vertices coincide
  - An undirected graph G is called connected if there is always a path between any two vertices of G

GRAPH REPRESENTATION DATA STRUCTURE

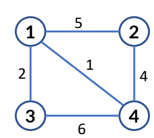
- **Adjacency matrix, weight matrix**
- **Adjacency list**
- **Edge list**

GRAPH REPRESENTATION DATA STRUCTURE

- **Adjacency matrix, weight matrix**



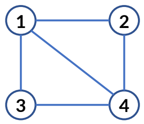
	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0



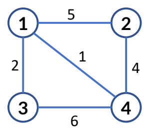
	1	2	3	4
1	0	5	2	1
2	5	0	0	4
3	2	0	0	6
4	1	4	6	0

GRAPH REPRESENTATION DATA STRUCTURE

- **Adjacency list**
  - **A[v]: list of vertices (or edges/arcs for weighted graphs) adjacent to vertex v**



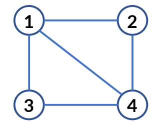
- A[1] = [2, 3, 4]
- A[2] = [1, 4]
- A[3] = [1, 4]
- A[4] = [1, 2, 3]



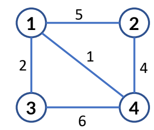
- A[1] = [(1,2,5), (1,3,2), (1,4,1)]
- A[2] = [(1,2,5), (2,4,4)]
- A[3] = [(1,3,2), (3,4,6)]
- A[4] = [(1,4,1), (2,4,4), (3,4,6)]

GRAPH REPRESENTATION DATA STRUCTURE

- **Edge list**
  - **E: list of edges/arcs of the graph**



Edge list: E = [(1,2), (1,3), (1,4), (2, 4), (3,4)]



List of edges with weights on edges
E = [(1,2, 5), (1,3, 2), (1,4, 1), (2, 4, 4), (3,4, 6)]

GRAPH TRAVERSAL

- Traverse the graph in depth: visit the vertices of the graph, each vertex exactly once
- A[v]: list of vertices adjacent to v

```
DFS(u, A) {// depth first search from u
  visited[u] = true; //Visit u;
  for v in A[u] do {
     if visited[v] = false then {
        DFS(v, A);
     }
  }
}
```

```
DFS(G = (V, A)){ // depth first search on G
  for v in V do visited[v] = false;
  for v in V do {
    if visited[v] = false then {
       DFS(v, A);
    }
  }
}
```


GRAPH TRAVERSAL

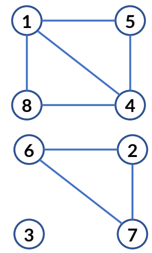
- Traverse the graph breadthwise: visit the vertices of the graph, each vertex exactly once
- A[v]: list of vertices adjacent to v

```
BFS(G = (V, A)){ // breadth first search on G
  for v in V do visited[v] = false;
  for v in V do {
    if visited[v] = false then {
       BFS(v, A);
    }
  }
}
```

```
BFS(u, A) {// breadth first search from u
  Q = Empty;
  Q.push(u);  visited[u] = true; // visit u
  while Q not empty do {
    v = Q.pop();
    for x in A[v] do
      if visited[x] = false then {
         Q.push(x); visited[x] = true; // visit x
      }
    }
}
```

FIND CONNECTED COMPONENTS

- Description of the problem
  - Given an undirected graph G = (V, A) in it
    - V is the set of vertices
    - A is an adjacency list structure: A[v] is a list of vertices adjacent to v
  - Need to find connected components of G
- Algorithm:
  - Apply depth-first search (or breadth-first search) on G
  - DFS(u) allows visiting all vertices belonging to the same connected component with u



FIND CONNECTED COMPONENTS

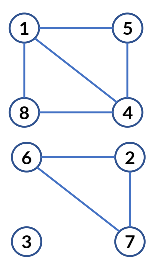
- nbCC: number of connected components of G
- C[v]: index (running from 1 to nbCC) of the connected component containing vertex v

```
DFS(u, A) {// DFS from u
  visited[u] = true; //Visit nh u;
  C[u] = nbCC;
  for v in A[u] do {
     if visited[v] = false then {
        DFS(v, A);
     }
  }
}
```

```
DFS(G = (V, A)){ // DFS on G
  for v in V do visited[v] = false;
  nbCC = 0;
  for v in V do {
    if visited[v] = false then {
       nbCC = nbCC + 1;
       DFS(v, A);
    }
  }
}
```


FIND CONNECTED COMPONENTS

- Illustrate with C language
- Data
  - Line 1: contains two positive integers n and m representing the number of vertices and edges, respectively
  - Line i + 1 (i = 1, 2, ..., m): contains 2 positive integers u and v which are the 2 endpoints of the ith edge
- Result
  - Write a list of vertices of each connected component found on one line



stdin	stdout
8 8 1 4 1 5 1 8 2 6 2 7 4 5 4 8 6 7	C[1]: 1 4 5 8 C[2]: 2 6 7 C[3]: 3

FIND CONNECTED COMPONENTS

```
#include <stdio.h>
#define N 100001
typedef struct Node{
  int id;
  struct Node* next;
}Node;
int n,m; // #nodes and #edges of G
Node* A[N]; // A[v]: pointer to an adj list
int nbCC; // #connected components
int C[N]; // C[v]: connected component
         //containing v
```

```
Node* makeNode(int id){
         Node* p = (Node*)malloc(sizeof(Node));
         p->id = id; p->next = NULL;
         return p;
}
Node* insert(int id, Node* h){
         Node* p = makeNode(id);
         p->next = h;
         return p;
}
```

FIND CONNECTED COMPONENTS

```
void input(){
  scanf("%d %d",&n,&m);
  for(int v = 1; v <= n; v++) A[v] = NULL;
  for(int i = 1; i <= m; i++){
    int u,v; scanf("%d%d",&u,&v);
    A[u] = insert(v,A[u]);
    A[v] = insert(u,A[v]);
  }
}
```

```
void DFS(int u){
  C[u] = nbCC;
  for(Node* p = A[u]; p != NULL; p = p->next){
    int v = p->id;
    if(C[v] == -1){
      DFS(v);
    }
  }
}
```


FIND CONNECTED COMPONENTS

```
void DFSG(){
  for(int u = 1; u <= n; u++) C[u] = -1;
  nbCC = 0;
  for(int u = 1; u <= n; u++){
    if(C[u] == -1){
      nbCC = nbCC + 1;
      DFS(u);
    }
  }
}
```

```
void printCC(){
  for(int k = 1; k <= nbCC; k++){
    printf("C[%d]: ",k);
    for(int v = 1; v <= n; v++){
      if(C[v] == k) printf("%d ",v);
    }
    printf("\n");
  }
}
```


FIND CONNECTED COMPONENTS

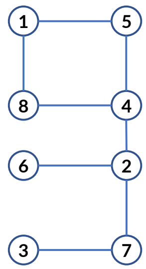
```
int main(){
  input();
  DFSG();
  printCC();
  return 0;
}
```


CHECK BIPARTITE GRAPHS

- Problem description
  - Given an undirected graph G = (V, A) in it
    - V is the set of vertices
    - A is an adjacency list structure: A[v] is a list of vertices adjacent to v
  - Check if G is a bipartite graph or not?

CHECK BIPARTITE GRAPHS

- Data
  - Line 1: contains two positive integers n and m representing the number of vertices and edges, respectively
  - Line i + 1 (i = 1, 2, ..., m): contains 2 positive integers u and v which are the 2 endpoints of the Ith edge
- Result
  - Write 1 if the graph is bipartite and 0 otherwise



stdin	stdout
8 8 1 5 1 8 2 4 2 6 2 7 3 7 4 5 4 8	1

CHECK BIPARTITE GRAPHS

- Algorithm:
  - Apply breadth-first search on G
  - d[v]: level (path length from starting vertex to v in BFS) of vertex v
  - BFS(u): visit all vertices with the same connected component with u
    - If the connected component containing u is a bipartite graph, then vertices v with even d[v] will be on the side containing u, and vertices v with odd d[v] will be on the other side.
    - If it is detected that edge (u,v) has d[u] and d[v] with the same parity, then the given graph is not a bipartite graph.

CHECK BIPARTITE GRAPHS - PSEUDOCODE

```
BFS(u) {
  Q = empty queue;  Q.push(u);  d[u] = 0;
  while Q not empty do {
    v = pop();
    for x in A[v] do {
      if(d[x] > -1){
          if  d[v] + d[x]  is even then return false;
      }
      else {  d[x] = d[v] + 1;   Q. push(x);  }
    }
  }
  return true;
}
```

```
solve(){
  for u = 1 to n do  d[u] = -1;
  for u = 1 to n do if(d[u] = -1){
    if(BFS(u) = false) {
       return false;
    }
  }
  return true;
}
```


CHECK BIPARTITE GRAPHS – COMPLETE CODE

```
#include <stdio.h>
#include <stdlib.h>
#define N 100001
typedef struct Node{
  int id;
  struct Node* next;
}Node;
Node* makeNode(int id){
  Node* p = (Node*)malloc(sizeof(Node));
  p->id = id; p->next = NULL;
  return p;
}
```

```
Node* head;
Node* tail;
void initQueue(){
  head = NULL; tail = NULL;
}
int queueEmpty(){
  return head == NULL && tail == NULL;
}
void push(int id){
  Node* p = makeNode(id);
  if(queueEmpty()){ head = p; tail = p; }
  else { tail->next = p; tail = p; }
}
```

CHECK BIPARTITE GRAPHS – COMPLETE CODE

```
int pop(){
  if(queueEmpty()) return -1;
  int r = head->id;  Node* tmp = head;
  head = head->next;
  if(head == NULL) tail = NULL;
  free(tmp);
  return r;
}
Node* add(int id, Node* h){
  Node* p = makeNode(id);
  p->next = h;  return p;
}
```

```
int n,m;
Node* A[N];
int d[N];// d[v] is the level of d
void input(){
  scanf("%d %d",&n,&m);
  for(int v = 1; v <= n; v++) A[v] = NULL;
  for(int i = 1; i <= m; i++){
     int u,v;
     scanf("%d%d",&u,&v);
     A[u] = add(v,A[u]);  A[v] = add(u,A[v]);
  }
}
```

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

CHECK BIPARTITE GRAPHS – COMPLETE CODE

```
int BFS(int u){
  initQueue();  push(u);  d[u] = 0;
  while(!queueEmpty()){
    int v = pop();
    for(Node* p = A[v]; p != NULL; p = p->next){
       int x = p->id;
       if(d[x] > -1){ if(d[v] % 2 == d[x] % 2) return 0;  }
       else{  d[x] = d[v] + 1;   push(x);  }
    }
  }
  return 1;
}
```

```
void solve(){
  for(int v = 1; v <= n; v++)  d[v] = -1;
  int ans = 1;
  for(int v= 1; v <= n; v++) if(d[v]== -1){
    if(!BFS(v)){  ans = 0; break;  }
  }
  printf("%d",ans);
}
int main(){
  input();
  solve();
  return 0;
}
```

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !