THUẬT TOÁN ỨNG DỤNG

PHAM QUANG DUNG

Graphs

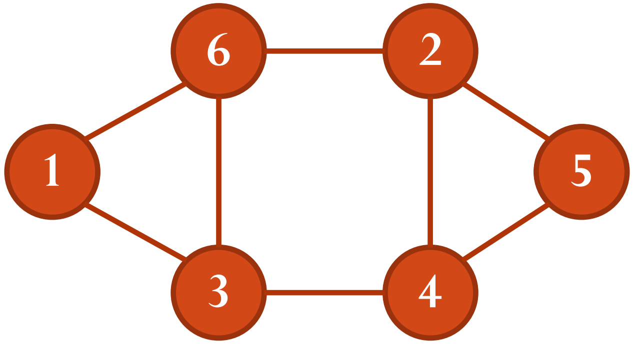
Phạm Quang Dũng
Bộ môn KHMT
dungpq@soict.hust.edu.vn

Nội dung

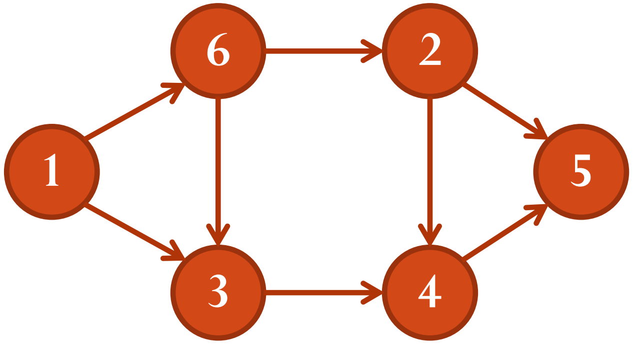
- Đồ thị và các thuật ngữ liên quan
- Tìm kiếm theo chiều sâu
- Tìm kiếm theo chiều rộng
- Chu trình Euler
- Thuật toán Dijkstra sử dụng hàng đợi ưu tiên
- Thuật toán Kruskal sử dụng disjoint-set structure
- Exercises

Đồ thị

- Đối tượng toán học bao gồm các đỉnh (node) và các liên kết giữa các đỉnh (cạnh, cung)
- Đồ thị *G* = (*V*,*E*), trong đó *V* là tập đỉnh, *E* là tập cạnh (cung)
  - $(u,v) \in E$, chúng ta nói u kề với v



Undirected graph
- *V* = {1, 2, 3, 4, 5, 6}
- *E* = {(1, 3), (1,6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 6), (4, 5)}



Directed graph
- *V* = {1, 2, 3, 4, 5, 6}
- *E* = {(1, 3), (1,6), (2, 4), (2, 5), (6, 2), (3, 4), (6, 3), (4, 5)}

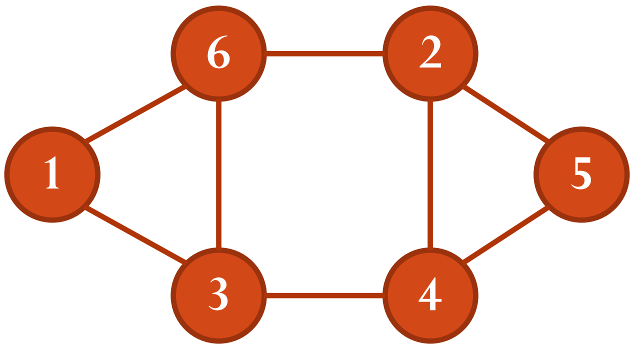
Đồ thị

- Bậc của một đỉnh là số đỉnh kề với nó

$$deg(v) = \#\{u \mid (u, v) \in E\}$$

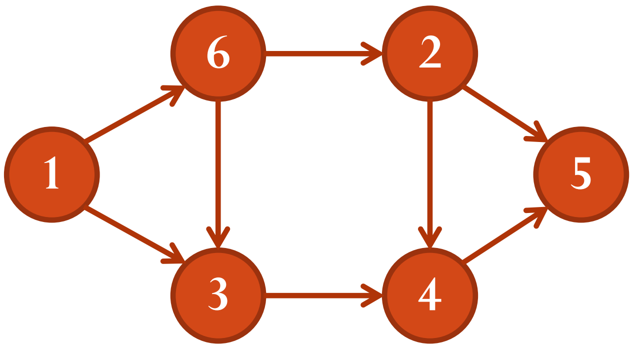
- Bán bậc vào (bán bậc ra) của một đỉnh là số cung đi vào (đi ra) khỏi đỉnh đó trên đồ thị có hướng:

$$deg^-(v) = \#\{u \mid (u, v) \in E\}, deg+(v) = \#\{u \mid (v, u) \in E\}$$



Undirected graph

- $deg(1) = 2$, $deg(6) = 3$

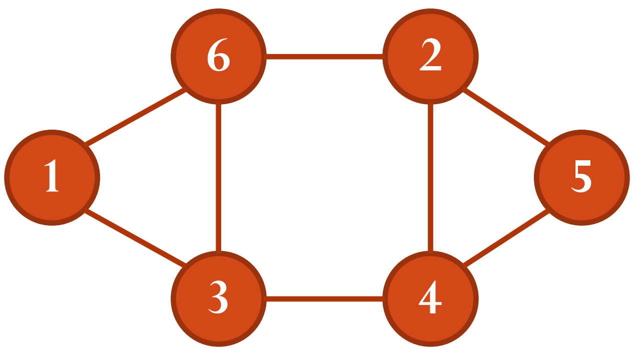


Directed graph

- $deg^-(1) = 0$, $deg^+(1) = 2$

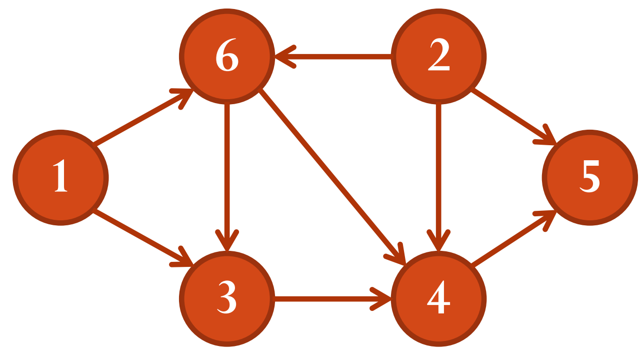
Đồ thị

- Cho đồ thị $G=(V, E)$ và 2 đỉnh $s, t \in V$, một đường đi từ s đến t trên G là chuỗi $s = x_0, x_1, \ldots, x_k = t$ trong đó $(x_i, x_{i+1}) \in E$, $\forall\, i = 0, 1, \ldots, k-1$



Path from 1 to 5:

- 1, 3, 4, 5
- 1, 6, 2, 5

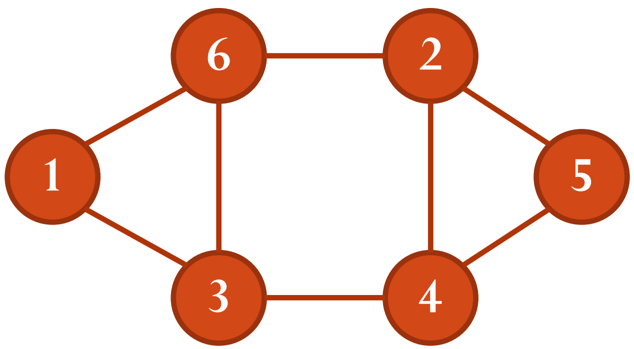


Path from 1 to 5:

- 1, 3, 4, 5
- 1, 6, 4, 5

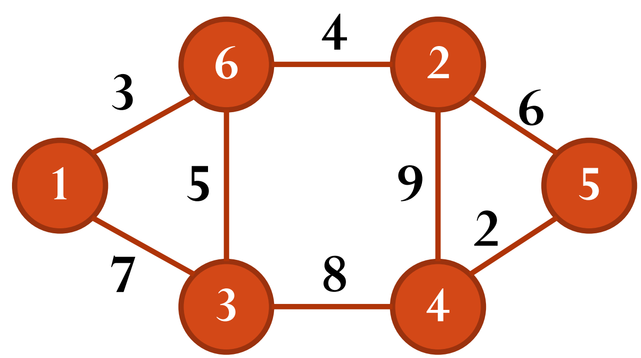
Đồ thị

- Ma trận kề



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

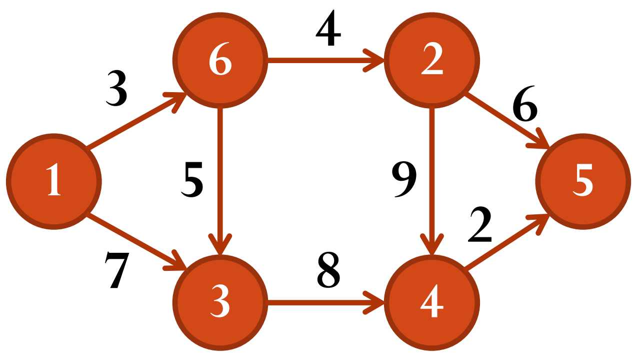
- Ma trận trọng số



	1	2	3	4	5	6
1	0	0	7	0	0	3
2	0	0	0	9	6	4
3	7	0	0	8	0	5
4	0	9	8	0	2	0
5	0	6	0	4	0	0
6	3	4	5	0	0	0

Biểu diễn đồ thị

- Danh sách kề
  - Với mỗi $v \in V$, $A(v)$ là tập các bộ (v, u, w) trong đó w là trọng số cung (v, u)
  - $A(1) = \{(1, 6, 3), (1, 3, 7)\}$
  - $A(2) = \{(2, 4, 9), (2, 5, 6)\}$
  - $A(3) = \{(3, 4, 8)\}$
  - $A(4) = \{(4, 5, 2)\}$
  - $A(5) = \{\}$
  - $A(6) = \{(6, 3, 5), (6, 2, 4)\}$



Duyệt đồ thị

- Thăm các đỉnh của đồ thị theo một thứ tự nào đó
- Mỗi đỉnh thăm đúng 1 lần
- Có 2 phương pháp chính
  - Duyệt theo chiều sâu: Depth-First Search (DFS)
  - Duyệt theo chiều rộng: Breadth-First Search (BFS)

Depth-First Search (DFS)

- DFS(*u*): DFS bắt đầu thăm từ đỉnh *u*
  - Nếu tồn tại một đỉnh *v* trong danh sách kề của *u* mà chưa được thăm → tiền hành thăm *v* và gọi DFS(*v*)
  - Nếu tất cả các đỉnh kề với *u* đã được thăm → DFS quay lui trở lại đỉnh *x* từ đó thuật toán thăm *u* và tiến hành thăm các đỉnh khác kề với *x* (gọi DFS(*x*)) mà chưa được thăm. Lúc này, đỉnh *u* được gọi là *đã duyệt xong*

Depth-First Search (DFS)

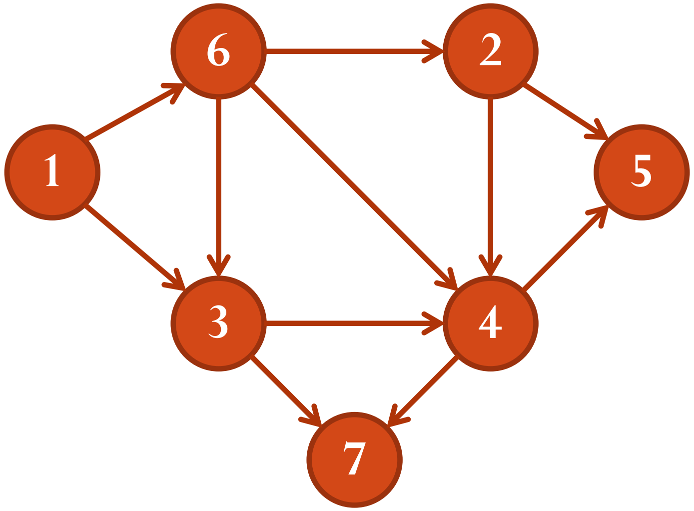
- Các thông tin liên quan đến mỗi đỉnh *v*
  - *p*(*v*): là đỉnh mà từ đó, DFS thăm đỉnh *v*
  - *d*(*v*): thời điểm đỉnh v *được thăm* nhưng *chưa duyệt xong*
  - *f*(*v*): thời điểm đỉnh v *đã duyệt xong*
  - color(*v*)
    - WHITE: *chưa thăm*
    - GRAY: *đã được thăm* nhưng *chưa duyệt xong*
    - BLACK: *đã duyệt xong*

Depth First Search - DFS

```
DFS(u) {
  t = t + 1;
  d(u) = t;
  color(u) = GRAY;
  foreach(adjacent node v to u)
  {
    if(color(v) = WHITE) {
      p(v) = u;
      DFS(v);
    }
  }
  t = t + 1;
  f(u) = t;
  color(u) = BLACK;
}
```

```
DFS() {
  foreach (node u of V) {
    color(u) = WHITE;
    p(u) = NIL;
  }
  foreach(node u of V) {
    if(color(u) = WHITE) {
      DFS(u);
    }
  }
}
```

Depth First Search - DFS

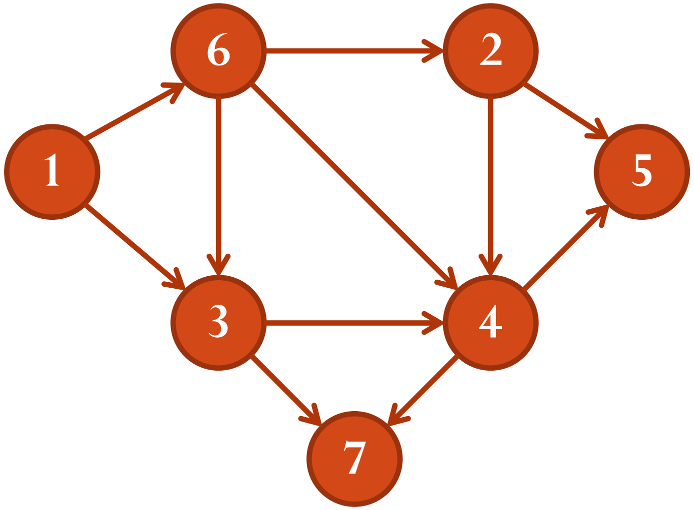


Node	1	2	3	4	5	6	7
d							
f							
p	-	-	-	-	-	-	-
color	W	W	W	W	W	W	W

Depth First Search - DFS

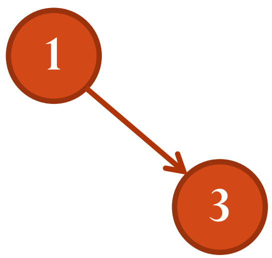
DFS(1)

Node	1	2	3	4	5	6	7
d	1						
f							
p	-	-	-	-	-	-	-
color	G	W	W	W	W	W	W

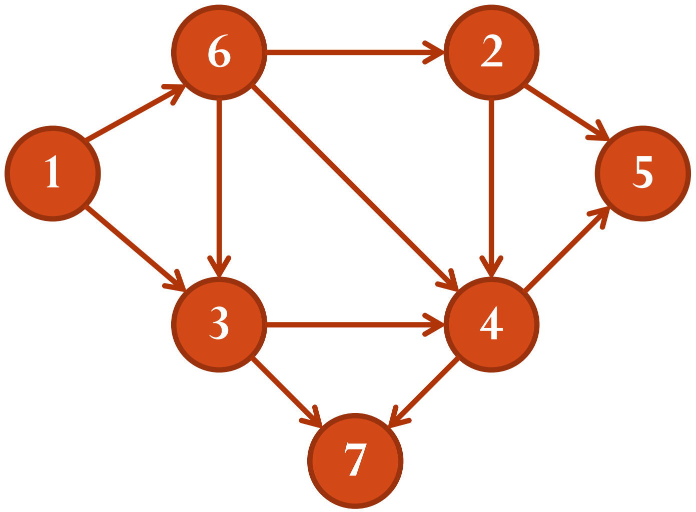


Depth First Search - DFS

DFS(1)

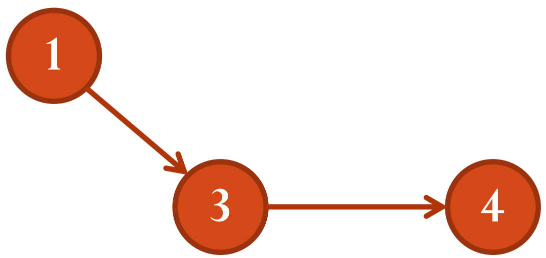


Node	1	2	3	4	5	6	7
d	1		2				
f							
p	-	-	1	-	-	-	-
color	G	W	G	W	W	W	W

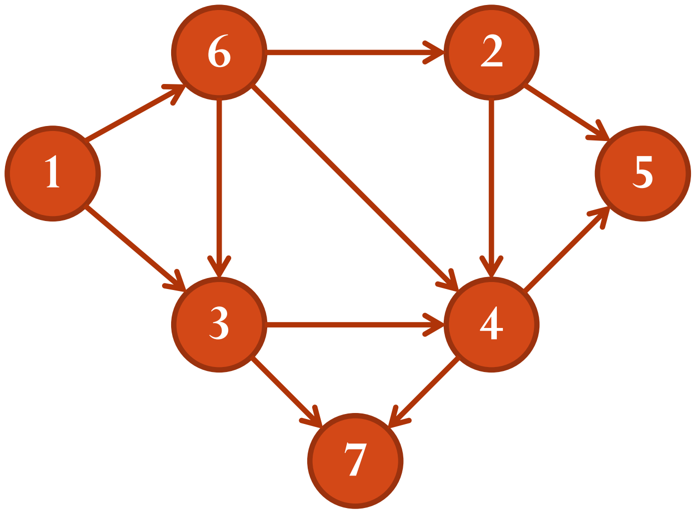


Depth First Search - DFS

DFS(1)

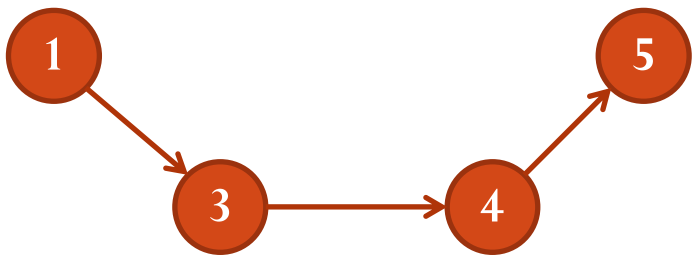


Node	1	2	3	4	5	6	7
d	1		2	3			
f							
p	-	-	1	3	-	-	-
color	G	W	G	G	W	W	W

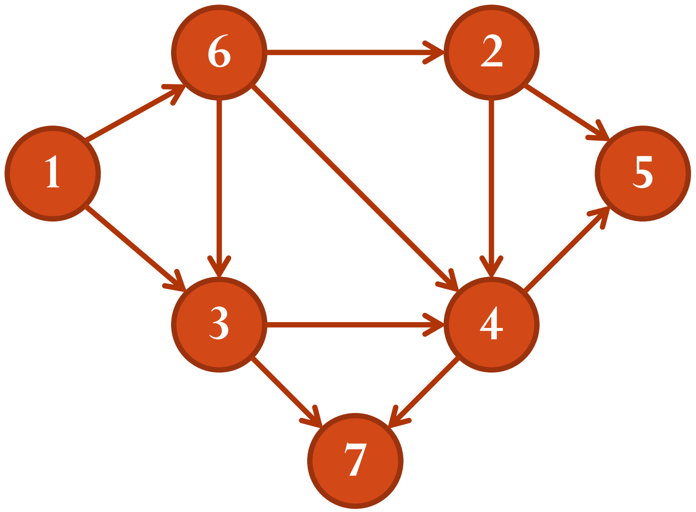


Depth First Search - DFS

DFS(1)

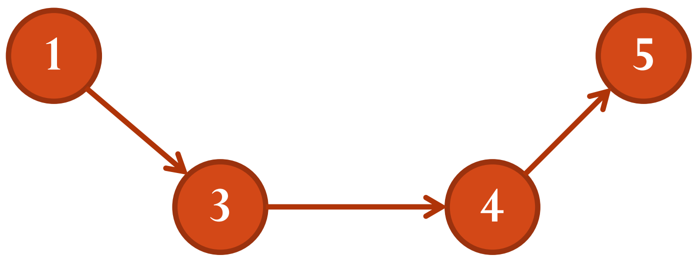


Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		
f							
p	-	-	1	3	4	-	-
color	G	W	G	G	G	W	W

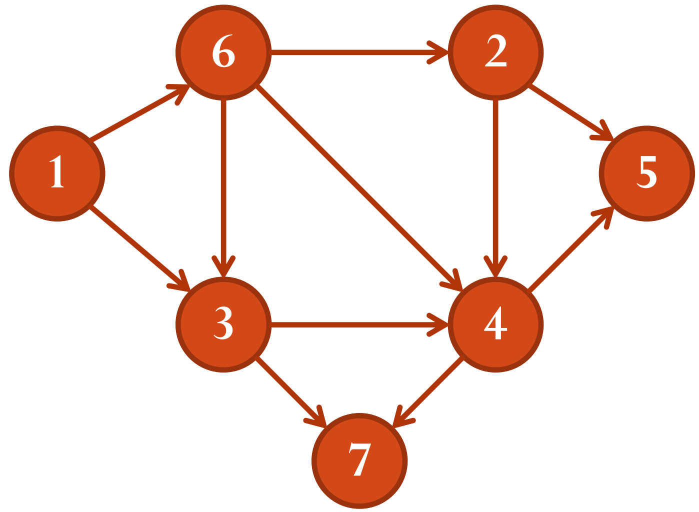


Depth First Search - DFS

DFS(1)

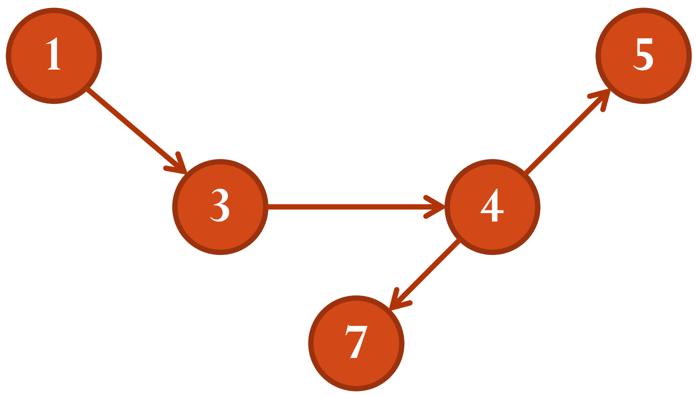


Node	1	2	3	4	5	6	7
d	1		2	3	4		
f					5		
p	-	-	1	3	4	-	-
color	G	W	G	G	B	W	W

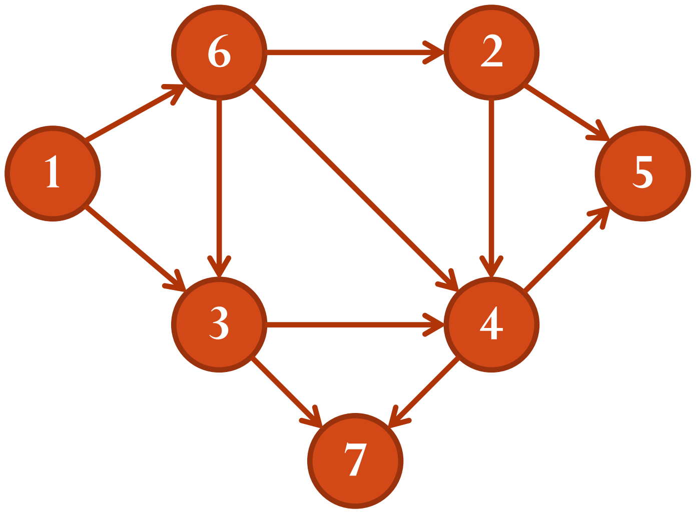


Depth First Search - DFS

DFS(1)

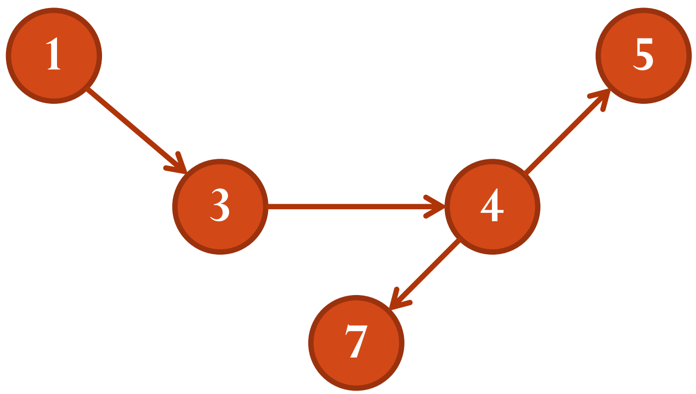


Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		6
f					5		
p	-	-	1	3	4	-	4
color	G	W	G	G	B	W	G

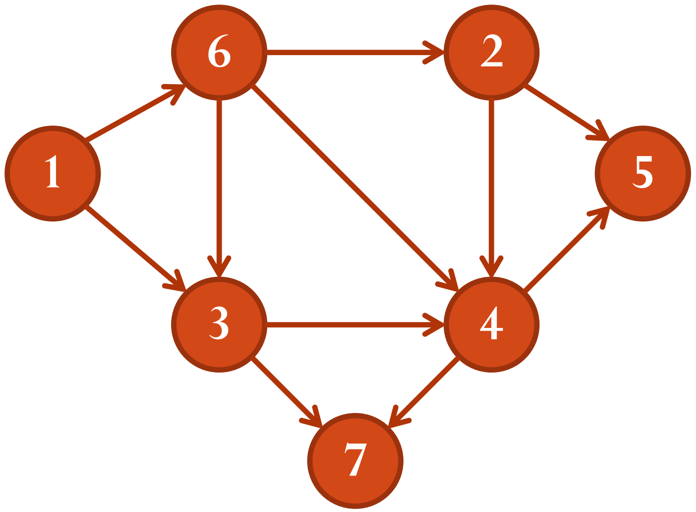


Depth First Search - DFS

DFS(1)

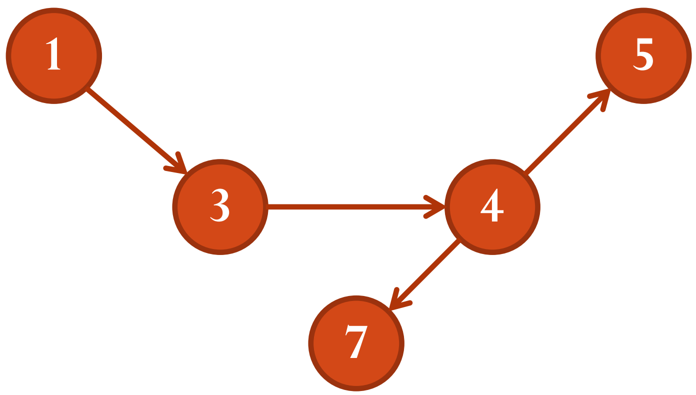


Node	1	2	3	4	5	6	7
d	1		2	3	4		6
f					5		7
p	-	-	1	3	4	-	4
color	G	W	G	G	B	W	B

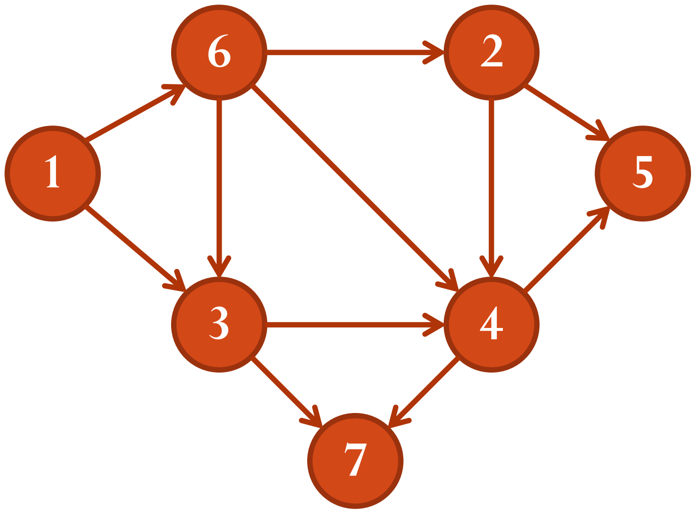


Depth First Search - DFS

DFS(1)

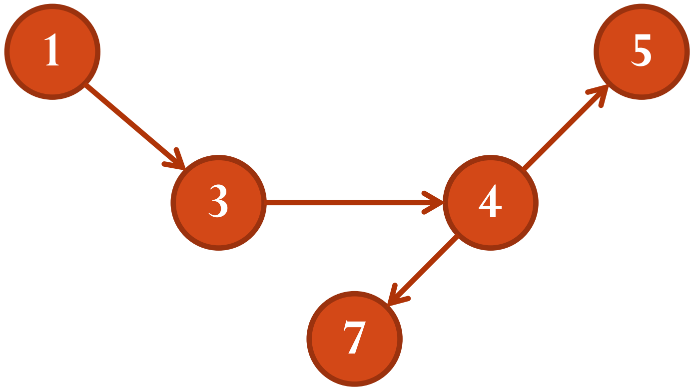


Node	1	2	3	4	5	6	7
d	1		2	3	4		6
f				8	5		7
p	-	-	1	3	4	-	4
color	G	W	G	B	B	W	B

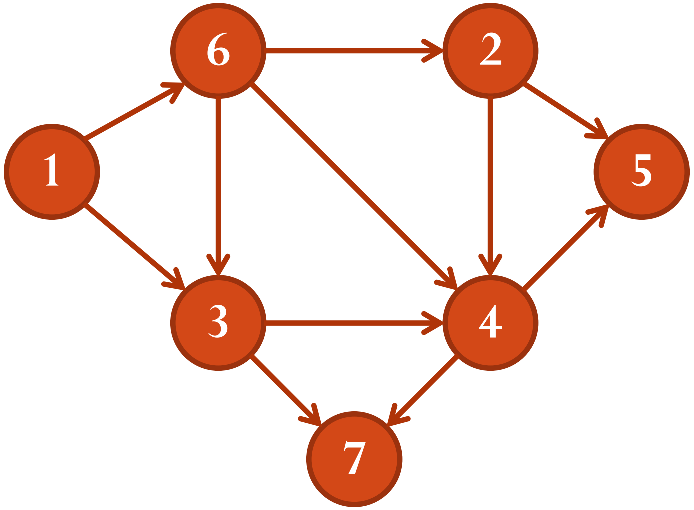


Depth First Search - DFS

DFS(1)

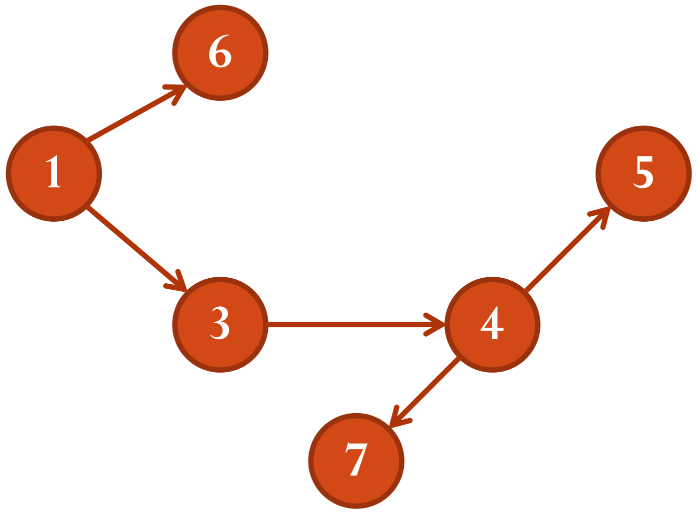


Node	1	2	3	4	5	6	7
d	1		2	3	4		6
f			9	8	5		7
p	-	-	1	3	4	-	4
color	G	W	B	B	B	W	B

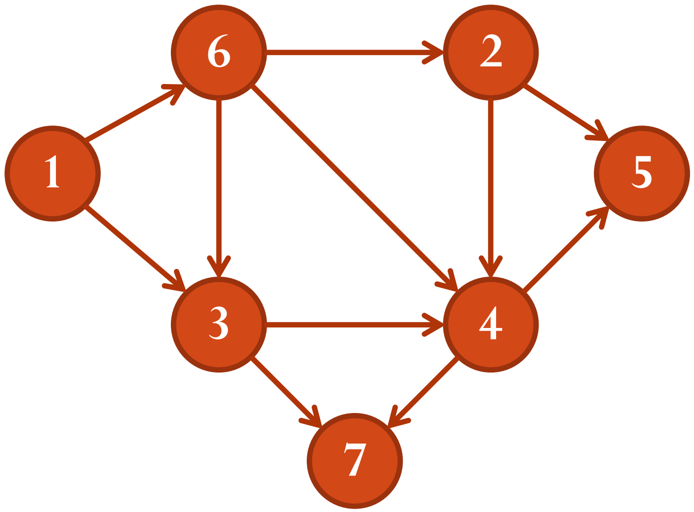


Depth First Search - DFS

DFS(1)

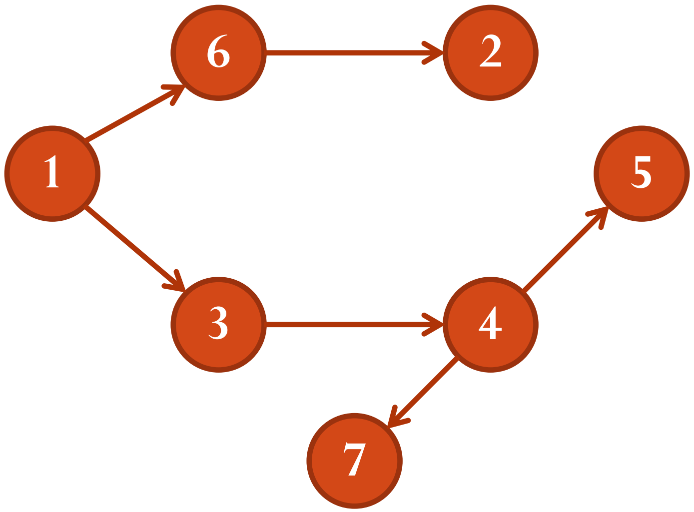


Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4	10	6
f			9	8	5		7
p	-	-	1	3	4	1	4
color	G	W	B	B	B	G	B

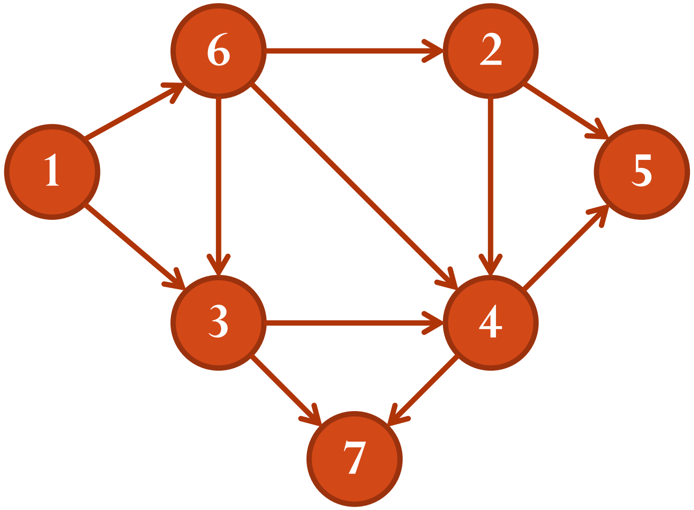


Depth First Search - DFS

DFS(1)

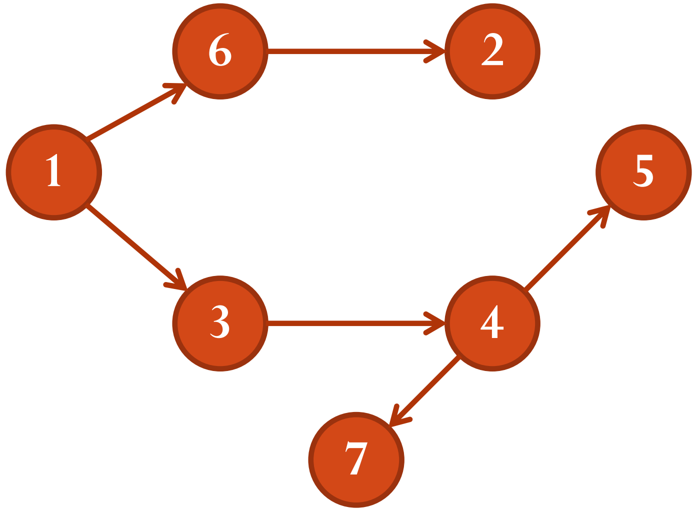


Node	1	2	3	4	5	6	7
d	1	11	2	3	4	10	6
f			9	8	5		7
p	-	6	1	3	4	1	4
color	G	G	B	B	B	G	B

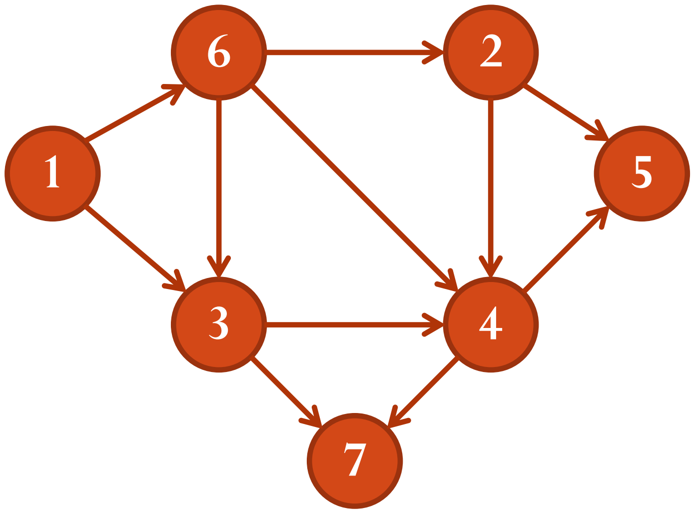


Depth First Search - DFS

DFS(1)

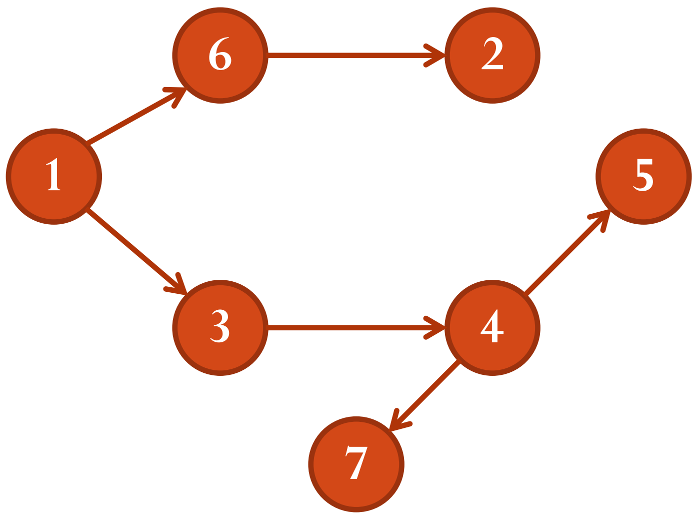


Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1	11	2	3	4	10	6
f		12	9	8	5		7
p	-	6	1	3	4	1	4
color	G	B	B	B	B	G	B

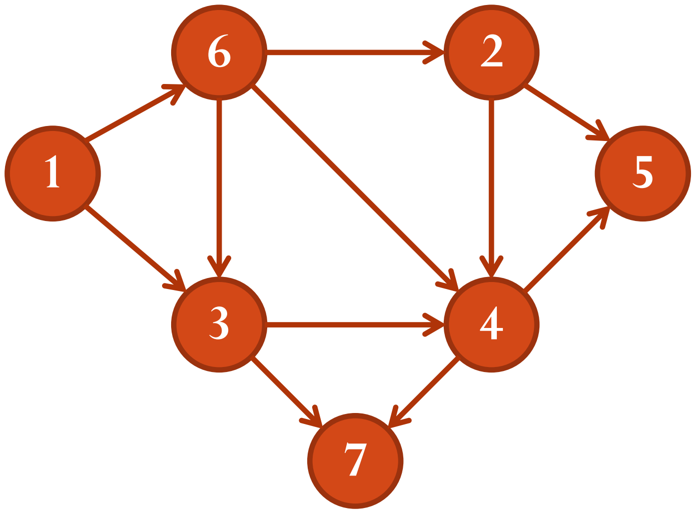


Depth First Search - DFS

DFS(1)

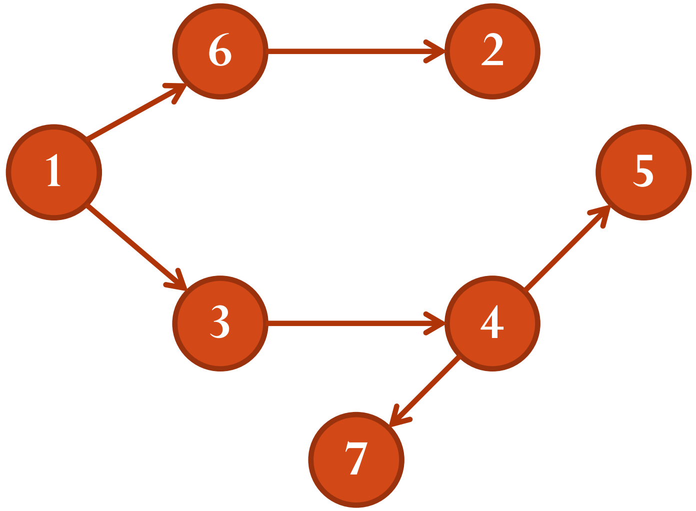


Node	1	2	3	4	5	6	7
d	1	11	2	3	4	10	6
f		12	9	8	5	13	7
p	-	6	1	3	4	1	4
color	G	B	B	B	B	B	B

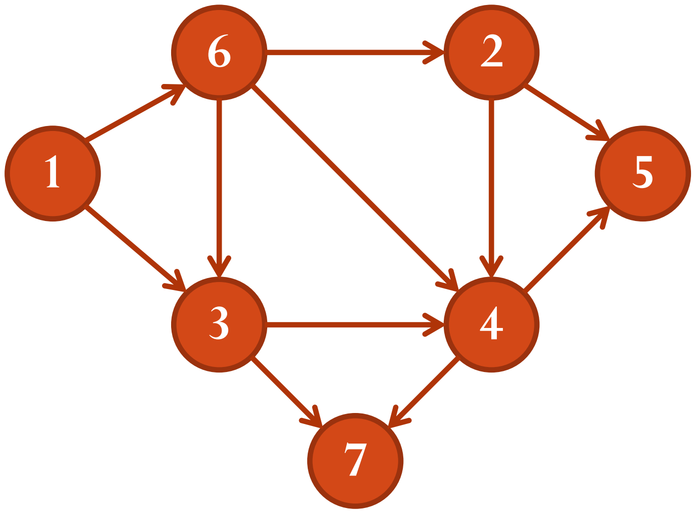


Depth First Search - DFS

DFS(1)



Node	1	2	3	4	5	6	7
d	1	11	2	3	4	10	6
f	14	12	9	8	5	13	7
p	-	6	1	3	4	1	4
color	B	B	B	B	B	B	B

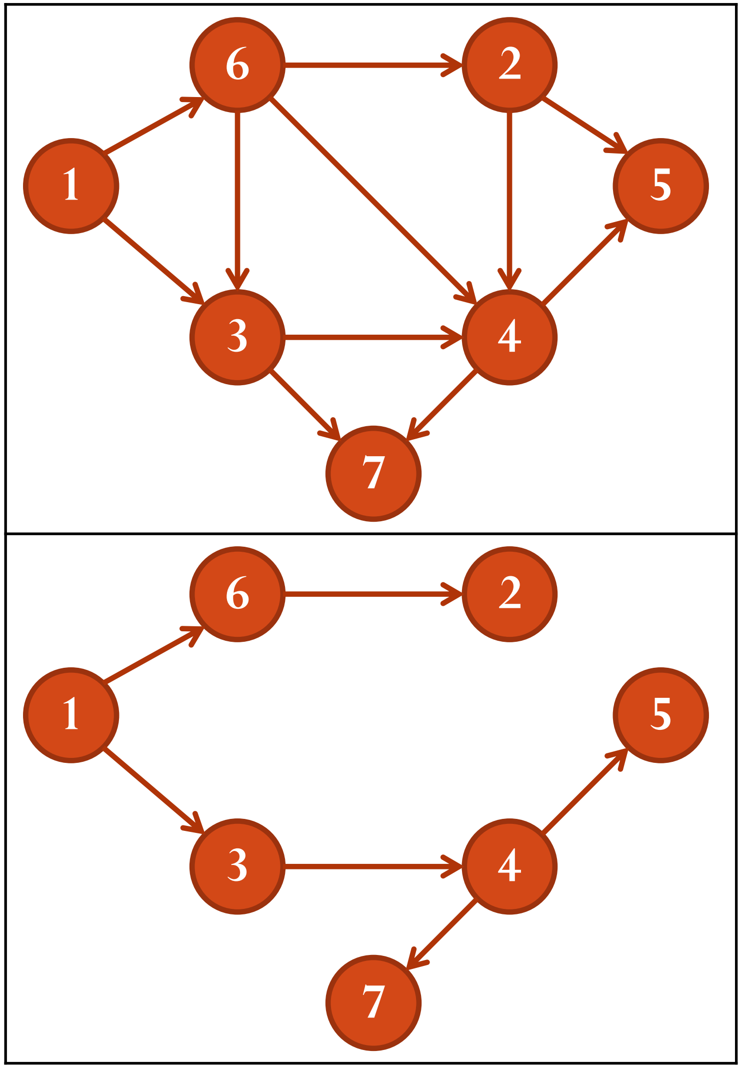


Depth First Search - DFS

- Kết quả của DFS là tập các cây DFS
- Phân loại cạnh
  - tree edge: (u,v) là cạnh cây nếu v được thăm từ u
  - back edge: (u,v) là cạnh ngược nếu v là tổ tiên của u
  - forward edge: (u,v) là cạnh xuôi nếu u là tổ tiên của v
  - crossing edge: cạnh ngang (các cạnh còn lại)

Depth First Search - DFS

- Phân loại cạnh
  - Cạnh cây: (1, 6), (1, 3), (6, 2), (3, 4), (4, 5), (4, 7)
  - Cạnh ngược:
  - Cạnh xuôi: (3, 7)
  - Cạnh ngang: (6, 3), (6, 4), (2, 4), (2,5)



Breadth First Search - BFS

- BFS(*u*): BFS xuất phát từ đỉnh *u*
  - Thăm *u*
  - Thăm các đỉnh kề với u và chưa được thăm (gọi là đỉnh mức 1)
  - Thăm các đỉnh kề với đỉnh mức 1 mà chưa được thăm (gọi là đỉnh mức 2)
  - Thăm các đỉnh mức 2 mà chưa được thăm (gọi là đỉnh mức 3)
  - . . .
- Cài đặt sử dụng cấu trúc hàng đợi

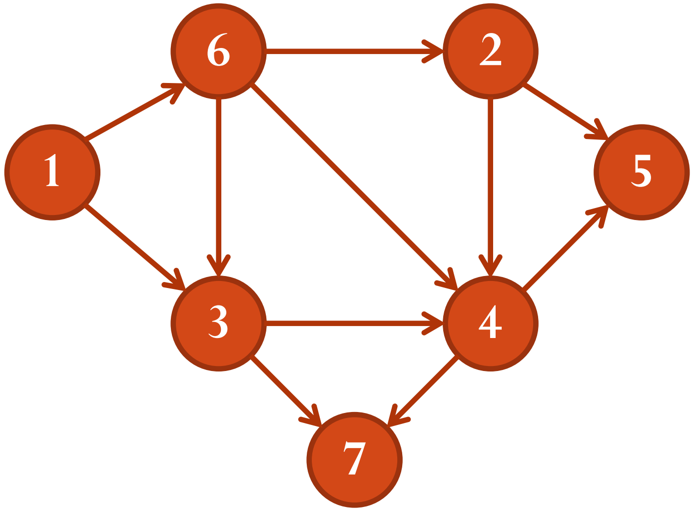
Breadth First Search - BFS

```
BFS(u) {
  d(u) = 0;
  Init a queue Q;
  enqueue(Q,u);
  color(u) = GRAY;
  while(Q not empty) {
   v = dequeue(Q);
   foreach(x adjacent node to
v) {
      if(color(x) = WHITE){
        d(x) = d(v) + 1;
        color(x) = GRAY;
        enqueue(Q,x);
      }
   }
  }
}
```

```
BFS() {
  foreach (node u of V) {
    color(u) = WHITE;
    p(u) = NIL;
  }
  foreach(node u of V) {
    if(color(u) = WHITE) {
       BFS(u);
    }
  }
}
```

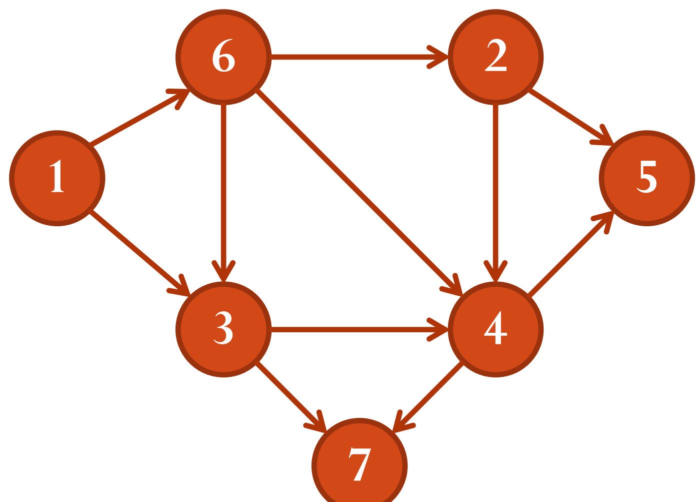
Breadth First Search - BFS

BFS(1)



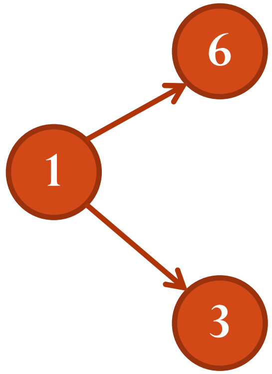
Breadth First Search - BFS

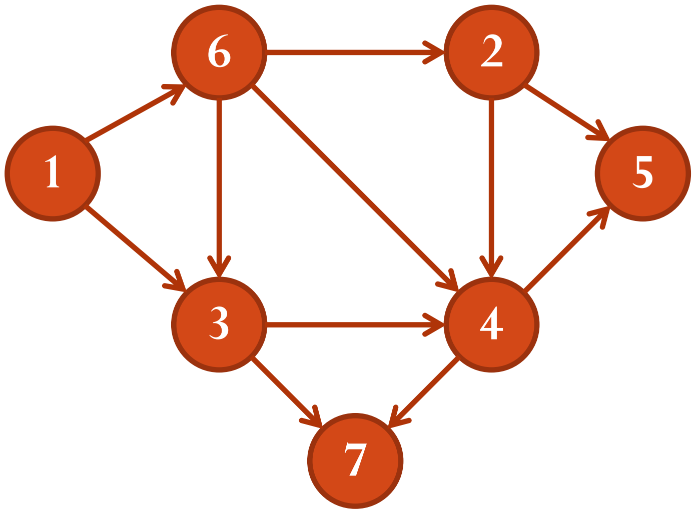
BFS(1)



Breadth First Search - BFS

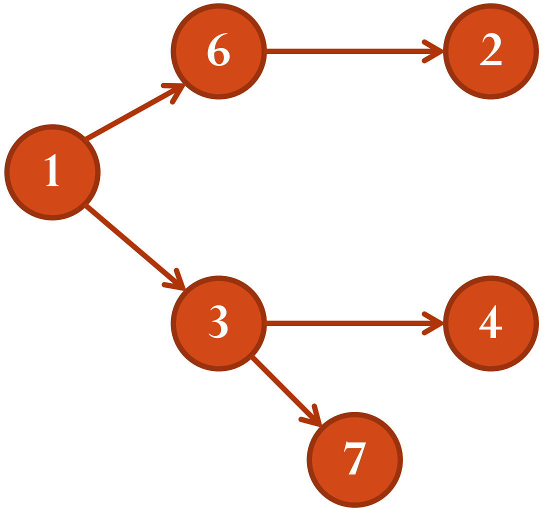
BFS(1)

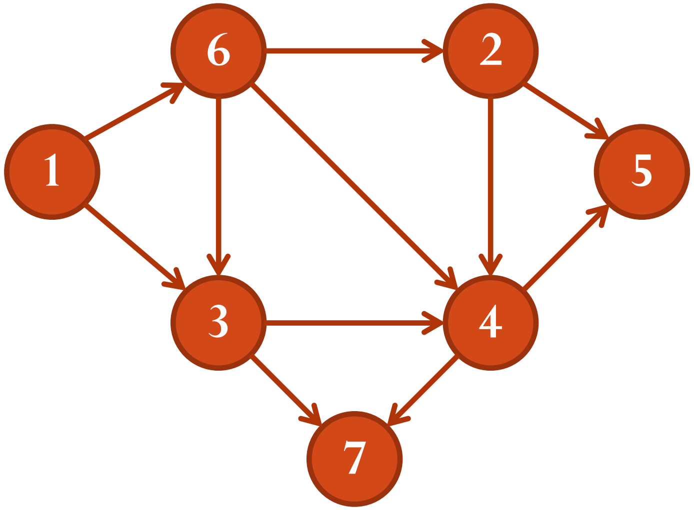




Breadth First Search - BFS

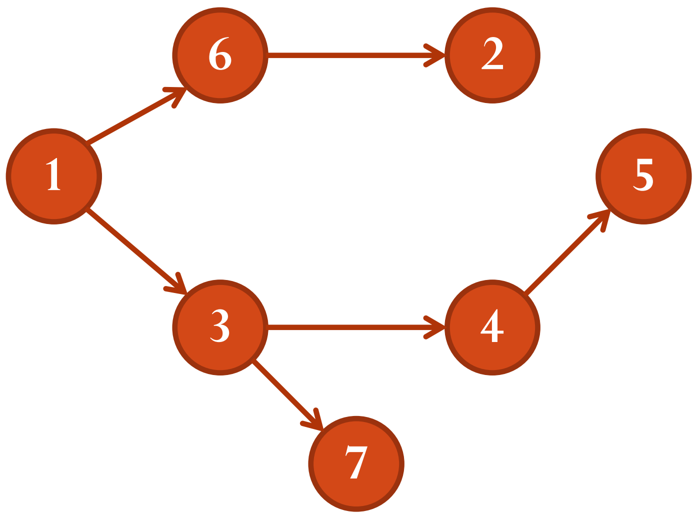
BFS(1)

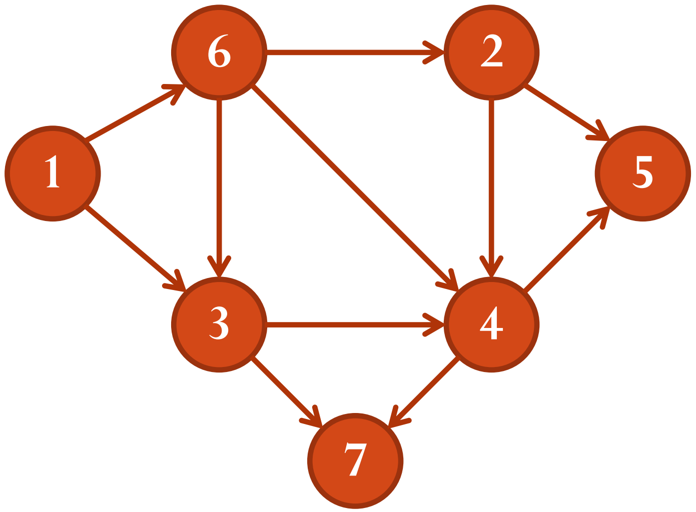




Breadth First Search - BFS

BFS(1)



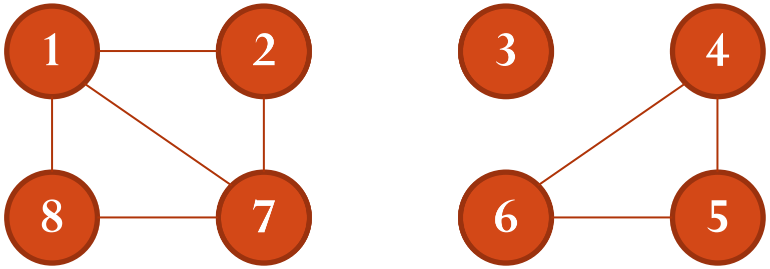


Ứng dụng DFS, BFS

- Tính toán thành phần liên thông của đồ thị vô hướng
- Tính thành phần liên thông mạnh của đồ thị có hướng
- Kiểm tra đồ thị hai phía
- Phát hiện chu trình
- Sắp xếp topo
- Tìm đường đi dài nhất trên cây

EXERCISE
CONNECTED COMPONENT

- Given a undirected graph G=(V, E) in which set of nodes V = {1,2,…,N}. Compute number of connected components.
- Example: following graph has 3 connected components
  - {1, 2, 7, 8}
  - {3}
  - {4, 5, 6}



EXERCISE
CONNECTED COMPONENT

- Input
  - Line 1: N and M ($1 \leq N \leq 10^6$, $1 \leq M \leq 10^7$)
  - Line i+1 (i = 1,…,M): u_i and v_i which are endpoints of the i^{th} edge
- Ouput
  - Number of connected components

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

EXERCISE
CONNECTED COMPONENT

```
#include <bits/stdc++.h>
#include <vector>
#include <iostream>
#define MAX_N 100001
using namespace std;
int N,M;
vector<int> A[MAX_N];
int visited[MAX_N];
int ans;
void input(){
    cin >> N >> M;
    for(int i = 1; i <= M; i++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(v);  A[v].push_back(u);
    }
}
```

EXERCISE
CONNECTED COMPONENT

```
void init(){
    for(int i = 1; i<=N; i++) visited[i] = 0;
}
void DFS(int u){
    for(int j = 0; j < A[u].size(); j++){
        int v = A[u][j];
        if(!visited[v]){
            visited[v] = 1;
            DFS(v);
        }
    }
}
```

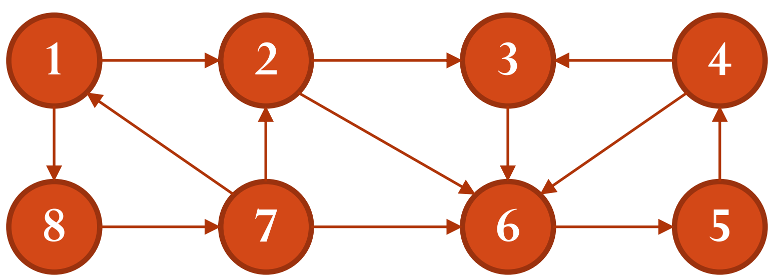
EXERCISE
CONNECTED COMPONENT

```
void solve(){
    init();
    ans = 0;
    for(int  v = 1; v <= N; v++)if(!visited[v]){
        ans++;
        DFS(v);
    }
    cout << ans;
}

int main(){
    input();
    solve();
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

- Given a directed graph G = (V,E) where V = {1,. . ., N} is the set of nodes. Compute the number of strongly connected components of G.
- Example, the graph below has 3 strongly connected components
  - {1,7,8}
  - {2}
  - {3,4,5,6}



EXERCISE
STRONGLY CONNECTED COMPONENT

- Input
  - Line 1: N and M ($1 \le N \le 10^6$, $1 \le M \le 10^7$)
  - Line i+1 ($i = 1,\ldots,M$): u_i and v_i which are endpoints of the i^{th} arc
- Ouput
  - Number of strongly connected components

stdin	stdout
8 13 1 2 1 8 2 3 2 6 3 6 4 3 4 6 5 4 6 5 7 1 7 2 7 6 8 7	3

EXERCISE
STRONGLY CONNECTED COMPONENT

```
#include <bits/stdc++.h>
#include <vector>
#include <iostream>
using namespace std;
#define MAX_N 100001

int n;
vector<int> A[MAX_N];
vector<int> A1[MAX_N];// residual graph

// data structure for DFS
int f[MAX_N];// finishing time
char color[MAX_N];
int t;
int icc[MAX_N];// icc[v] index of the strongly connected component containing v
int ncc;// number of connected components in the second DFS
int x[MAX_N];// sorted-list (inc finishing time) of nodes visited by DFS
int idx;
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
void buildResidualGraph(){// xay dung do thi bu
    for(int u = 1; u <= n; u++){
        for(int j = 0; j < A[u].size(); j++){
            int v = A[u][j];
            A1[v].push_back(u);
        }
    }
}
void init(){
    for(int v = 1; v <= n; v++){
        color[v] = 'W';
    }
    t = 0;
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
// DFS on the original graph
void dfsA(int s){
    t++;
    color[s] = 'G';
    for(int j = 0; j < A[s].size(); j++){
        int v = A[s][j];
        if(color[v] == 'W'){
            dfsA(v);
        }
    }
    t++;
    f[s] = t;
    color[s] = 'B';
    idx++;
    x[idx] = s;
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
void dfsA(){
    init();
    idx = 0;
    for(int v = 1; v <= n; v++){
        if(color[v] == 'W'){
            dfsA(v);
        }
    }
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
// DFS on the residual graph
void dfsA1(int s){
    t++;
    color[s] = 'G';
    icc[s] = ncc;
    for(int j = 0; j < A1[s].size(); j++){
        int v= A1[s][j];
        if(color[v] == 'W'){
            dfsA1(v);
        }
    }
    color[s] = 'B';
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
void dfsA1(){
    init();
    ncc = 0;
    for(int i = n; i >= 1; i--){
        int v = x[i];
        if(color[v] == 'W'){
            ncc++;
            dfsA1(v);
        }
    }
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

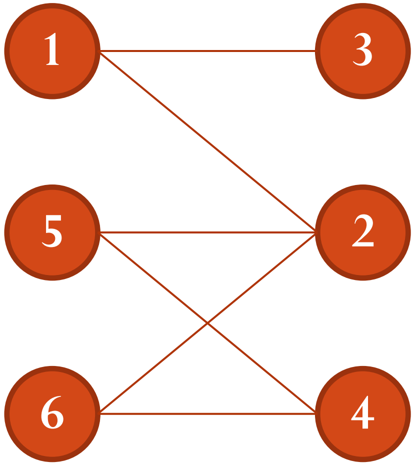
```
void solve(){
    dfsA();
    buildResidualGraph();
    dfsA1();
    cout << ncc;
}
void input(){
    int m;
    cin >> n >> m;
    for(int k = 1; k <= m; k++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(v);
    }
}
int main(){
    input(); solve();
}
```

EXERCISE
STRONGLY CONNECTED COMPONENT

```
void input(){
    int m;
    cin >> n >> m;
    for(int k = 1; k <= m; k++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(v);
    }
}
int main(){
    input();
    solve();
}
```

EXERCISE
BIPARTIE GRAPH

- Given undirected graph G = (V, E). Check whether or not G is a bipartie graph.



EXERCISE
BIPARTIE GRAPH

- Input
  - Line 1: N and M ($1 \leq N \leq 10^5$, $1 \leq M \leq 10^5$)
  - Line i+1 (i = 1,…,M): u_i and v_i which are endpoints of the i^{th} edge
- Ouput
  - Write 1 if G is bipartie graph, and write 0, otherwise

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

EXERCISE
BIPARTIE GRAPH

```
#include <bits/stdc++.h>
#include <vector>
#include <queue>
#include <iostream>
using namespace std;
#define MAX_N 100001
int N, M;
vector<int> A[MAX_N];
int d[MAX_N];// d[v] is the level of d
void input(){
    cin >> N >> M;
    for(int i = 1; i <= M; i++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(v);
        A[v].push_back(u);
    }
}
```

EXERCISE
BIPARTIE GRAPH

```
int BFS(int u){
    queue<int> Q;
    Q.push(u);
    d[u] = 0;
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        for(int i = 0; i < A[v].size(); i++){
            int x = A[v][i];
            if(d[x] > -1){
                if(d[v] % 2 == d[x] % 2) return 0;
            }else{
                d[x] = d[v] + 1;
                Q.push(x);
            }
        }
    }
}
```

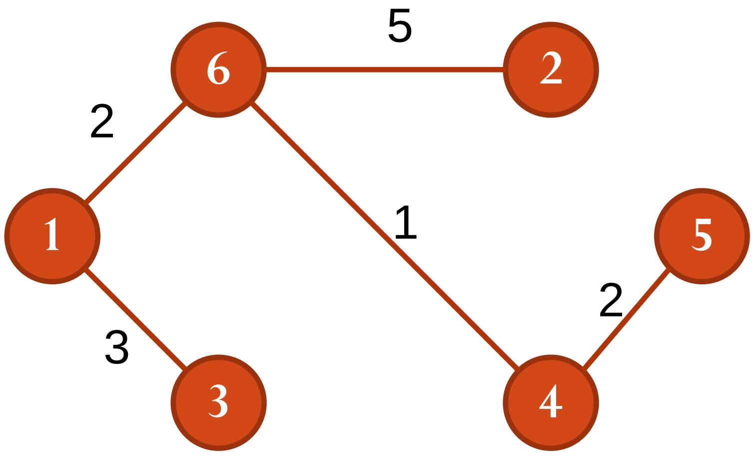
EXERCISE
BIPARTIE GRAPH

```
void solve(){
    init();
    int ans = 1;
    for(int v= 1; v <= N; v++) if(d[v]== -1){
        if(!BFS(v)){
            ans = 0; break;
        }
    }
    cout << ans ;
}

int main(){
    input();
    solve();
}
```

EXERCISE
LONGEST PATH ON A TREE

- Given a undirected tree G = (V, E) in which V = {1,…,N} is the set of nodes. Each edge (u,v) $\in$ E has weight w(u,v). The length of a path is defined to be the sum of weights of edges of this path. Find the longest elementary path on G.



EXERCISE
LONGEST PATH ON A TREE

- Input
  - Line 1: N ($1 \leq N \leq 10^5$)
  - Line i + 1 (i = 1,…,N-1): u, v, w in which w is the weight of edge (u,v) ($1 \leq w \leq 100$)
- Output
  - The weight of the longest path on the given tree

stdin	stdout
6 1 3 3 1 6 2 2 6 5 4 5 2 4 6 1	10

EXERCISE
LONGEST PATH ON A TREE

```
#include <bits/stdc++.h>
#include <vector>
#include <iostream>
using namespace std;
#define MAX_N 100001

int N;
vector<int> A[MAX_N];// A[v][i] is the i^th adjacent node to v
vector<int> c[MAX_N];// c[v][i] is the weight of the i^th adjacent edge to v
int d[MAX_N];// d[v] is the distance from the source node to v in BFS
int p[MAX_N];// p[v] is the parent of v in BFS
```

EXERCISE
LONGEST PATH ON A TREE

```
void input(){
    std::ios::sync_with_stdio(false);
    cin >> N;
    for(int i = 1; i <= N-1; i++){
        int u,v,w;
        cin >> u >> v >> w;
        A[v].push_back(u);
        A[u].push_back(v);
        c[v].push_back(w);
        c[u].push_back(w);
    }
}
```

EXERCISE
LONGEST PATH ON A TREE

```
void BFS(int u){
    queue<int> Q;
    d[u] = 0;
    Q.push(u);
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        for(int i = 0; i < A[v].size(); i++){
            int x = A[v][i];
            if(d[x] > -1){ if(p[v] != x) cout << "FALSE" << endl;continue;}
            int w = c[v][i];
            Q.push(x);
            d[x] = d[v] + w;
            p[x] = v;
        }
    }
}
```

EXERCISE
LONGEST PATH ON A TREE

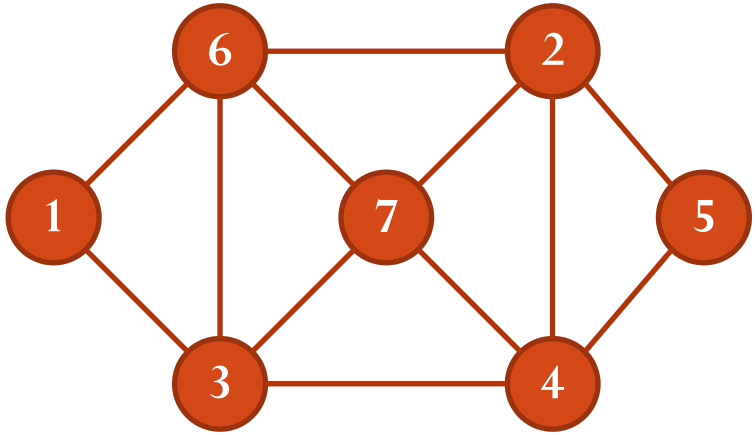
```
int findMax(){
    int max_d = -1;
    int u = -1;
    for(int v = 1; v <= N; v++){
        if(max_d < d[v]){
            max_d = d[v];
            u = v;
        }
    }
    return u;
}
void init(){
    for(int v = 1; v <= N; v++){ d[v] = -1; p[v] = -1;}
}
```

EXERCISE
LONGEST PATH ON A TREE

```
void solve(){
    init();
    BFS(1);
    int u = findMax();
    init();
    BFS(u);
    u = findMax();
    cout << d[u];
}
int main(){
    input();
    solve();
}
```

Eulerian and Hamiltonian cycles

- Given undirected graph G = (V, E)
  - Eulerian cycle of G is a cycle that passes each edge of G exactly once
  - Hamiltonian cycle of G is a cycle that visits each node of G exactly once
- A graph containing an Eulerian cycle is called Eulerian graph
- A graph containing a Hamiltonian cycle is called Hamiltonian graph



- Eulerian cycle: 1, 6, 3, 7, 6, 2, 5, 4, 2, 7, 4, 3, 1
- Hamiltonian cycle: 1, 6, 2, 5, 4, 7, 3, 1

Algorithm for finding an Eulerian cycle

```
euler(G = (V, A)) {
  Init stacks S, CE;
  select v of V;
  push(S,v);
  while(S is not empty) {
    x  = top(S);
    if(A(x) is not empty) {
      select y ∊ A(x);
      push(S,y);
      remove (x,y) from G;
    }else{
      x = pop(S); push(CE,x);
    }
  }
  sequence of nodes in CE forms an euler;
}
```

Algorithm for finding a Hamiltonian cycle

- Use backtracking
- Input $G = (V, E)$ in which
  - $V = \{1, 2, \ldots, n\}$
  - $A(v)$ set of adjacent nodes to v
- Solution representation: $x[1..n]$, the cycle will be x[1] → x[2] → . . . → x[n] → x[1]

```
TRY(k) {// try values for x[k] being aware of
        // x[1],…,x[k-1]
  for(v ∈ A(x[k-1]) {
    if(not mark[v]) {
      x[k] = v;
      mark[v] = true;
      if(k == n) {
        if(v ∈ A(x[1]) {
          retrieve a Hamiltonian cycle x;
        }
      }else{
        TRY(k+1);
      }
      mark[v] = false;
    }
  }
}
```

Minimum Spanning Tree

- Given a undirected graph $G = (V, E, w)$.
  - Each edge $(u, v) \in E$ has weight $w(u, v)$
  - If $(u, v) \notin E$ then $w(u, v) = \infty$
- A Spanning tree of G is a undirected connected graph with no cycle, and contains all node of G.
  - $T = (V, F)$ in which $F \subseteq E$
  - Weight of T: $w(T) = \sum_{e \in F} w(e)$
- Find a spanning tree such that the weight is minimal

Minimum Spanning Tree - KRUSKAL

- Main idea (greedy)
  - Each step, select the minimum-cost edge and insert it into the spanning tree (under construction) if no cycle is created.

```
KRUSKAL(G = (V,E)){
    ET = {}; C = E;
    while(|ET| < |V|-1 and |C| > 0){
        e = select minimum-cost edge of C;
        C = C \ {e};
        if(ET ∪ {e} create no cycle){
            ET = ET ∪ {e};
        }
    }
    if(|ET| = |V|-1) return ET;
    else return null;
}
```

Minimum Spanning Tree - KRUSKAL

```
#include <iostream>
#define MAX 100001

using namespace std;

// data structure for input graph
int N, M;
int u[MAX];
int v[MAX];
int c[MAX];
int ET[MAX];
int nET;
// data structure for disjoint-set
int r[MAX];// r[v] is the rank of the set v
int p[MAX];// p[v] is the parent of v
long long rs;
```

Minimum Spanning Tree - KRUSKAL

```
void link(int x, int y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
void makeSet(int x){
    p[x] = x;
    r[x] = 0;
}
int findSet(int x){
    if(x != p[x])
        p[x] = findSet(p[x]);
    return p[x];
}
```

Minimum Spanning Tree - KRUSKAL

```
void swap(int& a, int& b){
    int tmp = a; a = b; b = tmp;
}
void swapEdge(int i, int j){
    swap(c[i],c[j]);    swap(u[i],u[j]);    swap(v[i],v[j]);
}
int partition(int L, int R, int index){
    int pivot = c[index];
    swapEdge(index,R);
    int storeIndex = L;
    for(int i = L; i <= R-1; i++){
        if(c[i] < pivot){
            swapEdge(storeIndex,i);
            storeIndex++;
        }
    }
    swapEdge(storeIndex,R);
    return storeIndex;
}
```

Minimum Spanning Tree - KRUSKAL

```
void quickSort(int L, int R){
    if(L < R){
        int index = (L+R)/2;
        index = partition(L,R,index);
        if(L < index) quickSort(L,index-1);
        if(index < R) quickSort(index+1,R);
    }
}
void quickSort(){
    quickSort(0,M-1);
}
```

Minimum Spanning Tree - KRUSKAL

```
void solve(){
    for(int x = 1; x <= N; x++)  makeSet(x);
    quickSort();
    rs = 0;
    int count = 0;
    nET = 0;
    for(int i = 0;i < M; i++){
        int ru = findSet(u[i]);
        int rv = findSet(v[i]);
        if(ru != rv){
            link(ru,rv);
            nET++;  ET[nET] = i;
            rs += c[i];
            count++;
            if(count == N-1) break;
        }
    }
    cout << rs;
}
```

Minimum Spanning Tree - KRUSKAL

```
void input(){
    cin >> N >> M;
    for(int i = 0; i < M; i++){
        cin >> u[i] >> v[i] >> c[i];
    }
}
int main(){
    input();
    solve();
 }
```

Minimum Spanning Tree - PRIM

- Main idea (greedy)
  - Each step, select a node having minimum distance to the tree under construction for insertion
- Data structures
  - Each $v \notin V_T$
    - $d(v)$ is the distance from v to V_T:

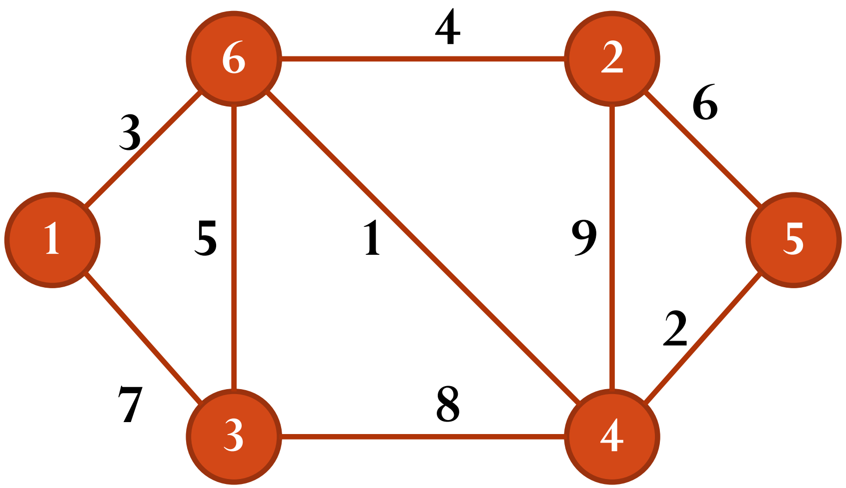
$$d(v) = \min\{w(v, u) \mid u \in V_T, (u, v) \in E\}$$

    - *near(v)*: the node $\in V_T$ having $w(v, \text{near}(v)) = d(v)$;

Minimum Spanning Tree - PRIM

```
PRIM(G = (V,E,w)) {
  select a random node s of V;
  for(v ∈ V) {
     d(v) = w(s,v); near(v) = s;
  }
  E_T = {}; V_T = {s};
  while(|V_T| ≠ |V|) {
    v = select a node ∈ V \ V_T having minimum d(v);
    V_T = V_T∪{v}; E_T = E_T ∪ {(v, near(v))};
    for(x ∈ V \ V_T) {
      if(d(x) > w(x,v) {
        d(x) = w(x,v);
        near(x) = v;
      }
    }
  }
  return (V_T, E_T);
}
```

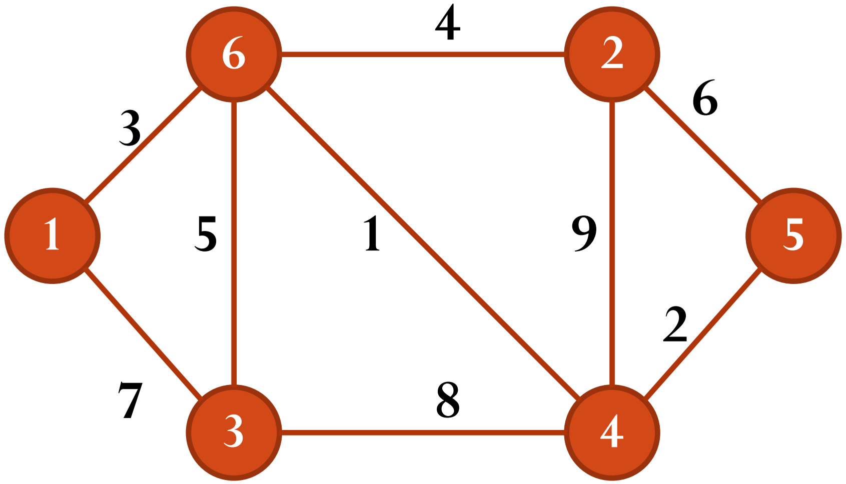
Minimum Spanning Tree - PRIM



- Each cell associated with a node v has label $(d(v), \text{near}(v))$
- Starting node $s = 1$

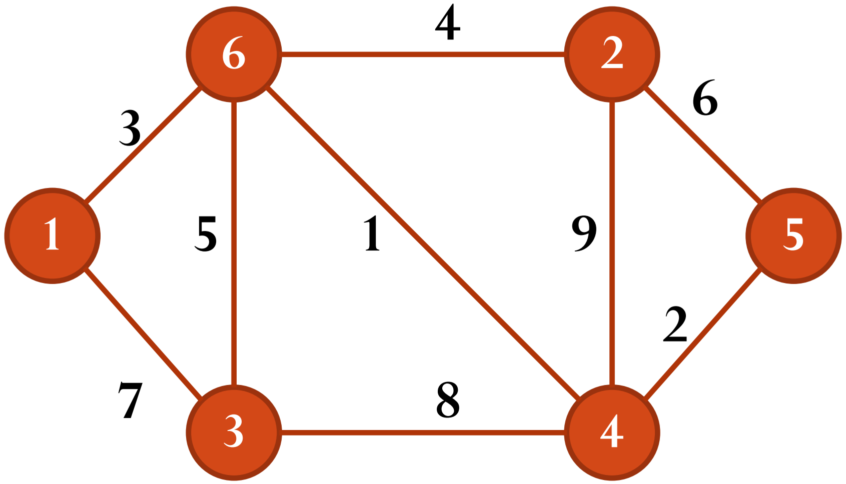
	1	2	3	4	5	6	E_T
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1)**	
Step 1	**-**						
Step 2	**-**						
Step 3	**-**						
Step 4	**-**						
Step 5	**-**						

Minimum Spanning Tree - PRIM



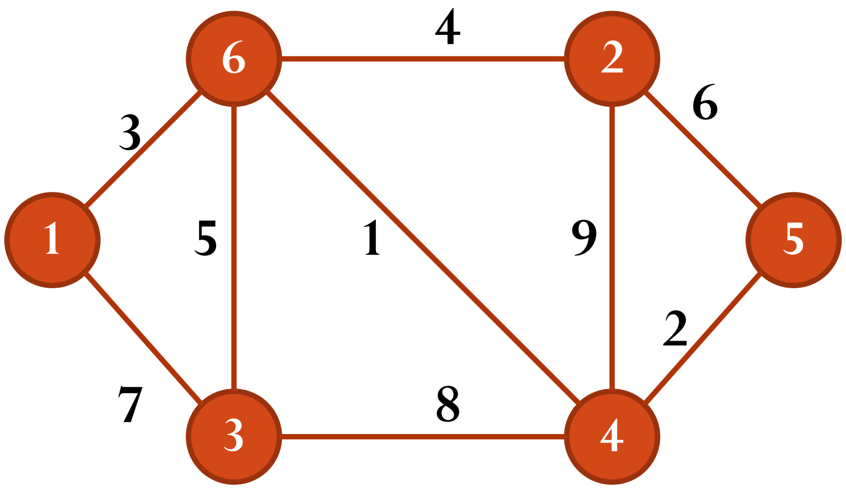
	1	2	3	4	5	6	E_T
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *	(1,6)
Step 1	-	(4,6)	(5,6)	(1,6)	(∞,1)	-	
Step 2	-					-	
Step 3	-					-	
Step 4	-					-	
Step 5	-					-	

Minimum Spanning Tree - PRIM



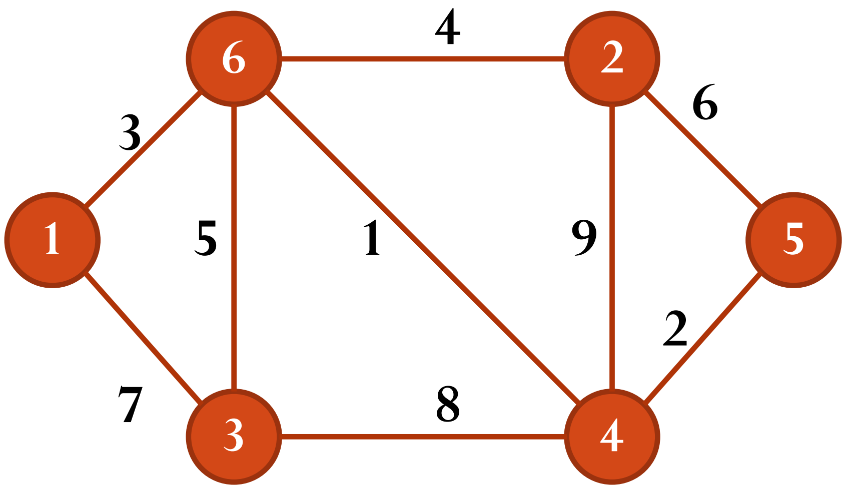
	1	2	3	4	5	6	E_T
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *	(1,6)
Step 1	-	(4,6)	(5,6)	(1,6) *	(∞,1)	-	(1,6), (4,6)
Step 2	-	(4,6)	(5,6)	-	(2,4)	-	
Step 3	-			-		-	
Step 4	-			-		-	
Step 5	-			-		-	

Minimum Spanning Tree - PRIM



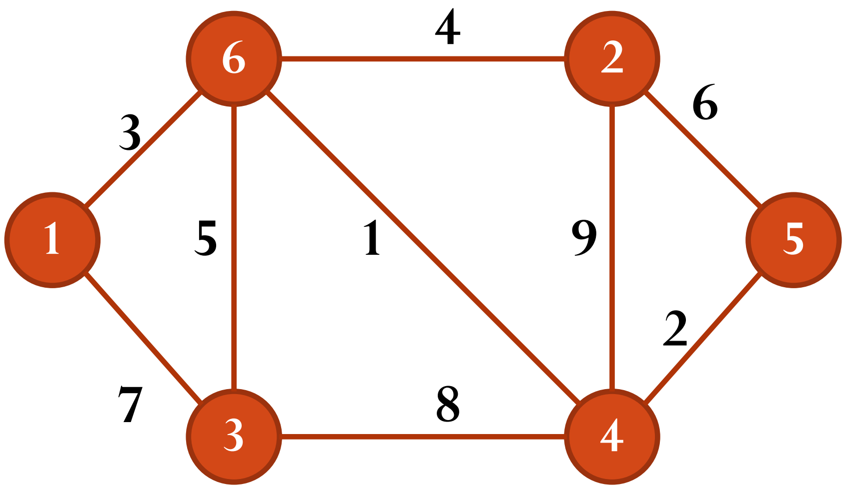
	1	2	3	4	5	6	E_T
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***	**(1,6)**
Step 1	**-**	**(4,6)**	**(5,6)**	**(1,6) ***	**(∞,1)**	**-**	**(1,6),(4,6)**
Step 2	**-**	**(4,6)**	**(5,6)**	**-**	**(2,4) ***	**-**	**(1,6),(4,6),(4,5)**
Step 3	**-**	**(4,6)**	**(5,6)**	**-**	**-**	**-**	
Step 4	**-**			**-**	**-**	**-**	
Step 5	**-**			**-**	**-**	**-**	

Minimum Spanning Tree - PRIM



	1	2	3	4	5	6	E_T
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***	**(1,6)**
Step 1	**-**	**(4,6)**	**(5,6)**	**(1,6) ***	**(∞,1)**	**-**	**(1,6),(4,6)**
Step 2	**-**	**(4,6)**	**(5,6)**	**-**	**(2,4) ***	**-**	**(1,6),(4,6),(4,5)**
Step 3	**-**	**(4,6) ***	**(5,6)**	**-**	**-**	**-**	**(1,6),(4,6),(4,5),(2,6)**
Step 4	**-**	**-**	**(5,6)**	**-**	**-**	**-**	
Step 5	**-**	**-**		**-**	**-**	**-**	

Minimum Spanning Tree - PRIM



	1	2	3	4	5	6	E_T
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1)	
Step 1	-	(4,6)	(5,6)	(1,6) *	(∞,1)	-	(1,6)
Step 2	-	(4,6)	(5,6)	-	(2,4) *	-	(1,6), (4,6)
Step 3	-	(4,6) *	(5,6)	-	-	-	(1,6), (4,6), (4,5)
Step 4	-	-	(5,6) *	-	-	-	(1,6), (4,6), (4,5), (2,6)
Step 5	-	-	-	-	-	-	(1,6), (4,6), (4,5), (2,6), (3,6)

Shortest Path Problem - Dijkstra

- Given a weighted graph $G = (V, E, w)$.
  - Each edge $(u, v) \in E$ has weight $w(u, v)$ which is non-negative
  - If $(u, v) \notin E$ then $w(u, v) = \infty$
- Given a node s of V, find the shortest path from s to other nodes of G

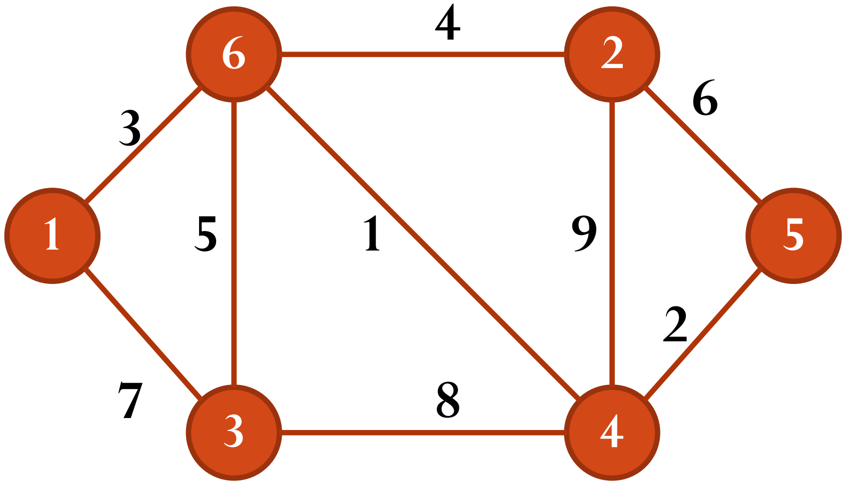
Shortest Path Problem - Dijkstra

- Main idea Dijkstra:
  - Each $v \in V$:
    - $\mathcal{P}(v)$ upper bound of the shortest path from s to v
    - $d(v)$: weight of $\mathcal{P}(v)$
    - $p(v)$: predecessor of v on $\mathcal{P}(v)$
  - Initialization
    - $\mathcal{P}(v) = \langle s, v \rangle$, $d(v) = w(s,v)$, $p(v) = s$
  - Upper bound improvement
    - If there exists a node u such that $d(v) > d(u) + w(u, v)$ then update:
      - $d(v) = d(u) + w(u, v)$
      - $p(v) = u$

Shortest Path Problem - Dijkstra

```
Dijkstra(G = (V,E,w)) {
  for(v ∈ V) {
     d(v) = w(s,v); p(v) = s;
  }
  S = V \ {s};
  while(S ≠ {}) {
    u = select a node ∈ S having minimum d(u);
    S = S \ {u};
    for(v ∈ S) {
      if(d(v) > d(u) + w(u,v) {
        d(v) = d(u) + w(u,v);
        p(v) = u;
      }
    }
  }
}
```

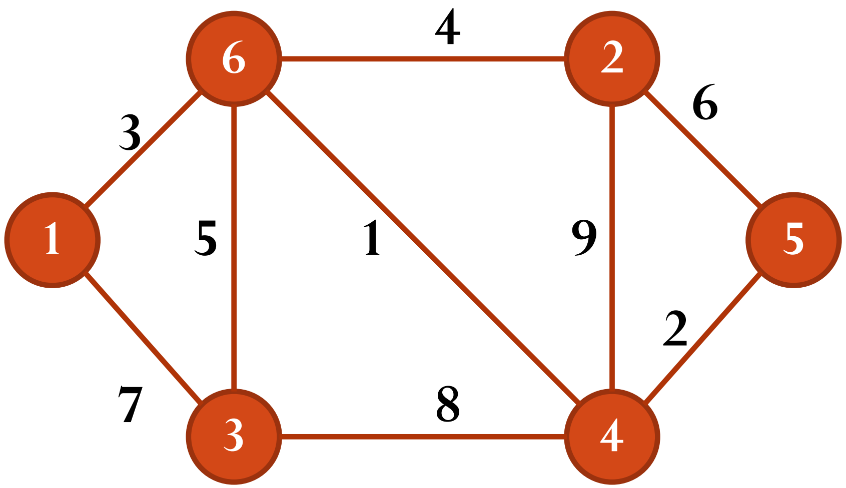
Shortest Path Problem - Dijkstra



- Each cell associated with v of the table has label $(d(v), p(v))$
- Starting node $s = 1$

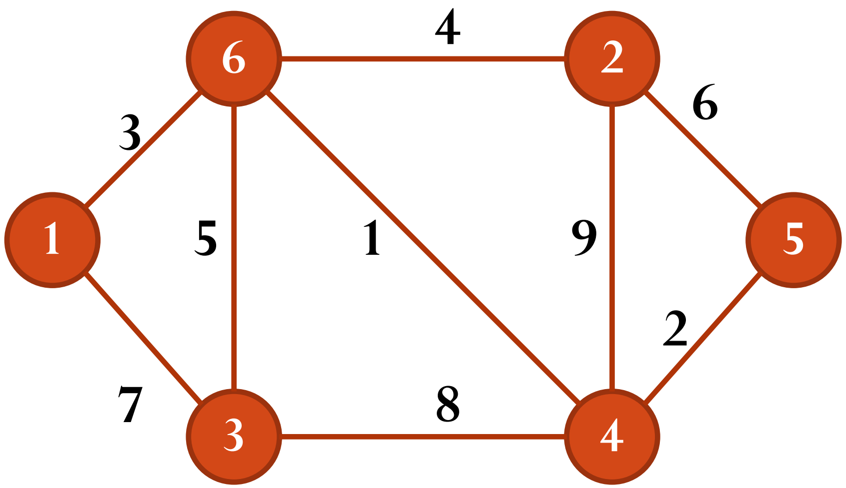
	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1)
Step 1	-					
Step 2	-					
Step 3	-					
Step 4	-					
Step 5	-					

Shortest Path Problem - Dijkstra



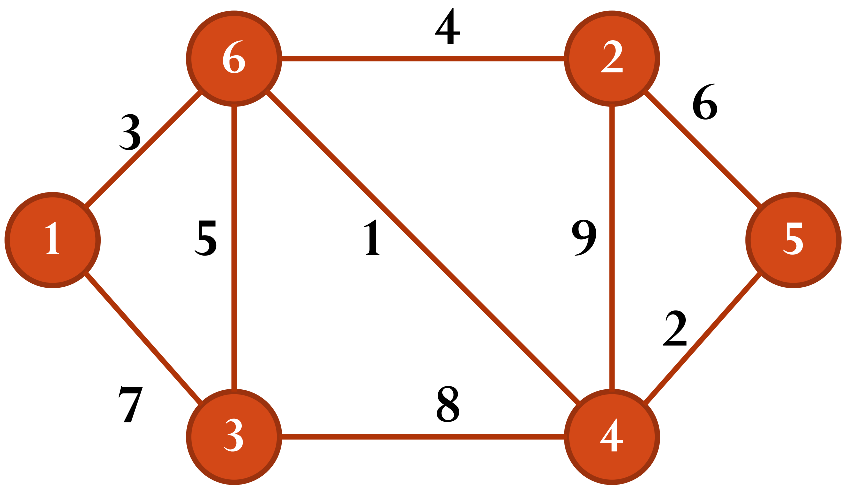
	1	2	3	4	5	6
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Step 1	**-**	**(7,6)**	**(7,1)**	**(4,6)**	**(∞,1)**	**-**
Step 2	**-**					**-**
Step 3	**-**					**-**
Step 4	**-**					**-**
Step 5	**-**					**-**

Shortest Path Problem - Dijkstra



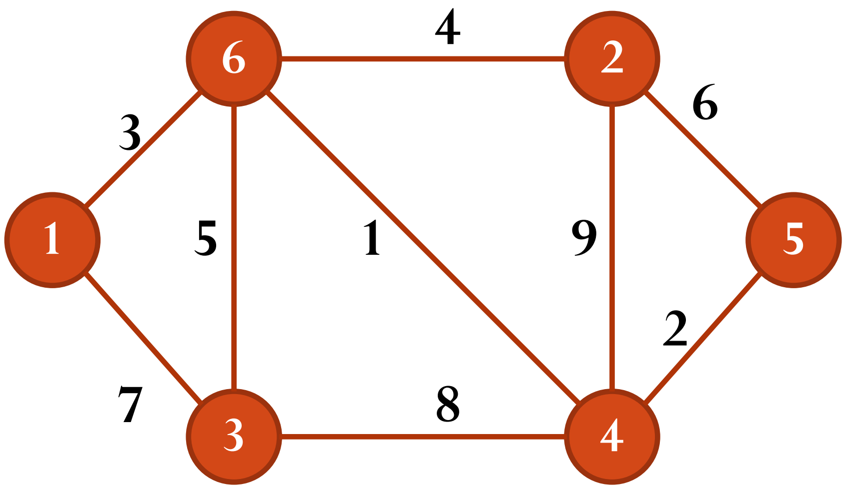
	1	2	3	4	5	6
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Step 1	**-**	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	**-**
Step 2	**-**	**(7,6)**	**(7,1)**	**-**	**(6, 4)**	**-**
Step 3	**-**			**-**		**-**
Step 4	**-**			**-**		**-**
Step 5	**-**			**-**		**-**

Shortest Path Problem - Dijkstra



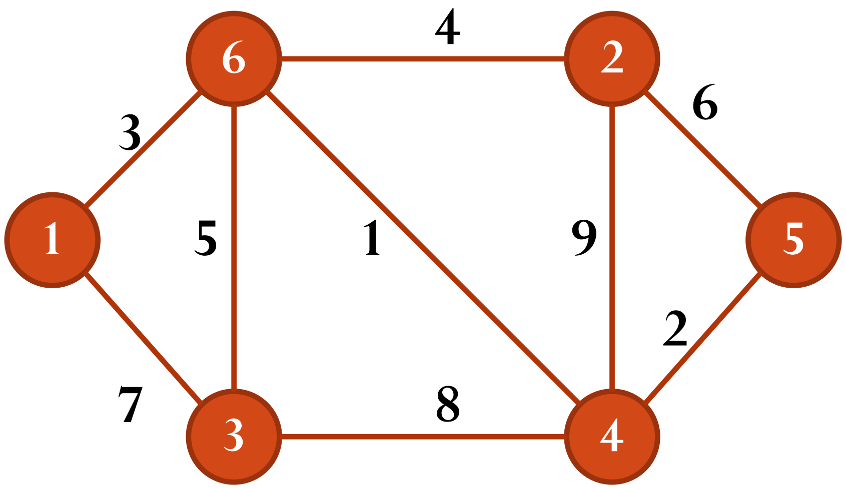
	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6) *	(∞,1)	-
Step 2	-	(7,6)	(7,1)	-	(6, 4) *	-
Step 3	-	(7,6)	(7,1)	-	-	-
Step 4	-			-	-	-
Step 5	-			-	-	-

Shortest Path Problem - Dijkstra



	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6) *	(∞,1)	-
Step 2	-	(7,6)	(7,1)	-	(6, 4) *	-
Step 3	-	(7,6)	(7,1) *	-	-	-
Step 4	-	(7,6)	-	-	-	-
Step 5	-		-	-	-	-

Shortest Path Problem - Dijkstra

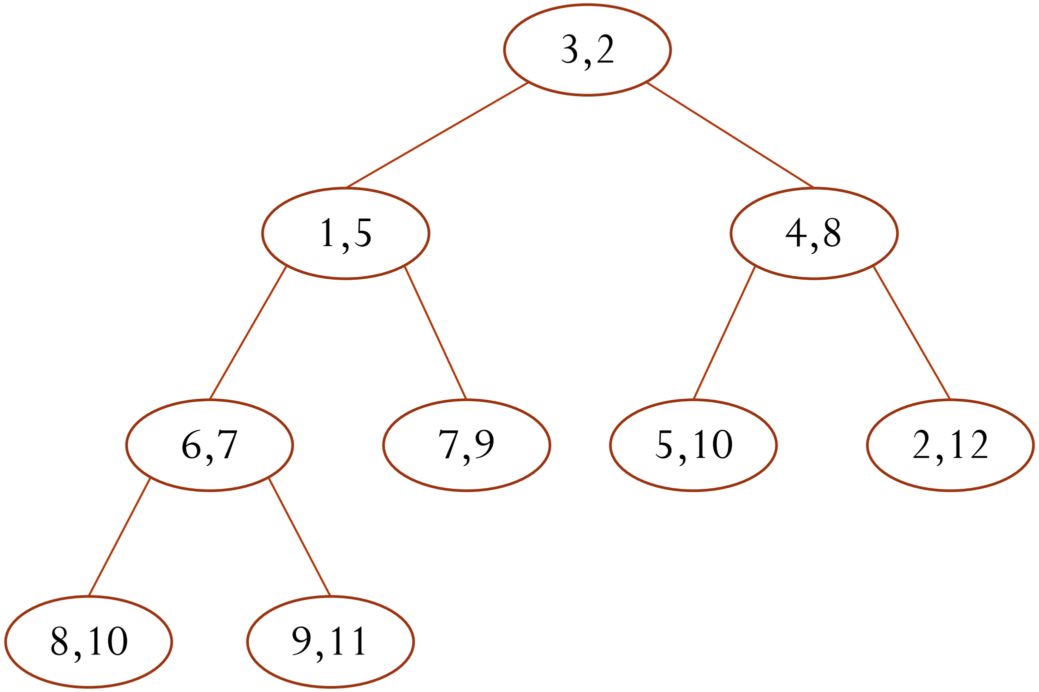


	1	2	3	4	5	6
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Step 1	**-**	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	**-**
Step 2	**-**	**(7,6)**	**(7,1)**	**-**	**(6, 4) ***	**-**
Step 3	**-**	**(7,6)**	**(7,1) ***	**-**	**-**	**-**
Step 4	**-**	**(7,6) ***	**-**	**-**	**-**	**-**
Step 5	**-**	**-**	**-**	**-**	**-**	**-**

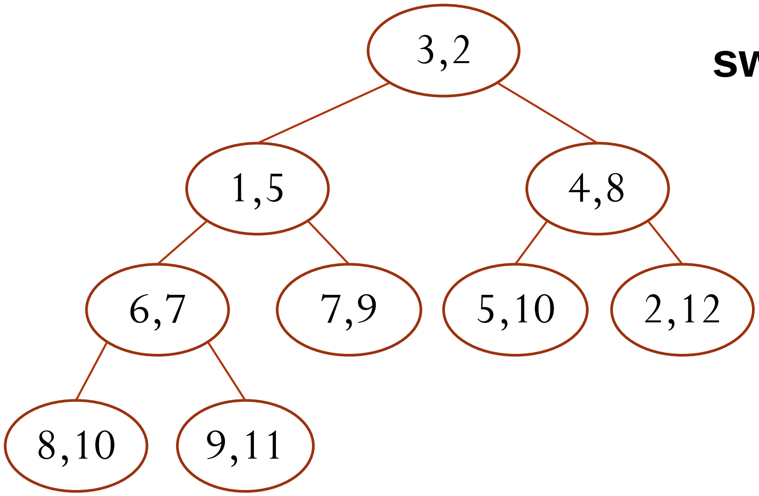
Shortest Path Problem - Dijkstra

- Priority queues
  - Data structure storing elements and their keys
  - Efficient operations
    - (e,k) = deleteMin(): extract element e having minimum key k
    - insert(v,k): insert element e and its key k into the queue
    - updateKey(v,k): update the element with new key k
  - Implementation as binary min-heap
    - Elements are organized in a complete binary tree
    - Key of each element is greater or equal to the keys of its children

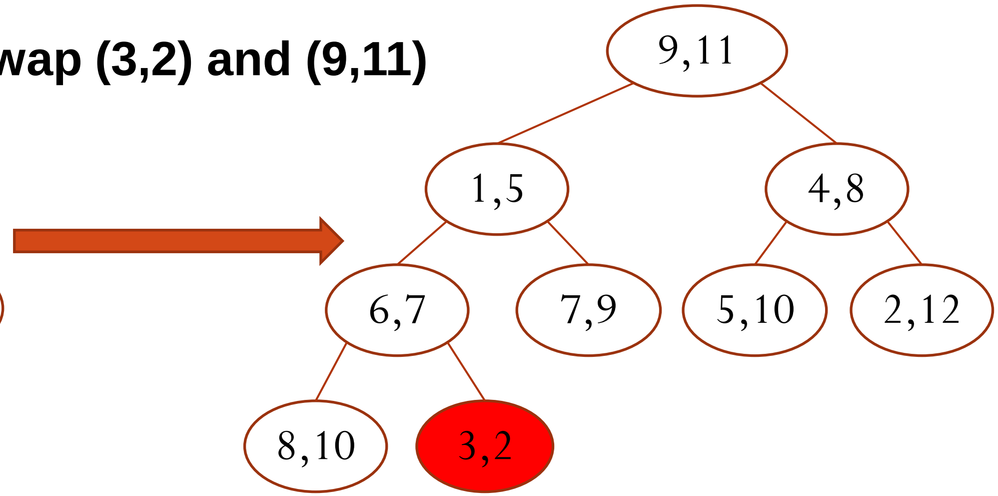
Shortest Path Problem - Dijkstra



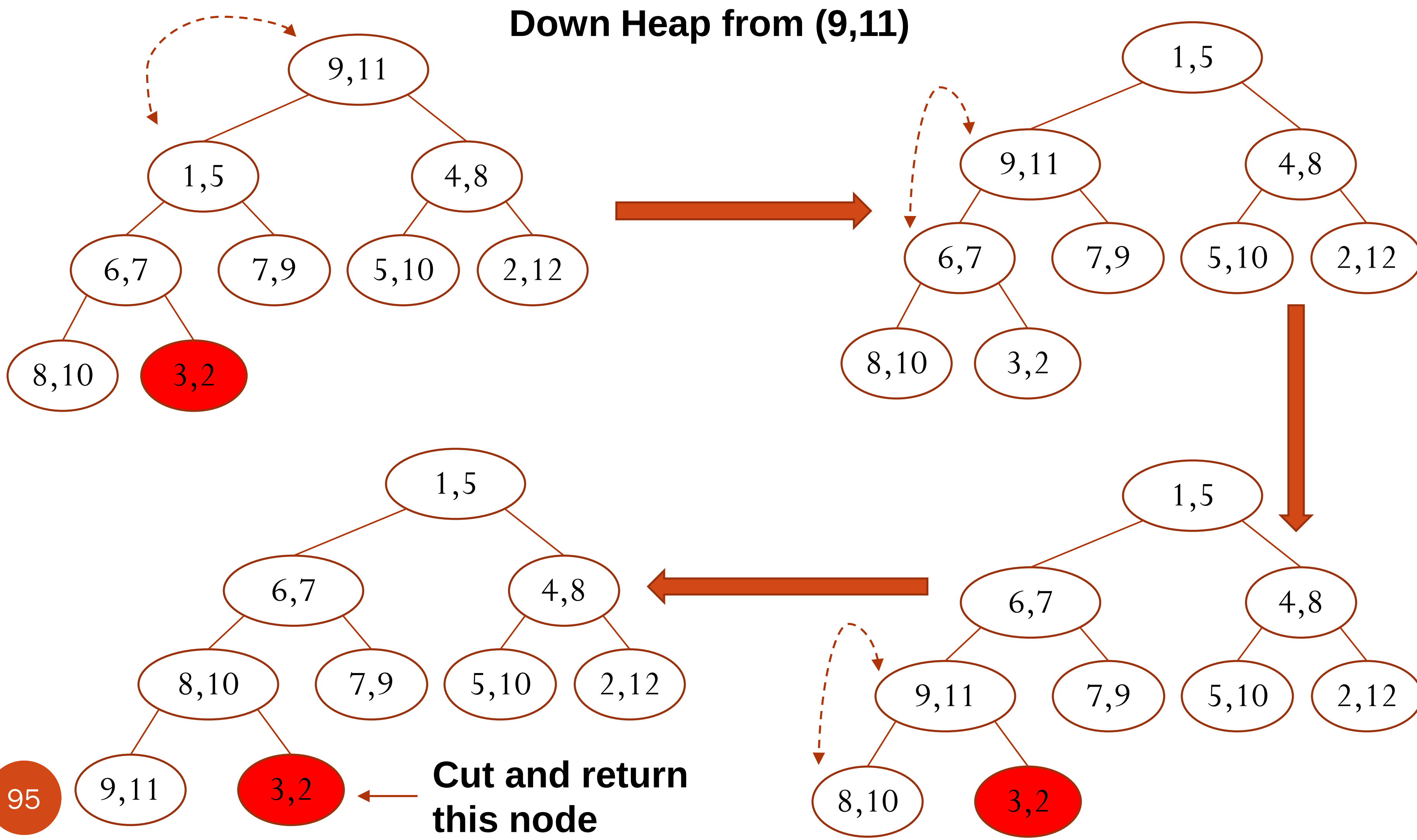
Shortest Path Problem - Dijkstra



swap (3,2) and (9,11)



Shortest Path Problem - Dijkstra



Shortest Path Problem - Dijkstra

```
#include <stdio.h>
#include <vector>
#define MAX 100001
#define INF 1000000
using namespace std;

vector<int> A[MAX]; // A[v][i] is the i^th adjacent node to v
vector<int> c[MAX]; // c[v][i] is the weight of the i^th adjacent arc
                    // (v,A[v][i]) to v
int n,m; // number of nodes and arcs of the given graph
int s,t; // source and destination nodes

// priority queue data structure (BINARY HEAP)
int d[MAX];// d[v] is the upper bound of the length of the shortest path
from s to v (key)
int node[MAX];// node[i] the i^th element in the HEAP
int idx[MAX];// idx[v] is the index of v in the HEAP (idx[node[i]] = i)
int sH;// size of the HEAP
bool fixed[MAX];
```

Shortest Path Problem - Dijkstra

```
void swap(int i, int j){
    int tmp = node[i]; node[i] = node[j]; node[j] = tmp;
    idx[node[i]] = i; idx[node[j]] = j;
}
void upHeap(int i){
    if(i == 0) return;
    while(i > 0){
        int pi = (i-1)/2;
        if(d[node[i]] < d[node[pi]]){
            swap(i,pi);
        }else{
            break;
        }
        i = pi;
    }
}
```

Shortest Path Problem - Dijkstra

```
void downHeap(int i){
    int L = 2*i+1;
    int R = 2*i+2;
    int maxIdx = i;
    if(L < sH && d[node[L]] < d[node[maxIdx]]) maxIdx = L;
    if(R < sH && d[node[R]] < d[node[maxIdx]]) maxIdx = R;
    if(maxIdx != i){
        swap(i,maxIdx); downHeap(maxIdx);
    }
}
void insert(int v, int k){
    // add element key = k, value = v into HEAP
    d[v] = k;
    node[sH] = v;
    idx[node[sH]] = sH;
    upHeap(sH);
    sH++;
}
```

Shortest Path Problem - Dijkstra

```
int inHeap(int v){
    return idx[v] >= 0;
}
void updateKey(int v, int k){
    if(d[v] > k){
        d[v] = k;
        upHeap(idx[v]);
    }else{
        d[v] = k;
        downHeap(idx[v]);
    }
}
```

Shortest Path Problem - Dijkstra

```
int deleteMin(){
    int sel_node = node[0];
    swap(0,sH-1);
    sH--;
    downHeap(0);
    return sel_node;
}
```

Shortest Path Problem - Dijkstra

```
void input(){
    scanf("%d%d",&n,&m);
    for(int k = 1; k <= m; k++){
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        A[u].push_back(v);
        c[u].push_back(w);

    }
    scanf("%d%d",&s,&t);
}
```

Shortest Path Problem - Dijkstra

```
void init(int s){
    sH = 0;
    for(int v = 1; v <= n; v++){
        fixed[v] = false;  idx[v] = -1;
    }
    d[s] = 0;    fixed[s] = true;
    for(int i = 0; i < A[s].size(); i++){
        int v = A[s][i];
        insert(v,c[s][i]);
    }
}
```

Shortest Path Problem - Dijkstra

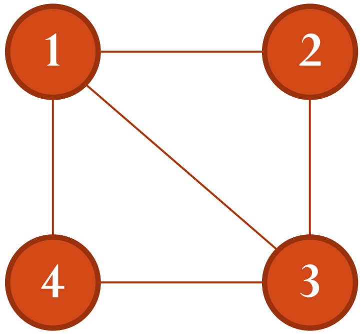
```
void solve(){
    init(s);
    while(sH > 0){
        int u = deleteMin();
        fixed[u] = true;
        for(int i = 0; i < A[u].size(); i++){
            int v = A[u][i];
            if(fixed[v]) continue;
            if(!inHeap(v)){
                    int w = d[u] + c[u][i];
                    insert(v,w);
            }else{
                if(d[v] > d[u] + c[u][i]){
                    updateKey(v,d[u]+c[u][i]);
                }
            }
        }
    }
    int rs = d[t]; if(!fixed[t]) rs = -1;
    printf("%d",rs);
}
```

Shortest Path Problem - Dijkstra

```
int main(){
    input();
    solve();
}
```

Exercises
COUNT SPANNING TREE

- Given a undirected connected graph G = (V,E) in which V = {1,…,N} is the set of nodes. Count the number of spanning trees of G.
- There are 8 spanning trees represented by list of edges as follows
  - (1,2) (1,3) (1,4)
  - (1,2) (1,3) (3,4)
  - (1,2) (1,4) (2,3)
  - (1,2) (1,4) (3,4)
  - (1,2) (2,3) (3,4)
  - (1,3) (1,4) (2,3)
  - (1,3) (2,3) (3,4)
  - (1,4) (2,3) (3,4)



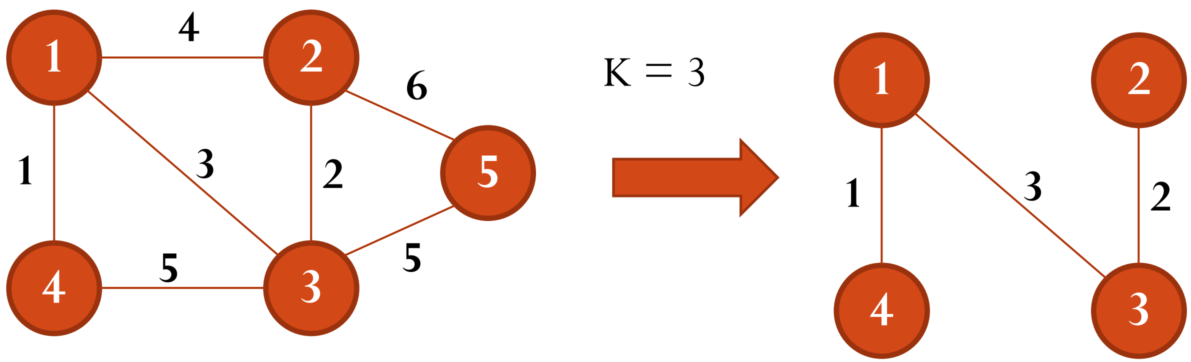
Exercises
COUNT SPANNING TREE

- Input
  - Line 1: contains positive integers N and M ($1 \le N \le 20$, $1 \le M \le 25$)
  - Line i+1 (i = 1,…,M): contains u and v which are endpoints of the i^{th} edge of G
- Output
  - Write the number of spanning trees of G

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

Exercises
K-MST

- Given a undirected graph G=(V,E), w(e) is the weight of the edge e (e $\in$ E). Given a positive integer K, find the subgraph of G which is a tree containing exactly K edges having minimal weight.



Exercises
BOUNDED-MST

- The diameter of a tree is defined to be the length of the longest path (in term of number of edges of the path) on that tree. Given a undirected graph G=(V,E), w(e) is the weight of the edge e (e $\in$ E). Given a positive integer K, find the minimum spanning tree T of G such that the diameter of T is less than or equal to K.