자료구조

L11: Graph

2022년 1학기

국민대학교 소프트웨어학부

Overview

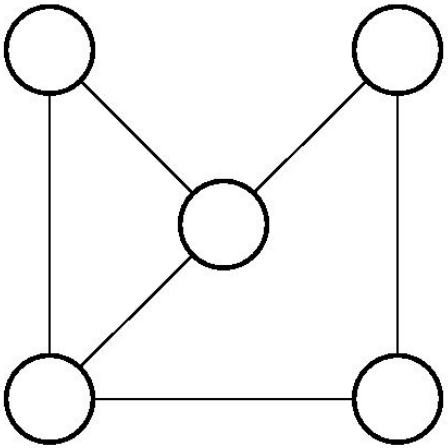
- **Basic terms and definitions of graphs**
- How to represent graphs
- Graph traversal methods

Graphs

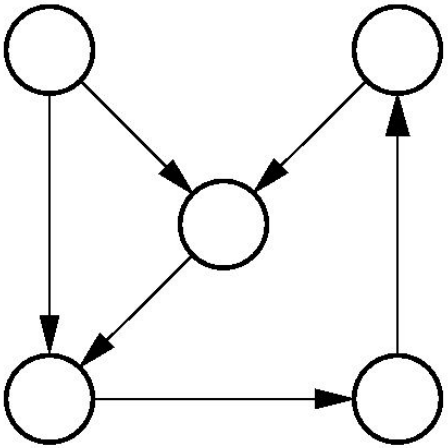
- A graph G = (V, E) consists of a set of vertices V, and a set of edges E, such that each edge in E is a connection between a pair of vertices in V.
  - Example: social network, phone call graph, computer network, ...

- The number of vertices is written |V|, and the number edges is written |E|.

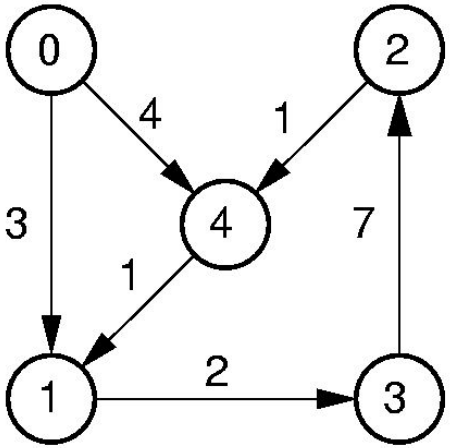
Graphs



Undirected Graph



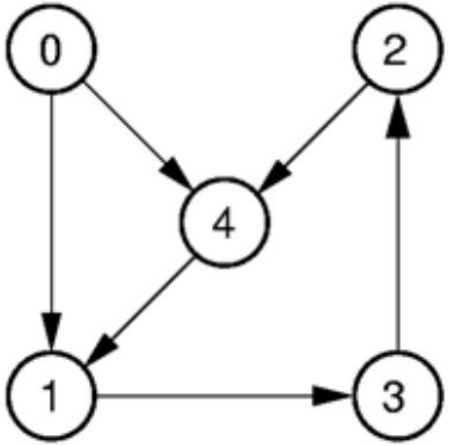
Directed Graph



Weighted Graph

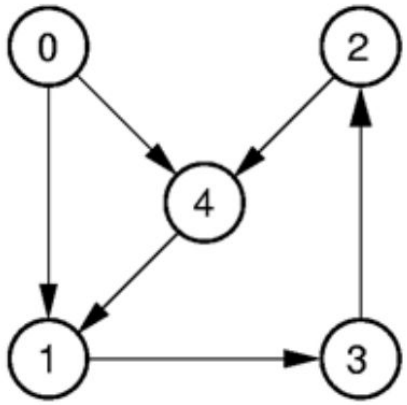
Paths and Cycles

- **Path**: A sequence of vertices $v_1, v_2, \cdots, v_n$ of length n-1 with an edge from v_i to v_{i+1} for $1 <= i < n$.
  - E.g., 0, 4, 1, 3, 2, 4 in the graph below is a path
- A path is **simple** if all vertices on the path are distinct.
  - E.g., 0, 4, 1, 3, 2 in the graph below is a simple path



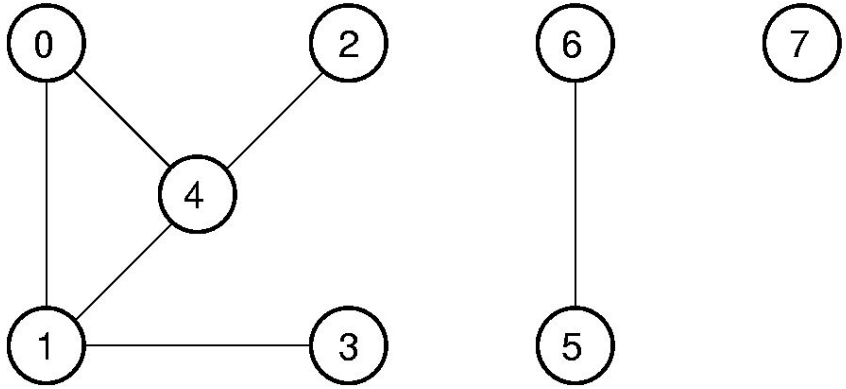
Paths and Cycles

- A **cycle** is a path of length 3 or more that connects v_i to itself.
  - E.g., 1,3,2,4,1,3,2,4,1 in the graph below is a cycle
- A cycle is **simple** if the path is simple, except the first and last vertices are the same.
  - E.g., 1,3,2,4,1 in the graph below is a simple cycle



Connected Components

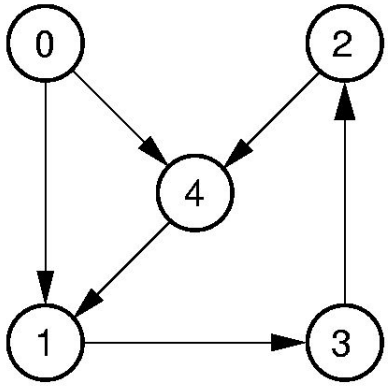
- An **undirected graph** is **connected** if there is at least one path from any vertex to any other.
- The maximum connected subgraphs of an undirected graph are called **connected components**.



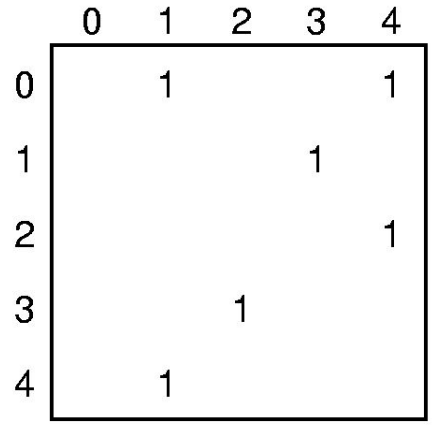
Overview

- Basic terms and definitions of graphs
- **How to represent graphs**
- Graph traversal methods

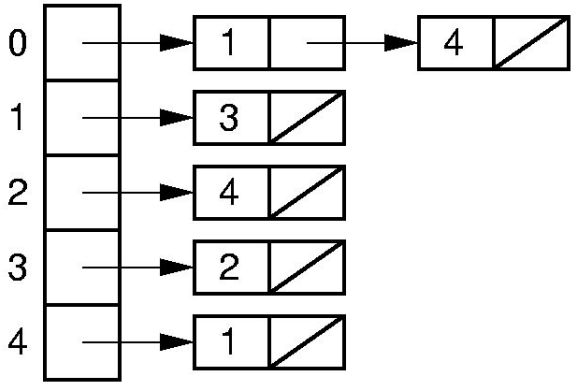
Directed Representation



Graph

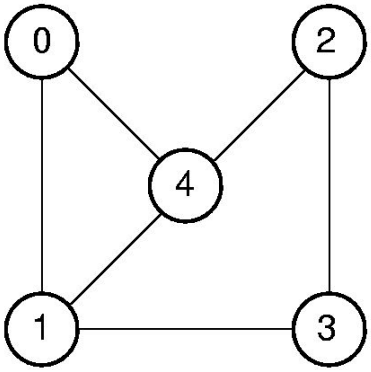


Adjacency Matrix



Adjacency List

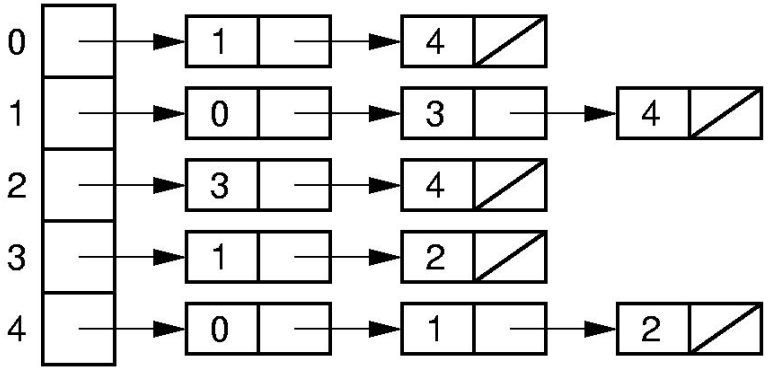
Undirected Representation



Graph

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

Adjacency Matrix



Adjacency List

Representation Costs

- Adjacency Matrix:
  - Space cost?
- Adjacency List:
  - Space cost?
  - The maximum size of |E|?
- When is Adjacency List more space efficient than Adjacency Matrix and vice versa?

Representation Costs

- Adjacency Matrix:
  - Space cost? $\Theta(|V|^2)$
- Adjacency List:
  - Space cost? $\Theta(|V| + |E|)$
  - The maximum size of |E|? $|V|^2$
- When is Adjacency List more space efficient than Adjacency Matrix and vice versa?
  - For **sparse graphs ($|E| << |V|^2$)**, and for **dense graph ($|E| \sim |V|^2$)**, respectively.

Overview

- Basic terms and definitions of graphs
- How to represent graphs
- **Graph traversal methods**

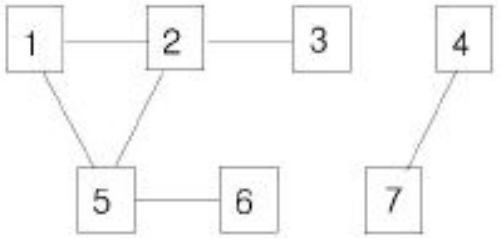
Graph Traversals

- Some applications require **visiting every vertex in the graph exactly once**.
- The application may require that **vertices be visited in some special order** based on **graph topology**.
- Examples: artificial intelligence search, shortest paths problems, connected components, …
- Important Traversals
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

Example: 바이러스

신종 바이러스인 웜 바이러스는 네트워크를 통해 전파된다. 한 컴퓨터가 웜 바이러스에 걸리면 그 컴퓨터와 네트워크 상에서 연결되어 있는 모든 컴퓨터는 웜 바이러스에 걸리게 된다.

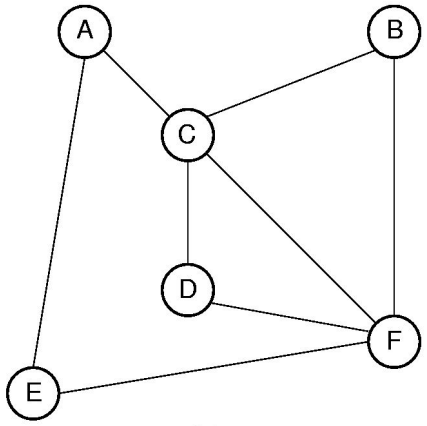
어느 날 1번 컴퓨터가 웜 바이러스에 걸렸다. 컴퓨터의 수와 네트워크 상에서 서로 연결되어 있는 정보가 주어질 때, 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 출력하는 프로그램을 작성하시오.



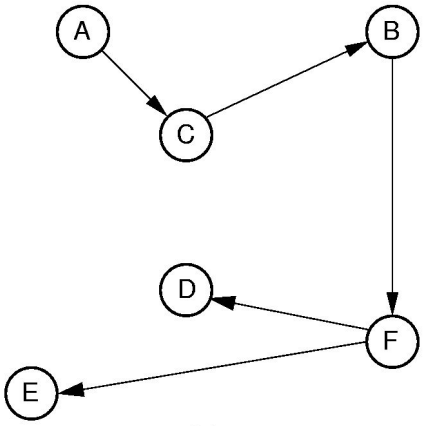
https://www.acmicpc.net/problem/1697

Depth First Search

- Main Idea
  - Start from a vertex s
  - Visit an unvisited neighbor v of s
  - Visit an unvisited neighbor v' of v
  - … continue until all vertices are visited



(a)



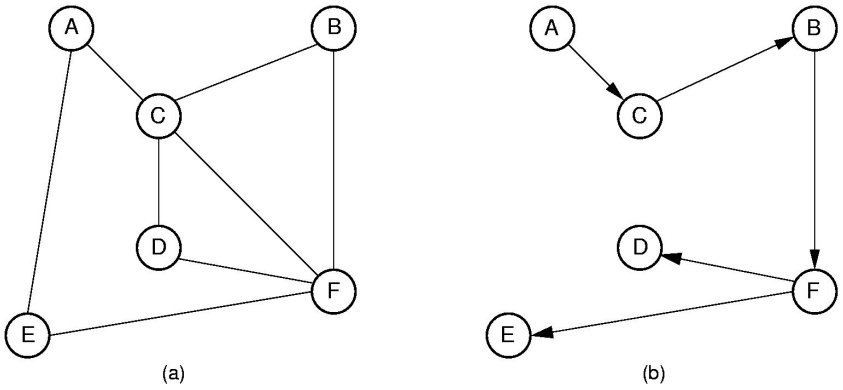
(b)

Depth First Search

- Pseudocode

```
function DFS(v, G):
    # write here something to do before visiting v
    mark v is 'visited'
    for w in G.neighbors():
        if w is not 'visited':
            DFS(w, G)
```

- Cost: $\Theta(|V| + |E|)$

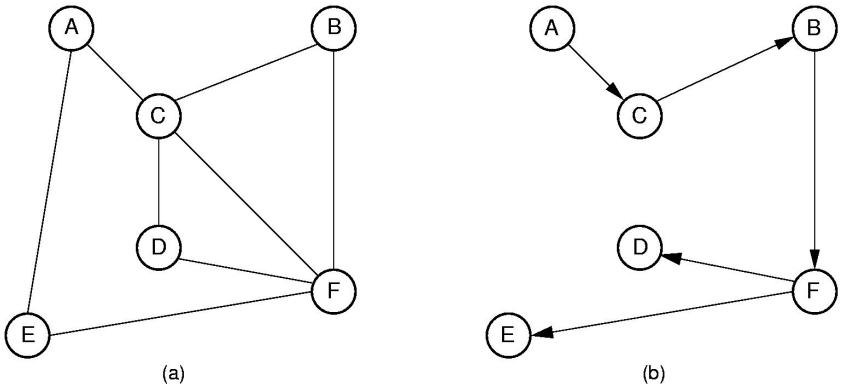


Depth First Search

- Pseudocode (using recursive function)

```
function DFS(v, G):
    # write here something to do before visiting v
    mark v is 'visited'
    for w in G.neighbors():
        if w is not 'visited':
            DFS(w, G)
```

- Cost: $\Theta(|V| + |E|)$



Depth First Search

- Pseudocode (using loop)

```
function DFS(u, G):
    initialize a stack S
    S.push(u)
    while S is not empty:
        v = S.pop()
        # do something
        mark v is 'visited'

        for w in G.neighbors():
            if w is not 'visited':
                S.push( w)
```

Breadth First Search

- Breadth First Search (BFS)
  - Like DFS, but replace stack with a queue.
  - Visit vertex's neighbors before continuing deeper in the tree.

- BFS Algorithm
  - Start from a vertex s
  - Visit all neighbors of s
  - Visit all neighbors of neighbors of s
  - … continue until all vertices are visited

Breadth First Search

- Breadth First Search (BFS)
  - Like DFS, but replace stack with a queue.
  - Visit vertex's neighbors before continuing deeper in the tree.

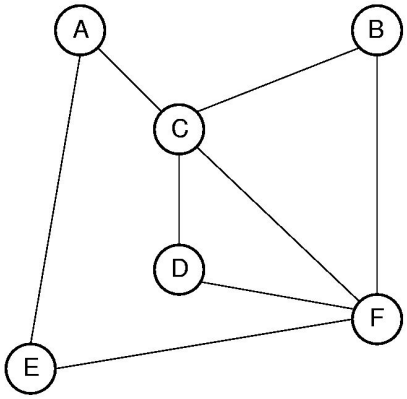
- BFS Algorithm
  - Start from a vertex s
  - Visit all neighbors of s
  - Visit all neighbors of neighbors of s
  - … continue until all vertices are visited

Breadth First Search

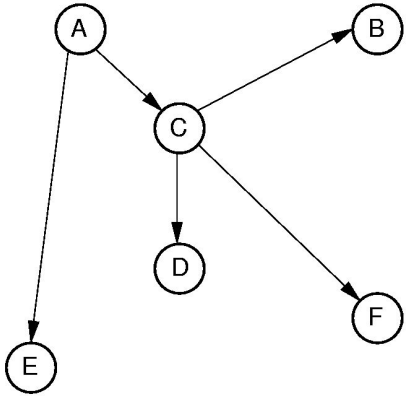
- Pseudocode (using loop)

```
function BFS(u, G):
    initialize a queue Q
    Q.enqueue(u)
    while Q is not empty:
        v = S.dequeue()
        # do something
        mark v is 'visited'

        for w in G.neighbors():
            if w is not 'visited':
                S.enqueue(w)
```



(a)



(b)

Breadth First Search

- Pseudocode (using loop)

```
function BFS(u, G):
    initialize a queue Q
    Q.enqueue(u)
    while Q is not empty:
        v = S.dequeue()
        # do something
        mark v is 'visited'

        for w in G.neighbors():
            if w is not 'visited':
                S.enqueue(w)
```

```
function DFS(u, G):
    initialize a stack S
    S.push(u)
    while S is not empty:
        v = S.pop()
        # do something
        mark v is 'visited'

        for w in G.neighbors():
            if w is not 'visited':
                S.push(w)
```

Questions?