HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

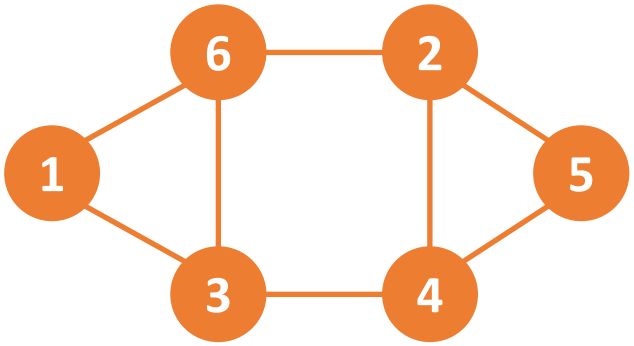
Graphs

Content

- Graphs and terminology
- Depth First Search
- Breadth First Search
- Dijkstra algorithm using priority queue
- Kruskal algorithm using disjoint-set structure

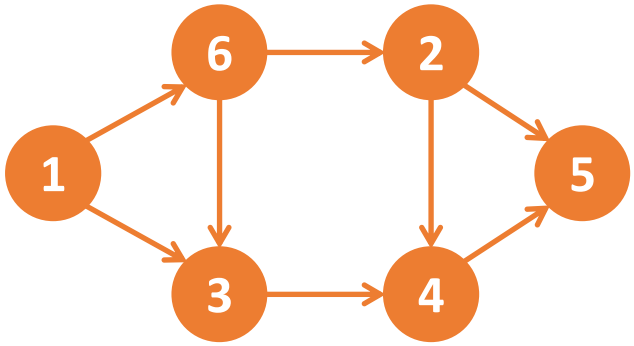
Graphs

- Mathematical objects including set of nodes and links between these nodes
- Graph $G = (V,E)$, V is the set of nodes, E is the set of edges (arcs)
  - $(u,v) \in E$, we say u is adjacent to 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

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 3), (1,6), (2, 4), (2, 5), (6, 2), (3, 4), (6, 3), (4, 5)\}$

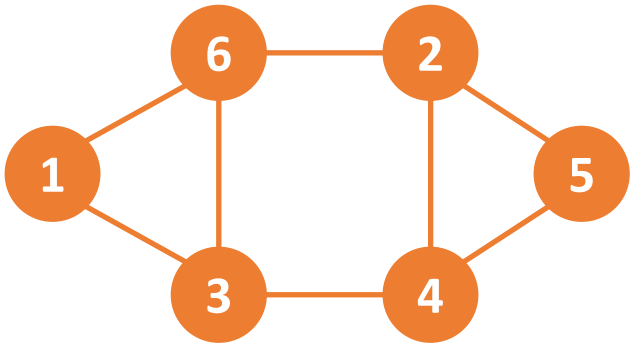
Graphs

- Degree of a node is the number of adjacent nodes to that node

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

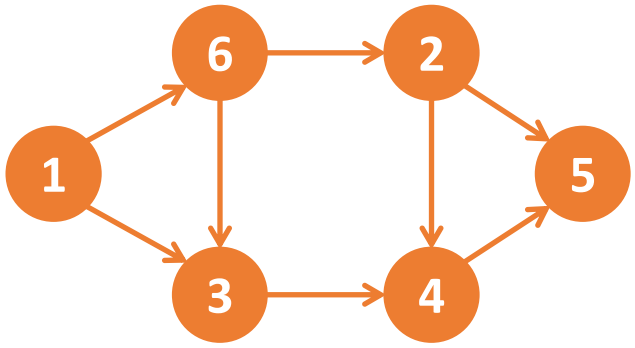
- Incoming (outgoing) degree of a node in a directed graph is the number of ars entering (leaving) that node:

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



Undirected graph

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

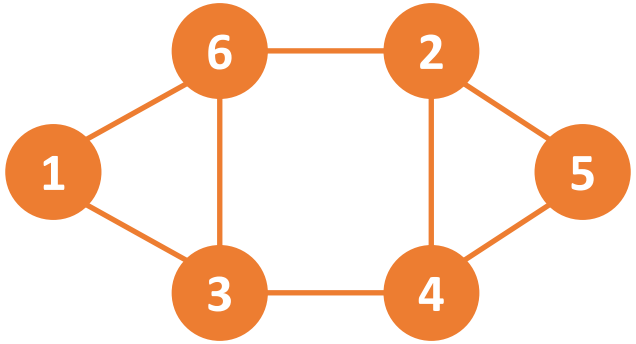


Directed graph

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

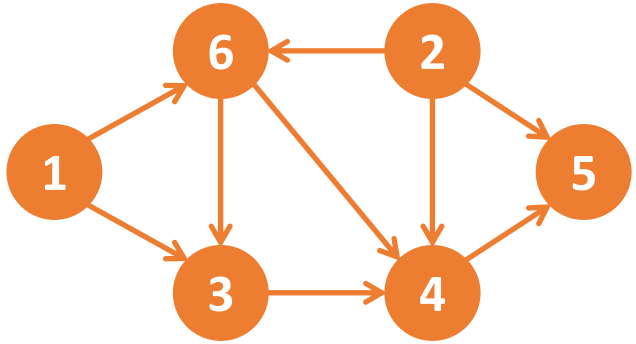
Graphs

- Given a graph $G=(V, E)$ and 2 nodes $s, t \in V$, a path from s to t in G is a sequence $s = x_0, x_1, \ldots, x_k = t$ in which $(x_i, x_{i+1}) \in E, \forall\, i = 0, 1, \ldots, k\text{-}1$



Path from 1 to 5:

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

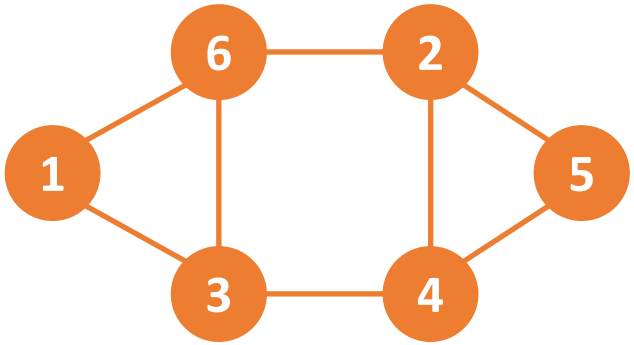


Path from 1 to 5:

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

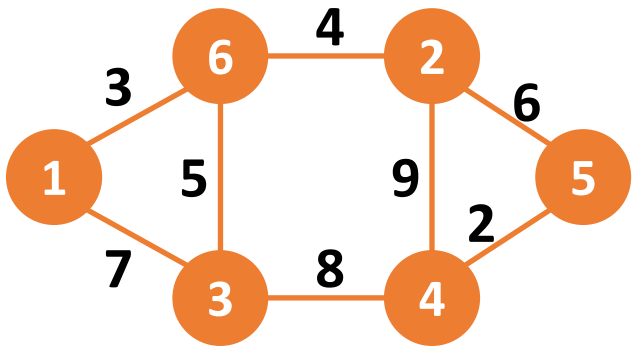
Graph representation

- Adjacent matrix



	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

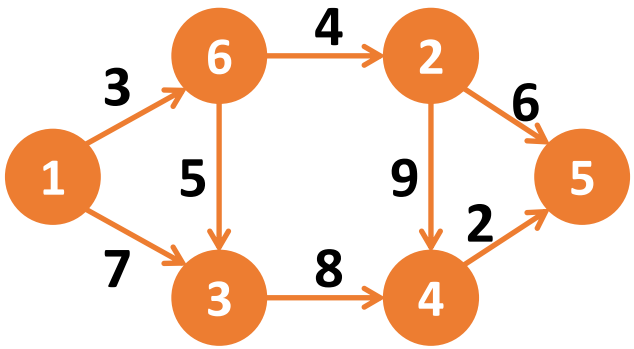
- Weight matrix



	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

Graph representation

- Adjacency list
  - For each $v \in V$, $A(v)$ is a set of triple (v, u, w) in which w is the weight of arc (v, u)
  - $A(1) = \{(1, 6, 3), (1, 3, 7)\}$
  - $A(2) = \{(2, 4, 9), (2, 5, 6)\}$
  - $A(3) = \{(3, 4, 8)\}$
  - $A(4) = \{(4, 5, 2)\}$
  - $A(5) = \{\}$
  - $A(6) = \{(6, 3, 5), (6, 2, 4)\}$



Graph traversal

- Visit nodes of a given graph in some order
- Each node is visited exactly once
- Two main methods
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Depth-First Search (DFS)

- DFS(*u*): DFS start from node *u*
  - If there exists a node *v* in the adjacency list of *u* which has not been visited, then visit *v* and call DFS(*v*)
  - If all the adjacent nodes to *u* have been visited, then DFS backtrack to node *x* from which the algorithm visits *u* to visit other nodes which are adjacent to *x* and have not been visited. At this time, the node *u* is call *finished-exploration*

Depth-First Search (DFS)

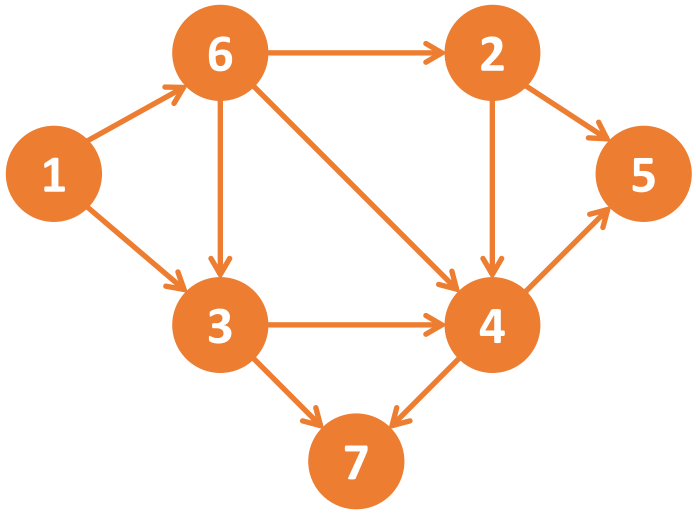
- Information related to each node *v*
  - *p*(*v*): the node from which the algorithm visits *v*
  - *d*(*v*): time point where *v* is *visited* but not *finished-exploration*
  - *f*(*v*): time point at which *v* is *finished-exploration*
  - color(*v*)
    - WHITE: *not-visited*
    - GRAY: *visited but* not *finished-exploration*
    - BLACK: *finished-exploration*

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

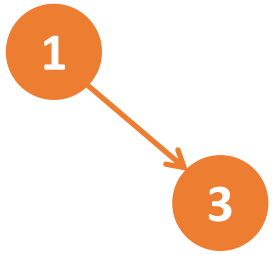
Depth-First Search (DFS)

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)

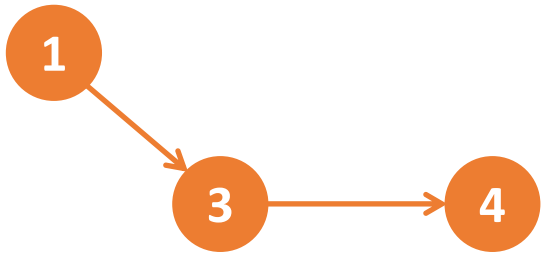
DFS(1)



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)

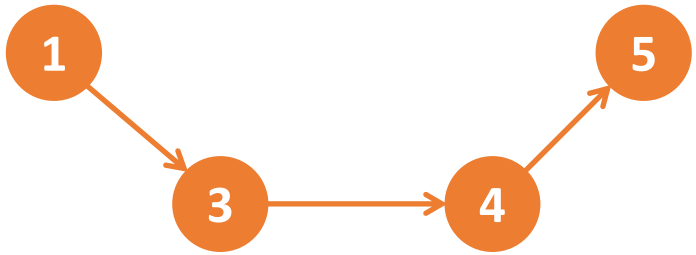
DFS(1)



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)

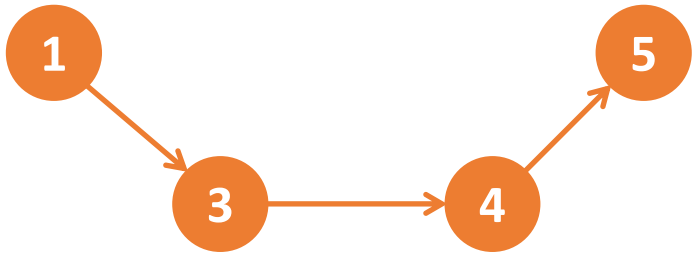
DFS(1)



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)

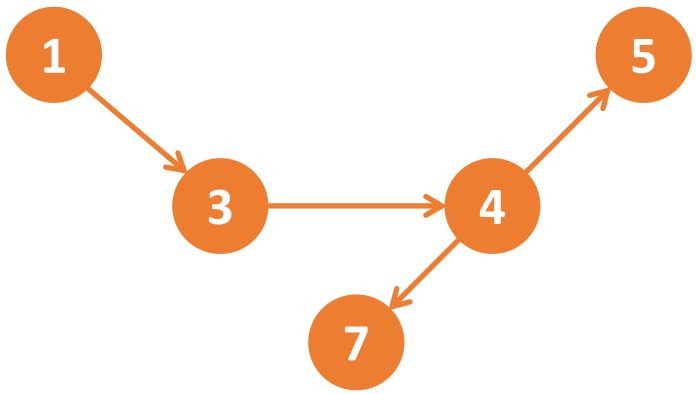
DFS(1)



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)

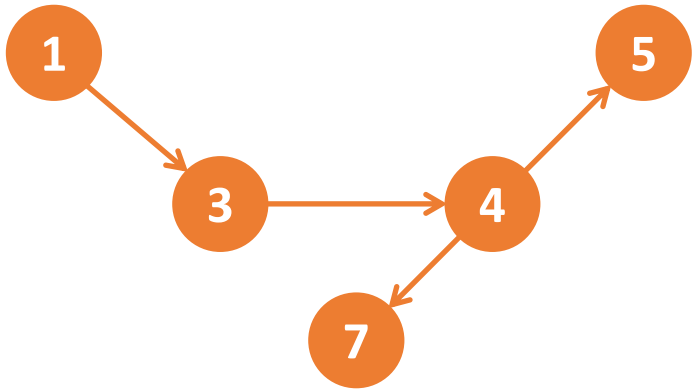
DFS(1)



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)

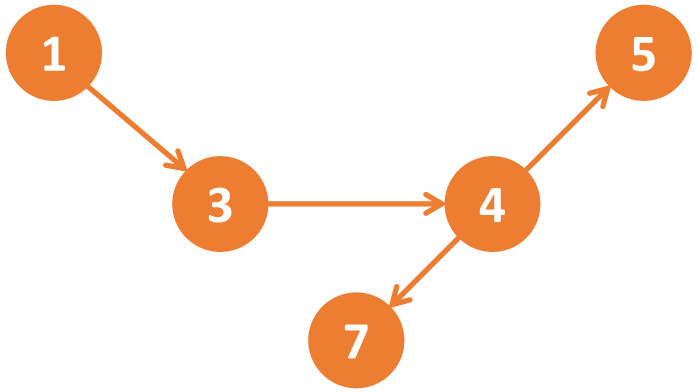
DFS(1)



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)

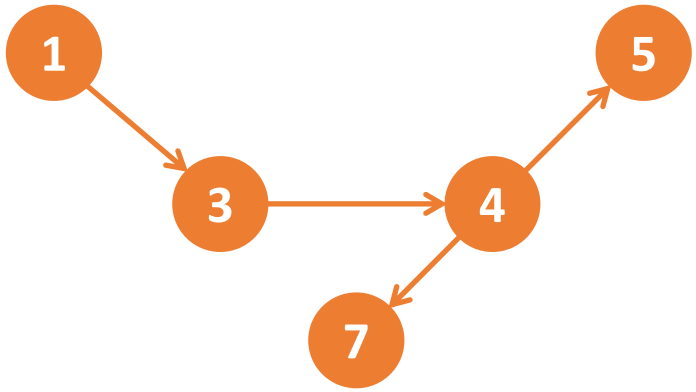
DFS(1)



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)

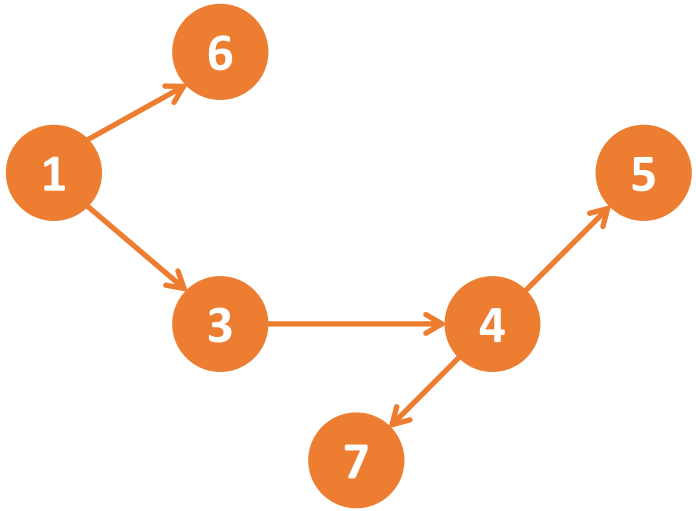
DFS(1)



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)

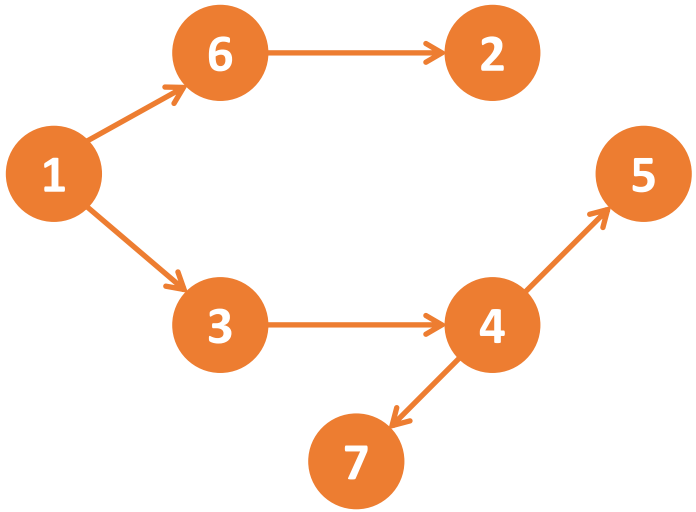
DFS(1)



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)

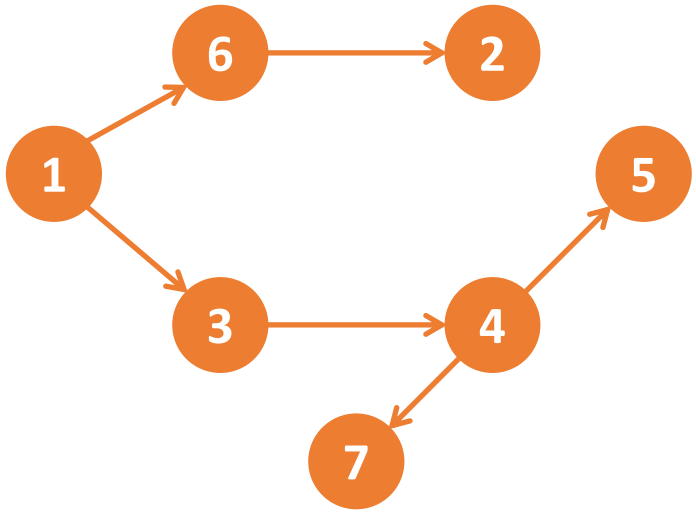
DFS(1)



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)

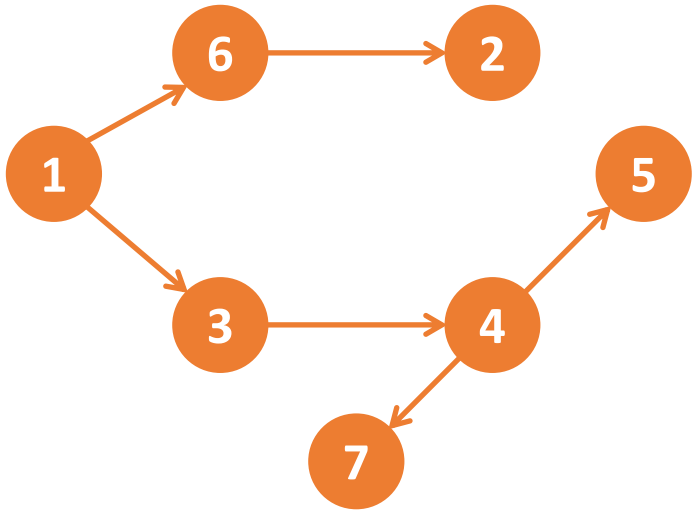
DFS(1)



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)

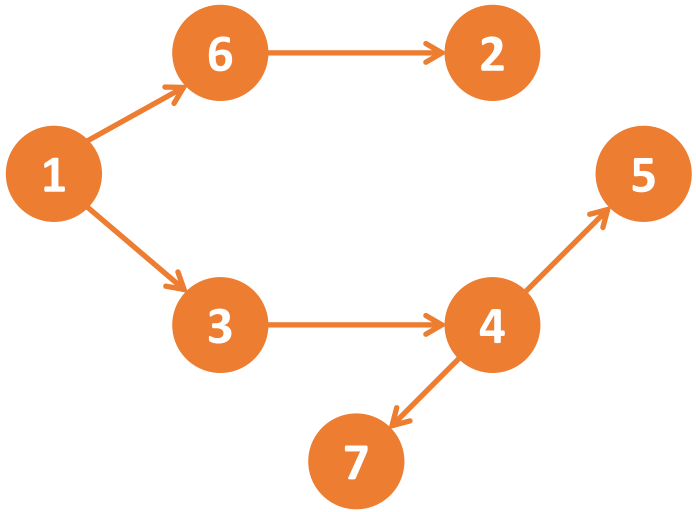
DFS(1)



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)



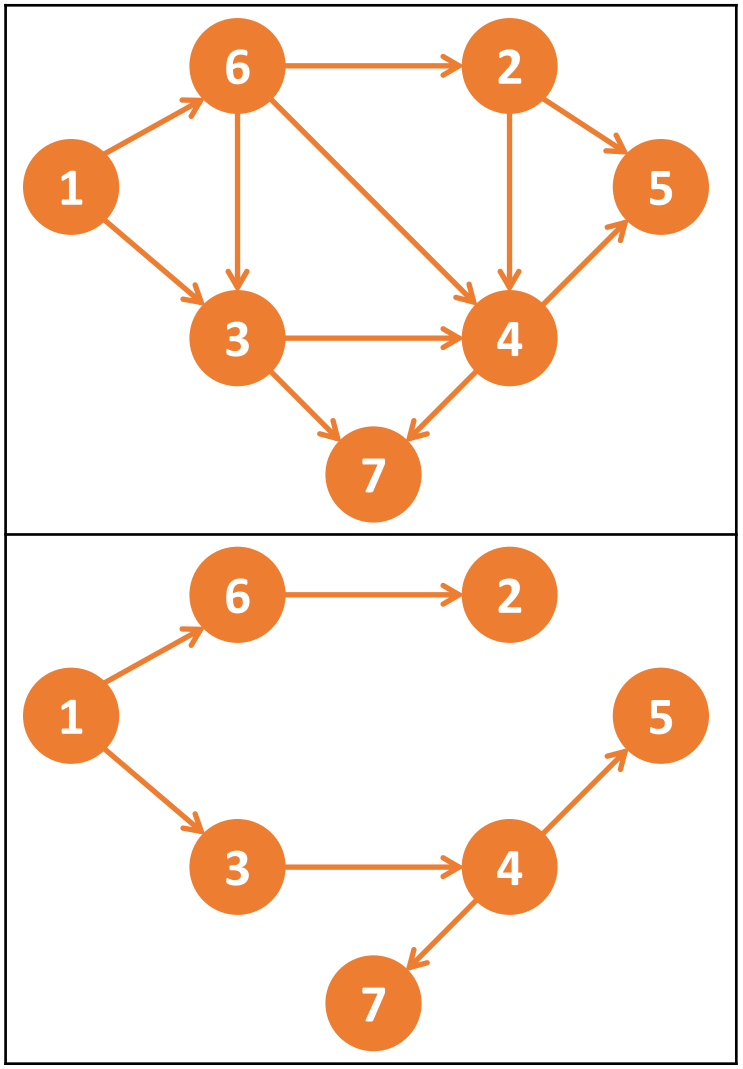
Node	1	2	3	4	5	6	7
d	1	11	2	3	4	10	6
f	14	12	9	8	5	13	7
p	-	6	1	3	4	1	4
color	B	B	B	B	B	B	B

Depth-First Search (DFS)

- Result of DFS is a set of DFS trees
- Edge classification
  - tree edge: (u,v) is a tree edge if v is visited from u
  - back edge: (u,v) is a back edge if v is an ancestor of u
  - forward edge: (u,v) is a forward edge if u is an ancestor of v
  - crossing edge: remaining edges

Depth-First Search (DFS)

- Edge classification
  - Tree edges: (1, 6), (1, 3), (6, 2), (3, 4), (4, 5), (4, 7)
  - Back edges:
  - Forward edges: (3, 7)
  - Crossing edges: (6, 3), (6, 4), (2, 4), (2,5)



Breadth-First Search (BFS)

- BFS(*u*): BFS from node *u*
  - Visit *u*
  - Visit adjacent nodes to u which have not been visited (called 1-level nodes)
  - Visit adjacent nodes to 1-level nodes which have not been visited (called 2-level nodes)
  - Visit adjacent nodes to 2-level nodes which have not been visited (called 3-level nodes)
  - . . .
- Use queue data structures

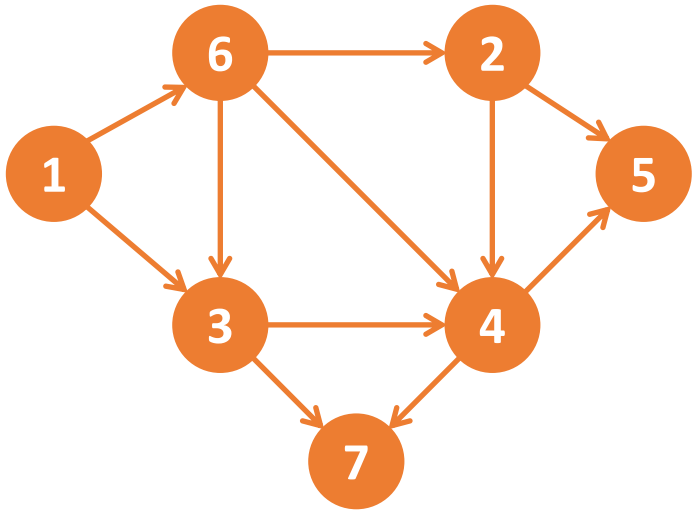
Breadth-First Search (BFS)

```
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)

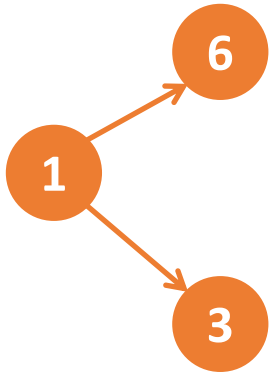


Breadth-First Search (BFS)

BFS(1)

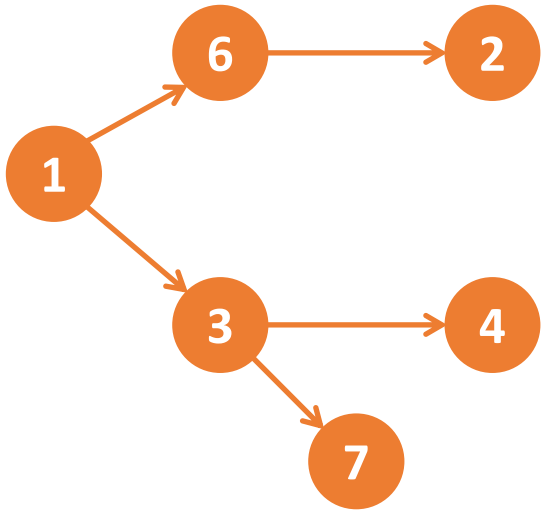
Breadth-First Search (BFS)

BFS(1)



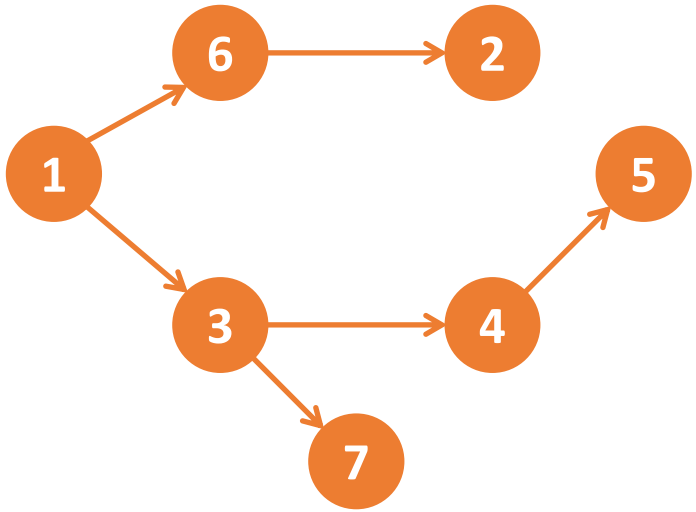
Breadth-First Search (BFS)

BFS(1)



Breadth-First Search (BFS)

BFS(1)

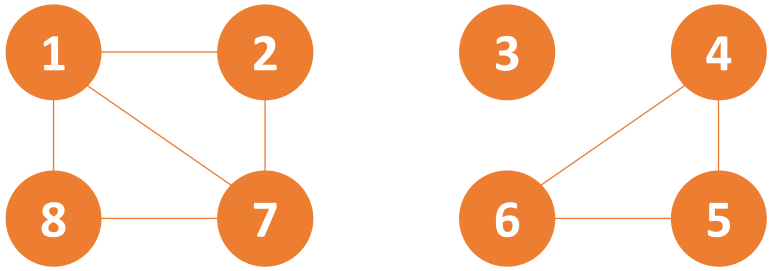


Applications of DFS, BFS

- Compute connected components of a given undirected graph
- Compute strongly connected components of a given directed graph
- Checking bipartite graph
- Cycle detection
- Topological sort
- Find longest path on trees

CONNECTED COMPONENTS

- 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}



CONNECTED COMPONENTS

- 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} 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

CONNECTED COMPONENTS

```
#include <bits/stdc++.h>
#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);
    }
}
```

CONNECTED COMPONENTS

```
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);
        }
    }
}
```

CONNECTED COMPONENTS

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

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

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

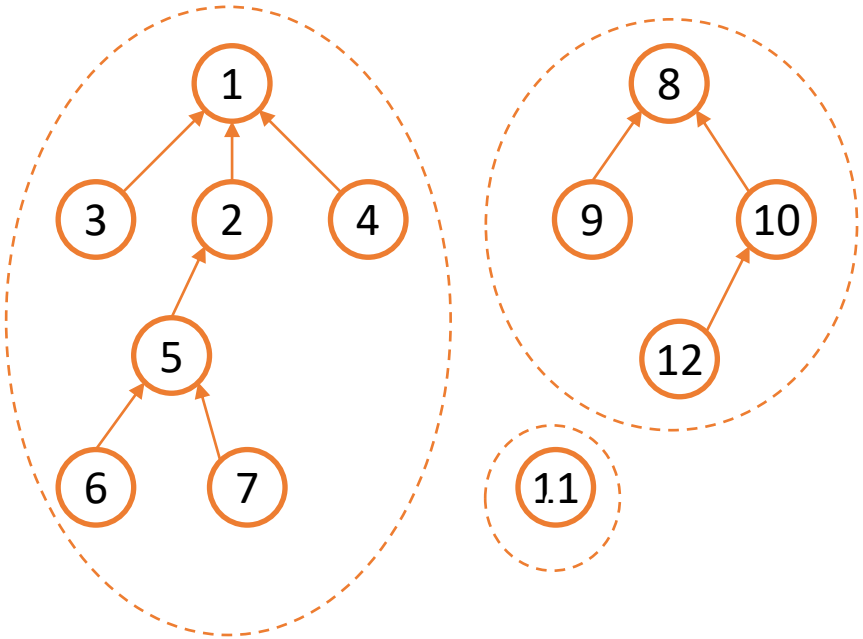
Minimum Spanning Tree: Kruskal algorithm

- 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;
}
```

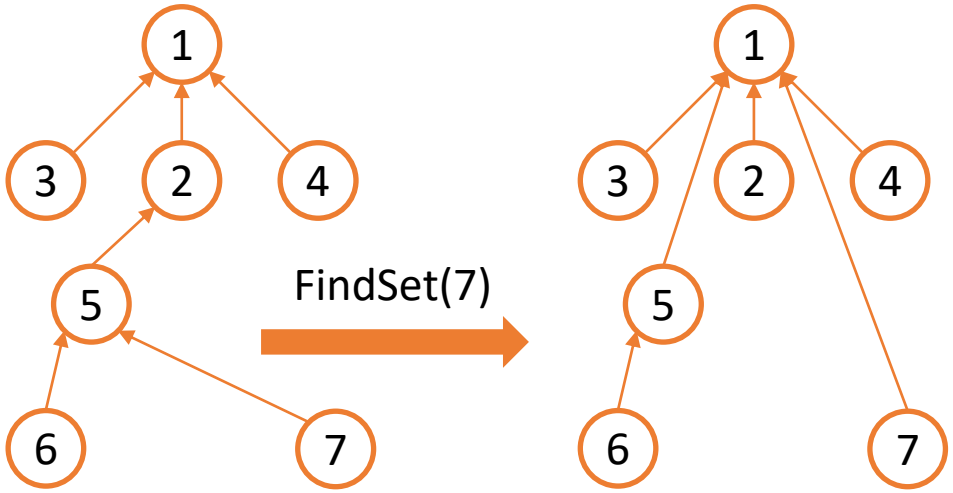
Disjoint Sets

- Data structure representing disjoint sets with two main operations
  - FindSet(x): return the identifier of the set containing x
  - Unify(r1, r2): unify two sets whose identifiers are r1 and r2 into one
- Each set is represented by a rooted tree
  - Nodes of the trees are elements of the set
  - Each node has a unique father node (parent of the root node is itself, by convention)
  - The root node of the tree is identifier of that set



Disjoint Sets

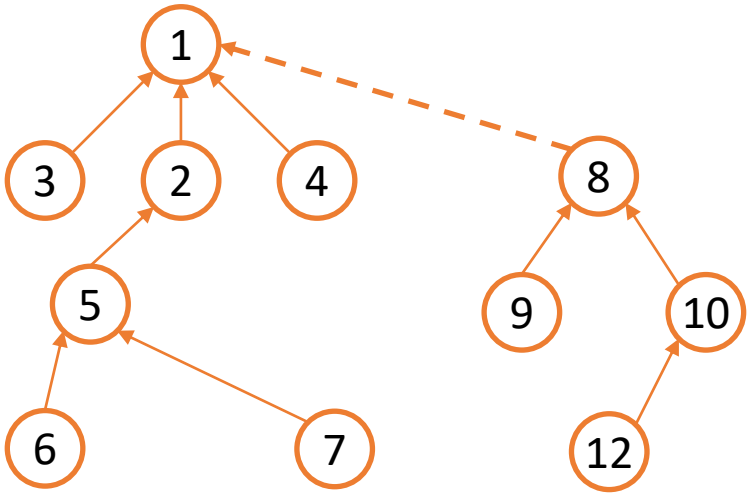
- Operation FindSet(x)
  - Traverse from x along the path to the root of the tree containing x (running time is proportional to the length of that path)
  - Adjust the tree during the query operation: marking nodes on the path from x to the root to as the children of the root → save time for subsequent queries



Disjoint Sets

- Operation Unify(r1, r2)
  - Mark the node having lower height as a child of the remaining node in order to keep the height of the unified tree small

- Implementation: maintain two data structures
  - p(x): parent of x
  - r(x): rank of x



The height of 8 is smaller than the height of 1, so mark 8 as a child of 1

Minimum Spanning Tree: Kruskal algorithm

```
#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 algorithm

```
void unify(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 algorithm

```
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 algorithm

```
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 algorithm

```
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){
            unify(ru,rv);
            nET++;  ET[nET] = i;
            rs += c[i];
            count++;
            if(count == N-1) break;
        }
    }
    cout << rs;
}
```

Minimum Spanning Tree: Kruskal algorithm

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

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

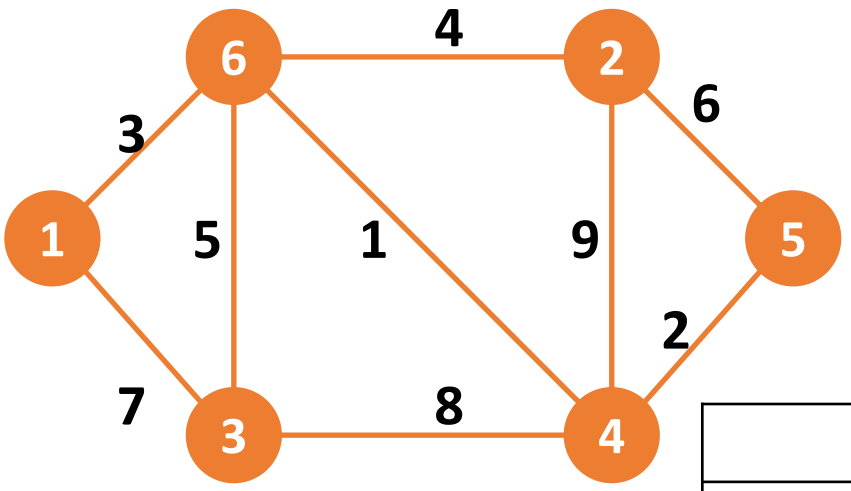
Shortest Path Problem - Dijkstra

- Main idea Dijkstra:
  - Each $v \in V$:
    - **P**(v) upper bound of the shortest path from s to v
    - $d(v)$: weight of **P**(v)
    - $p(v)$: predecessor of v on **P**(v)
  - Initialization
    - **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;
      }
    }
  }
}
```

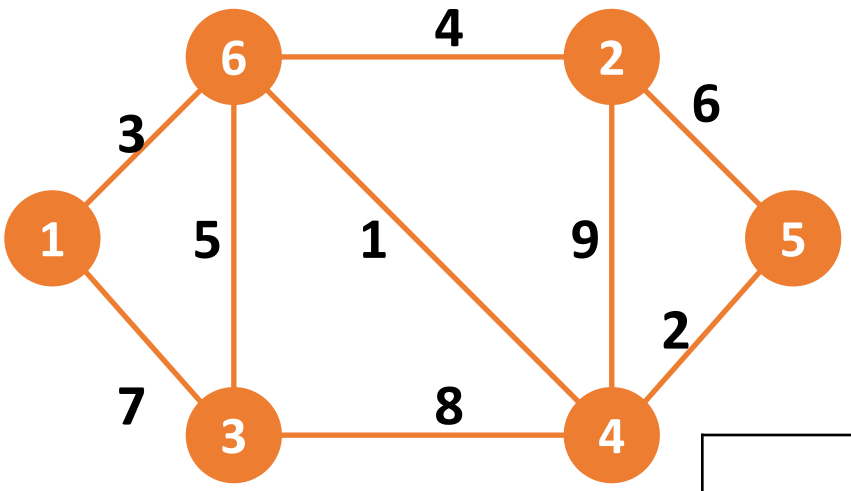
Shortest Path Problem - Dijkstra



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

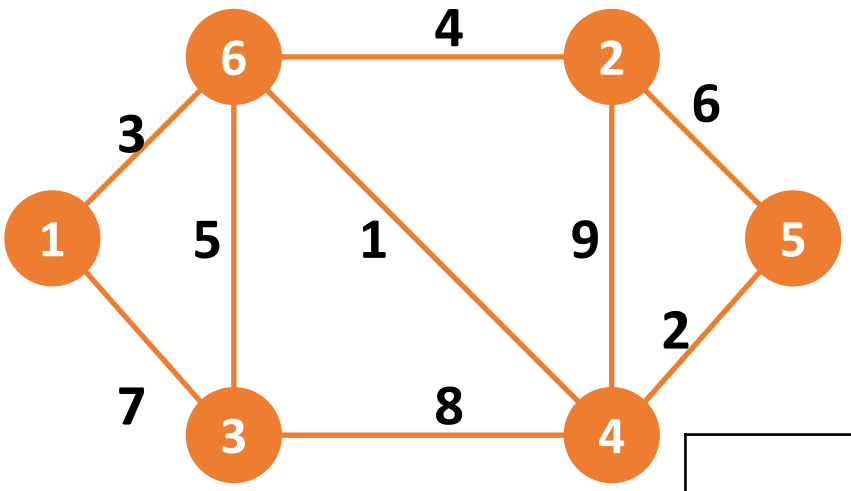
	1	2	3	4	5	6
Init	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1)**
Step 1	-					
Step 2	-					
Step 3	-					
Step 4	-					
Step 5	-					

Shortest Path Problem - Dijkstra



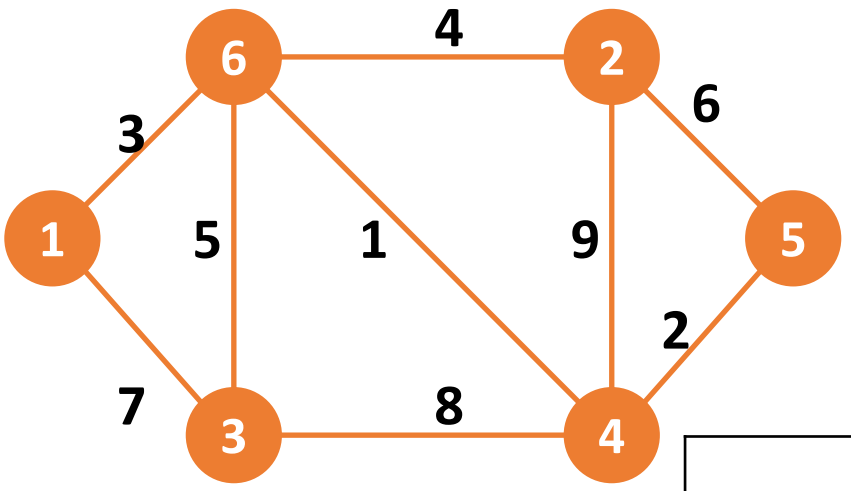
	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6)	(∞,1)	-
Step 2	-					-
Step 3	-					-
Step 4	-					-
Step 5	-					-

Shortest Path Problem - Dijkstra



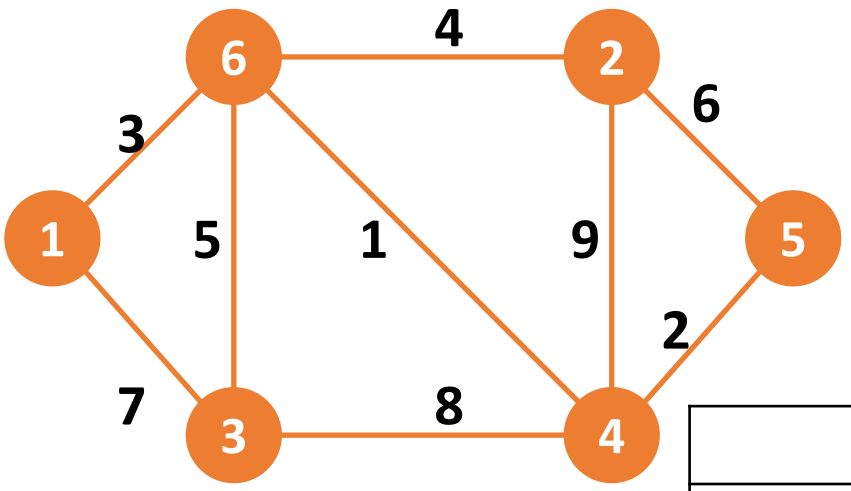
	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6) *	(∞,1)	-
Step 2	-	(7,6)	(7,1)	-	(6, 4)	-
Step 3	-			-		-
Step 4	-			-		-
Step 5	-			-		-

Shortest Path Problem - Dijkstra



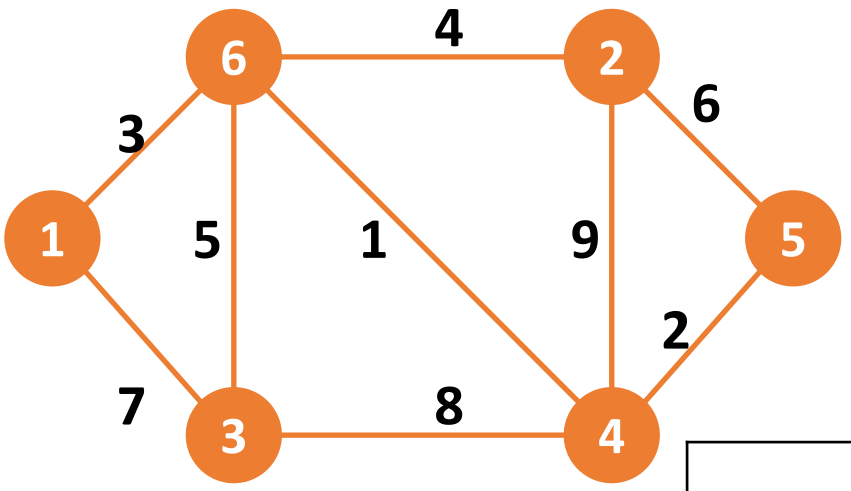
	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6) *	(∞,1)	-
Step 2	-	(7,6)	(7,1)	-	(6, 4) *	-
Step 3	-	(7,6)	(7,1)	-	-	-
Step 4	-			-	-	-
Step 5	-			-	-	-

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

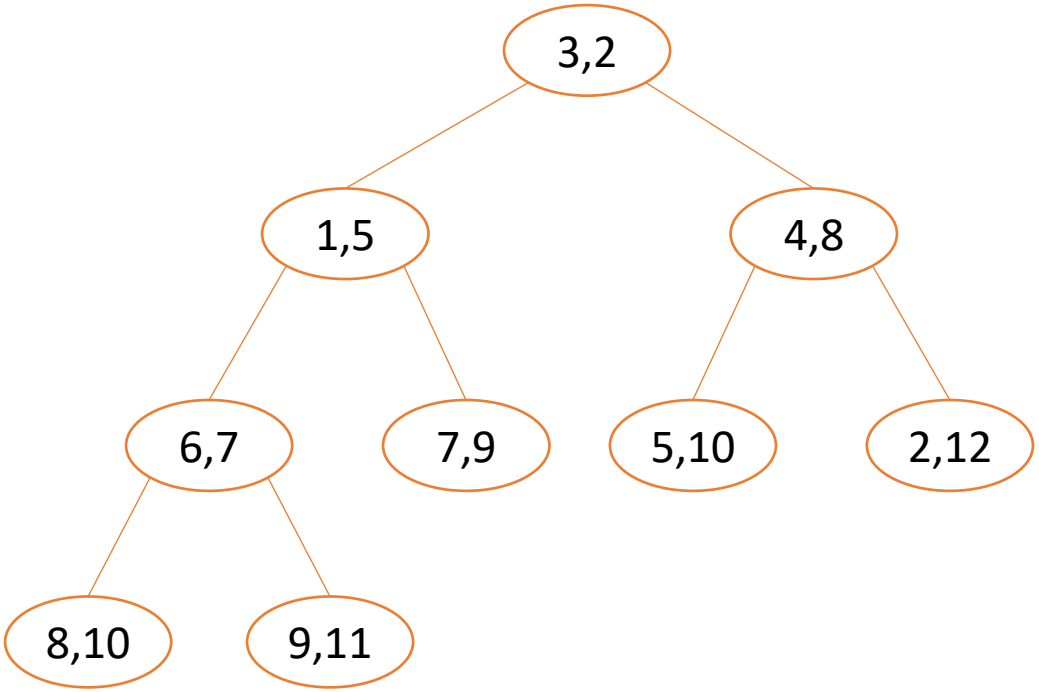


	1	2	3	4	5	6
Init	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *
Step 1	-	(7,6)	(7,1)	(4,6) *	(∞,1)	-
Step 2	-	(7,6)	(7,1)	-	(6, 4) *	-
Step 3	-	(7,6)	(7,1) *	-	-	-
Step 4	-	(7,6) *	-	-	-	-
Step 5	-	-	-	-	-	-

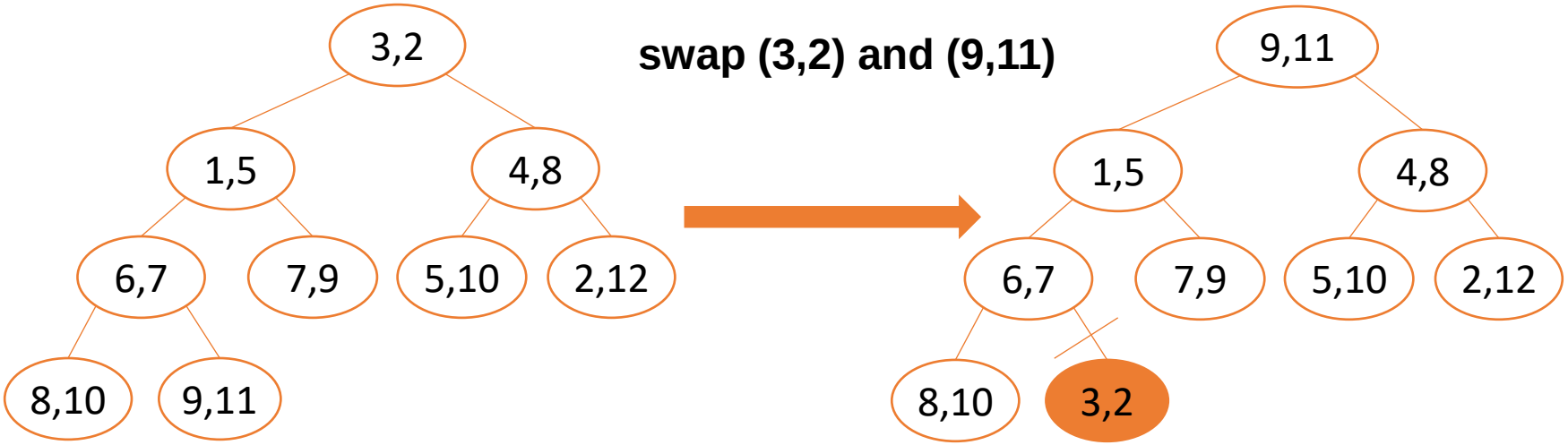
Shortest Path Problem - Dijkstra

- **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 less than or equal to the keys of its children

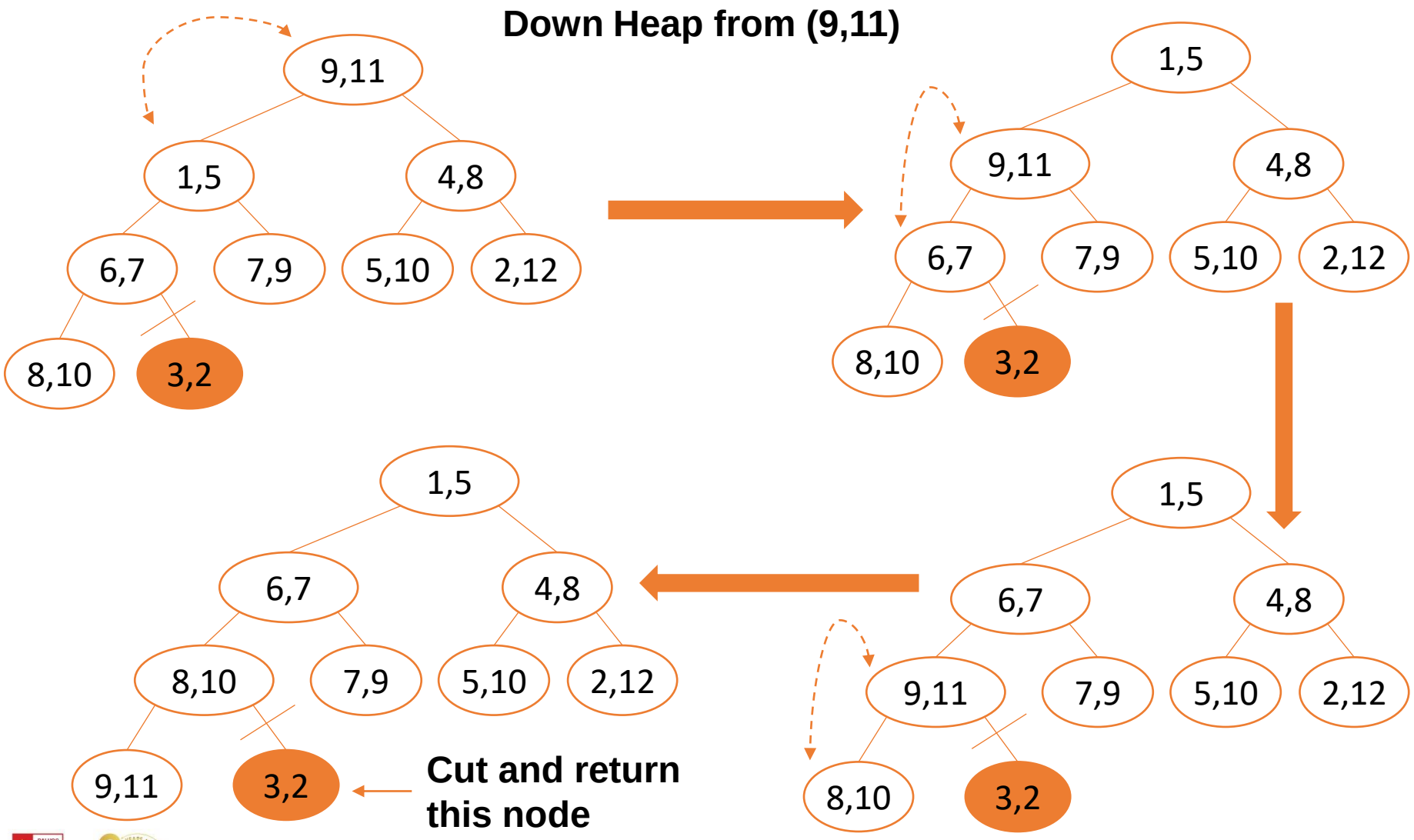
Shortest Path Problem - Dijkstra



Shortest Path Problem - Dijkstra



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();
}
```