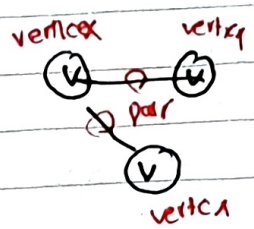
Wk 7 Graphs

graph G is a pair (V, E) where

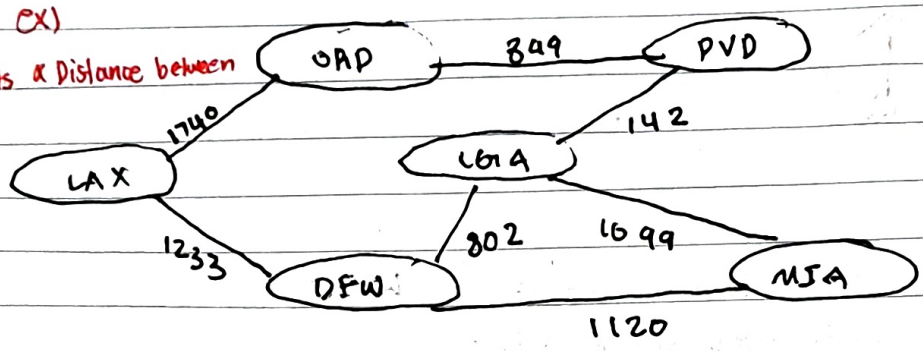
V: set of nodes called vertices

E: collection of pairs of vertices



ex)

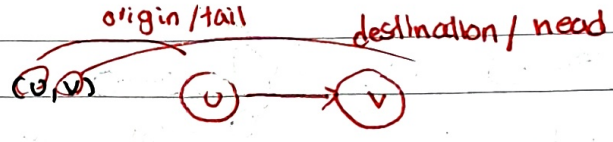
Airports & Distance between



Edge Types

- Directed Edge
  - ordered pair of vertices (u, v)

    eg. a flight



- Undirected Edge
  - unordered pair of vertices (u, v)

    eg. Two way road

some Applications

- electronic circuits (circuit board)
- transportation networks (train, flight)
- Computer networks (internet, web)
- Modelling (CPA - diagram, Gantt precedence constraints)

Path

- sequence of vertices such that every consecutive vertices is connected by edge
- Simple path: path consisted of unique vertices (no repeats)
- cycle: starts and ends at same vertex
  - simple cycle: vertices are unique

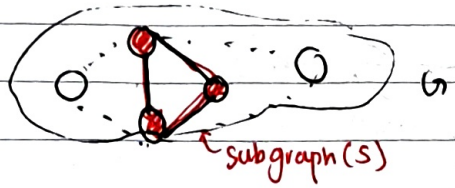
Acyclic graph: No cycles

Subgraphs

$G = (V, E)$

$S = (U, F)$

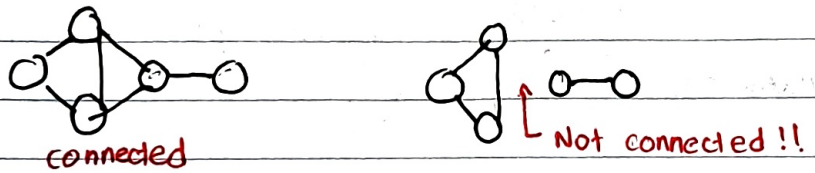
if $U \subseteq V$ and $F \subseteq E$, S is a subgraph of G



Connectivity

A graph $G = (V, E)$ is connected if

there is path between every pair of vertices



Trees & Forests

Tree T is

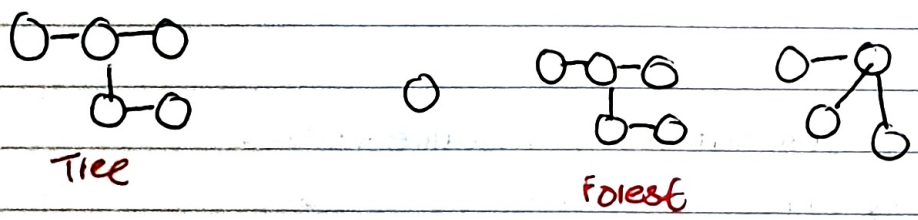
- connected
- has no cycles

∴ For T with n nodes, there are n-1 edges.

Forest is a graph without cycles.

a.k.a. bunch of trees

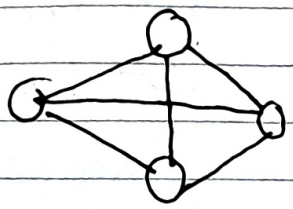
its unconnected components



Degree

Degree of a vertex: # of edges that are incident to that vertex

Degree of graph: maximum degree of any node



n (number of vertices) : 4

m (number of edges) : 6

Δ (maximum degree) : 3

In a simple undirected graph...

$\sum_{v \text{ in } V} deg(v) = 2m$ edge 개수 x2 (생각해)

$m \leq (n)(n-1)/2$

$= (n-1) + (n-2) + \cdots + 1$

Graph ADT

Vertex: stores an associated object (eg. airport)

Edge: stores an associated object (eg. flight code, distance)

] retrieved by getElement()

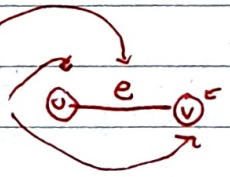
vertices(): all vertices in a graph , numVertices()

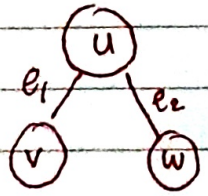
edges() : all edges in a graph , numEdges()

getEdge(u, v) : returns edge from u → v (if exists)

endVertices(e) : returns endpoint vertices of edge e

opposite(u, e) : returns the other vertex connected by e.





vertices() : [u, v, w]

edges() : [e1, e2]

getEdge(u, w) = e2 , getEdge(v, w) = null

endVertices(e2) = [u, w]

opposite(v, e1) = u , opposite(v, e2) = null

outDegree(v): # of outgoing edges

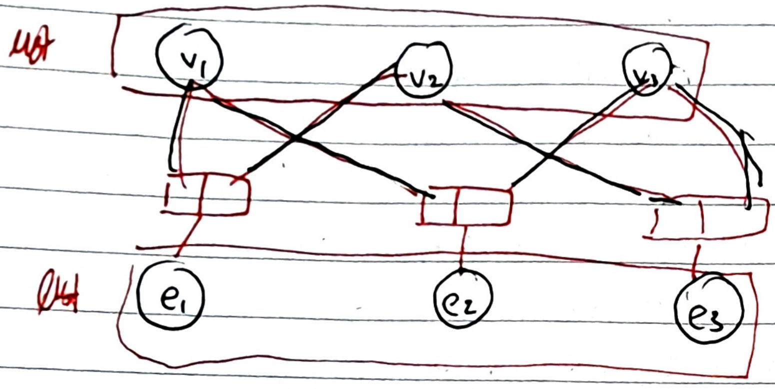
inDegree(v): # of ingoing edges

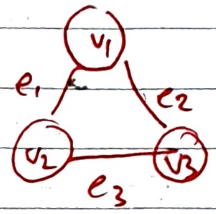
outgoing : 2

ingoing : 1

Insert, Remove Edges

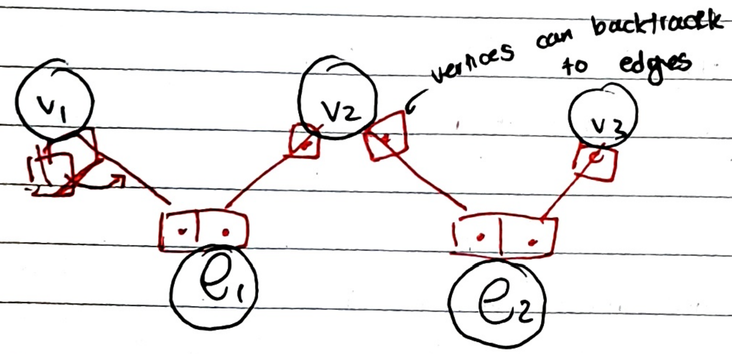
Implementation : Edge List Structure

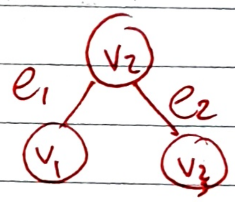




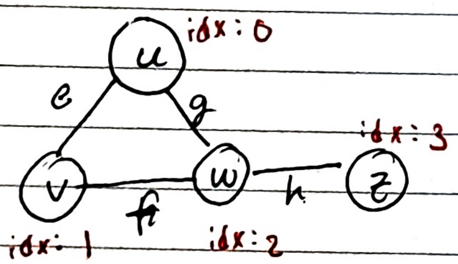
Adjacency List (Good choice!)

each vertex also keeps a sequence of edges incident on it





Adjacency Matrix



		0	1	2	3
u → 0	0		e	g	
v → 1	1	e		f	
w → 2	2	g	f		h
z → 3	3			h	

edge 주라 더 적은 애만 보기로 됨(undirected)

* undirected

	Edge List	Adjacency List	Adjacency Matrix
space	O(n + m)	O(n+m)	O(n^2)
incidentEdges(v)	O(m)	O(deg(v))	O(n)
getEdge(u,v)	O(m)	O(min(deg(u), deg(v)))	O(1)
insertVertex(v)	O(1)	O(1)	O(n^2) has to augment the matrix
insertEdge(u,v,e)	O(1)	O(1)	O(1)
removeVertex(v)	O(m)	O(deg(v))	O(n^2)
removeEdge(e)	O(1)	O(1)	O(1)

19-2: Graph Traversals

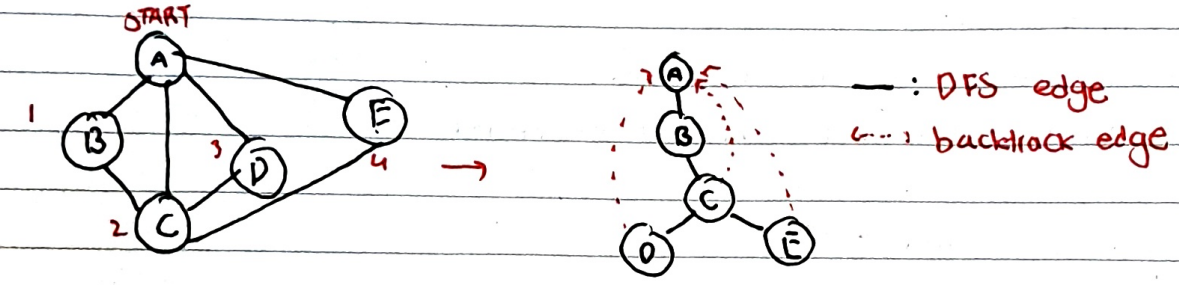
Traversal: systematic procedure for exploring a graph by examining all of its vertices & edges.

* We want efficient approach O(n)*

Depth-first Search

Start from one node

→ follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if 'stuck'.




```
def DFS(G):
    for u in G.vertices() do                    O(n)
        visited[G] ← False
        parent[G] ← None                        } hash tables

def DFS_visit(u):
    visited[u] ← True
    for v in G.incident(u) do                   O(deg(u))
        if not visited[v] then                  O(1)
            parent[v] ← u
            DFS_visit(v)

for u in G.vertices() do                        O(n)
    if not visited[G] do ←  (not connected parts)
        DFS_visit(u)

return parent   - defines path took by DFS
```

forms a spanning tree of u

forms collection of spanning trees of unvisited u.

forest

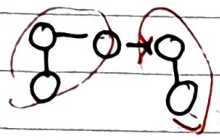
DFS Applications

- find path between 2 given vertices
- find a cycle in the graph
- Test whether graph is connected
- Compute connected components of a graph

Identifying cut edges

In a connected graph $G = (V, E)$
if $G = (V, E \setminus \{(u,v)\})$ is not connected

edge connecting u, v in E



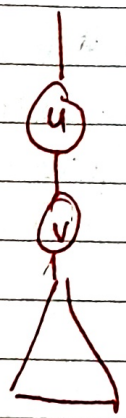
Algorithm 1 $O(m^2)$ ← trivial

1. try removing one edge at a time.
2. DFS to test if G is connected/ disconnected.

Algorithm 2 $O(nm)$ ← better

- disregard ~~path not~~ edges not taken by the path of DFS

Algorithm 3 $O(n+m)$ ← even better !!



when $u = parent(v)$

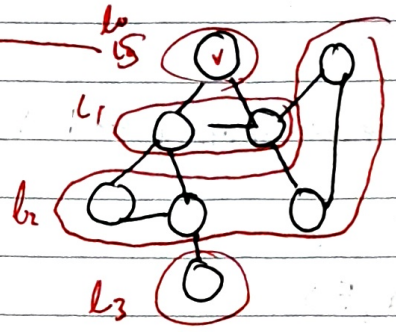
if $down\_up(v) \leq level(u)$
remain connected

else:
edge v-u is cut edge

Breadth-first search (BFS)

tries to visit all vertices at distance k from a start vertex s before visiting vertices at distance k+1.

layer 1 : {s}
2 : {s} ... 3
3 : {} ... 3




```
def BFS(G, s):          # L0
    for u in G.vertices() do
        seen[u] ← false
        parent[u] ← None

    seen[s] ← True
    layer ← [ ]
    current ← [s]       # L0
    next ← [ ]

    while not current.is_empty() do
        layers.append(current)
        for u in current do
            for v in G.incident(u) do
                if not seen[v] then
                    next.append(v)
                    seen[v] ← True
                    parent[v] ← u
        current ← next
        next ← None [ ]
```

if seen, cross-edge
otherwise, BFS-edge

Applications

(same as DFS) • find a cycle, test if graph is connected

(unique) • Find shortest path between two vertices
- Compute spanning tree) wk 8
- bipartite test (last)