Nonlinear Data Structure

Unit#3

Marwadi University

Department of Computer Engineering

Data Structure
01CE0301 / 3130702

Ravikumar Natarajan

Highlights

- Trees
- Graphs

Till Now

- **Trees**
- Graphs

- Tree definitions and their concepts
- Representation of binary tree
- Binary tree traversal
- Binary search trees
- General trees vs binary trees
- Threaded binary tree
- Applications of Trees
- Balanced tree and its mechanism
- Height and Weight Balanced Trees

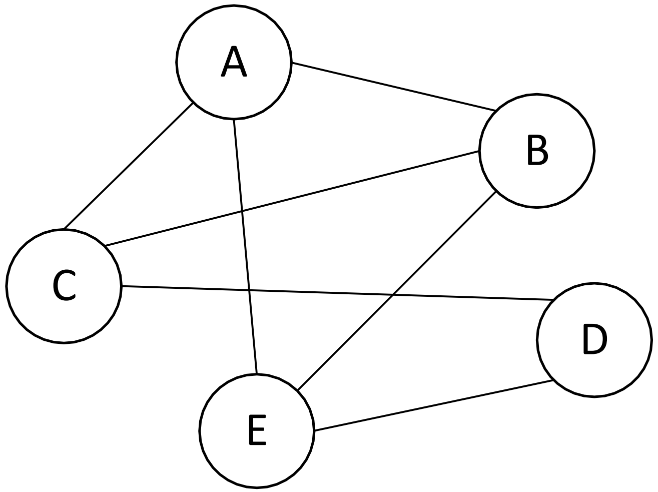
Up Next

- Trees
- **Graphs**

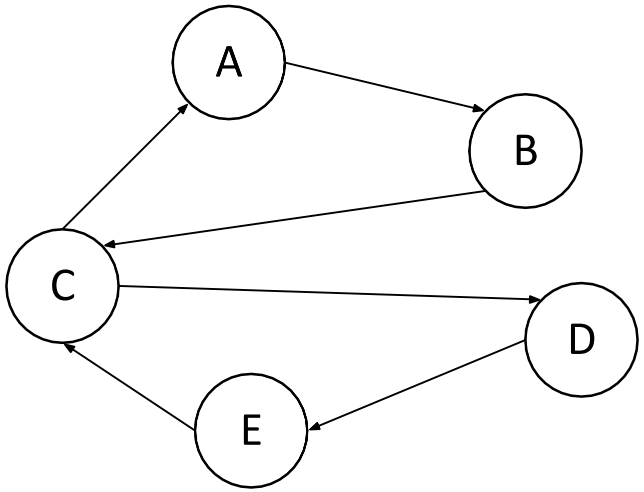
- Graphs and their understanding
- Matrix representations of a given graph
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Minimum Spanning Trees Algorithms (Prims, Kruskal, Dijkstra)
- Path Matrix
- Warshall's Algorithm

Introduction

- **Graph G = (V, E)**
  - V = set of vertices
  - E = set of edges ⊆ (V ×V)



Undirected Graph



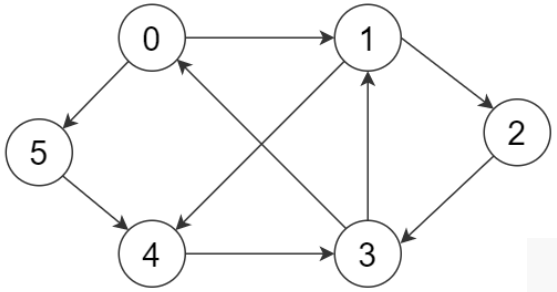
Directed Graph

Introduction

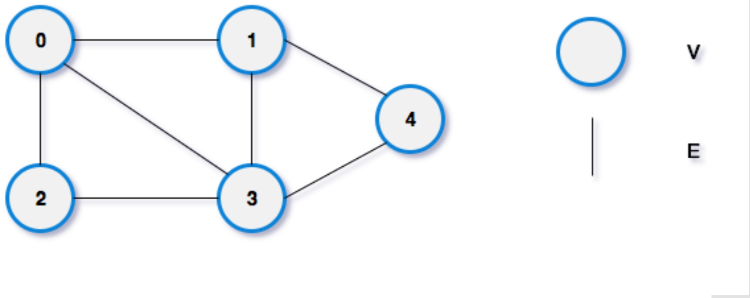
- **Types of graphs**
  - **Undirected**: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - **Directed**: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - **Weighted**: each edge has an associated weight, given by a weight function $w : E \rightarrow R$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
  - **Complete**: Undirected. Every vertex has an edge to all other vertex. Complete graph with N vertices has $N(N+1)/2$ *edges.*

Types of graphs

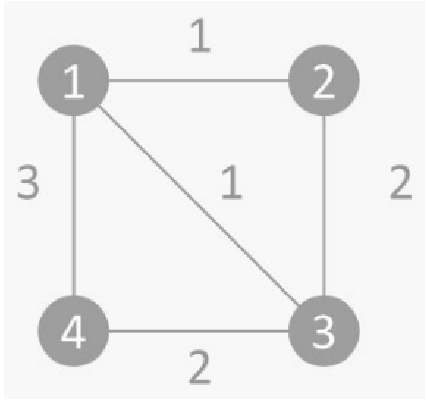
Directed Graph

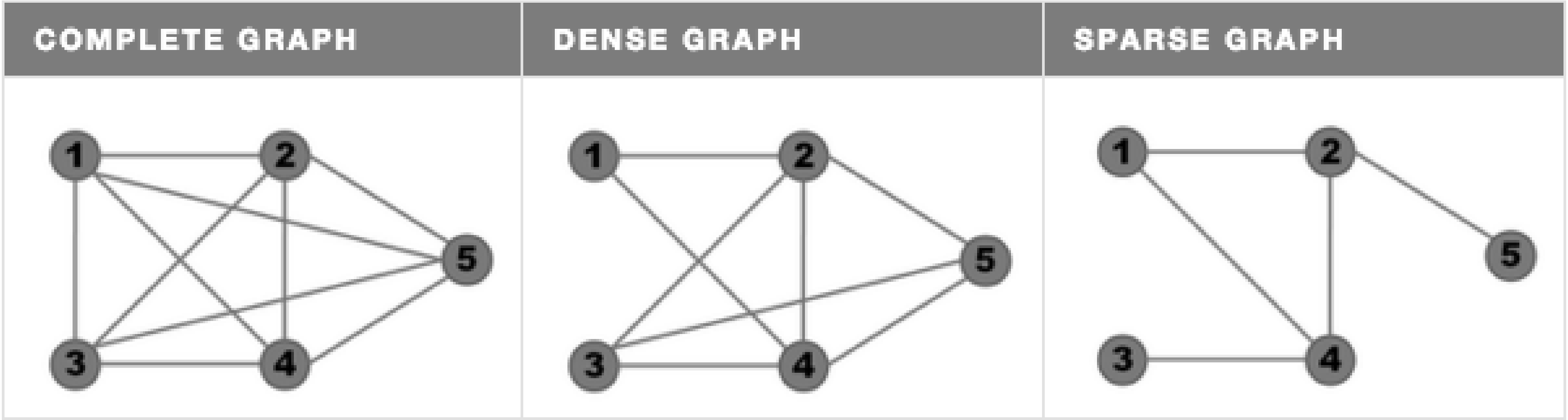


Undirected Graph



Weighted Graph





Introduction

- **Adjacent Nodes**: Two vertices are adjacent if they are endpoints of the same edge.
- An edge is **incident** on a vertex if the vertex is an endpoint of the edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin.
- **Incoming edges** of a vertex are directed edges that the vertex is the destination.
- **Degree of a vertex**, v, denoted deg(v) is the number of incident edges.
- **Out-degree**, outdeg(v), is the number of outgoing edges.
- **In-degree**, indeg(v), is the number of incoming edges.

Introduction

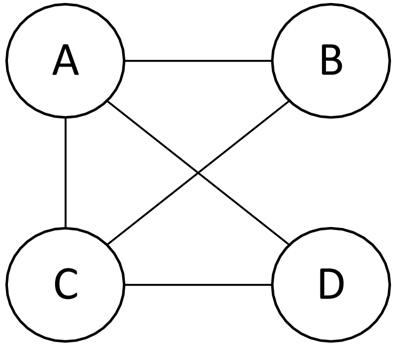
- A **path** is a sequence of vertices such that each vertex is adjacent to the next. In a path, each edge can be traveled only once.
- The **length of a path** is the number of edges in that path.
- **Cycle(loop)** is a path that starts and end at the same vertex.
- In a **weighted graph**, every edge is assigned some weight or length(positive value).

Introduction

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so, if G is directed.
- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.

Representation of Graphs

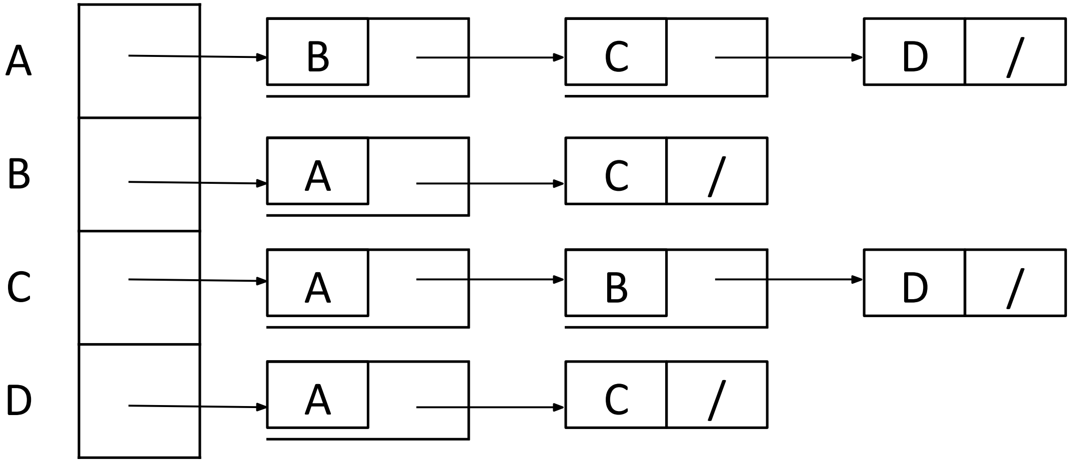
- Two standard ways.



Adjacency Matrix

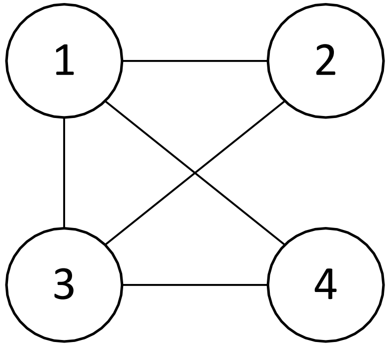
	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

Adjacency Lists

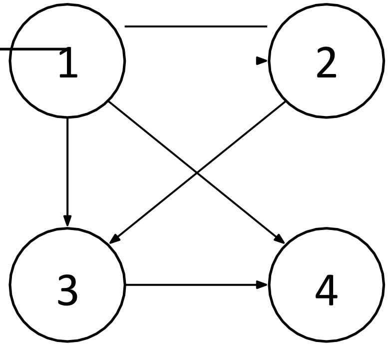


Adjacency Matrix

- |V| × |V| matrix A.
- Number vertices from 1 to |V| in some arbitrary manner.
- Used for dense graph. Space complexity θ (n^2)
- A is then given by: $A[i, j] = a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



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



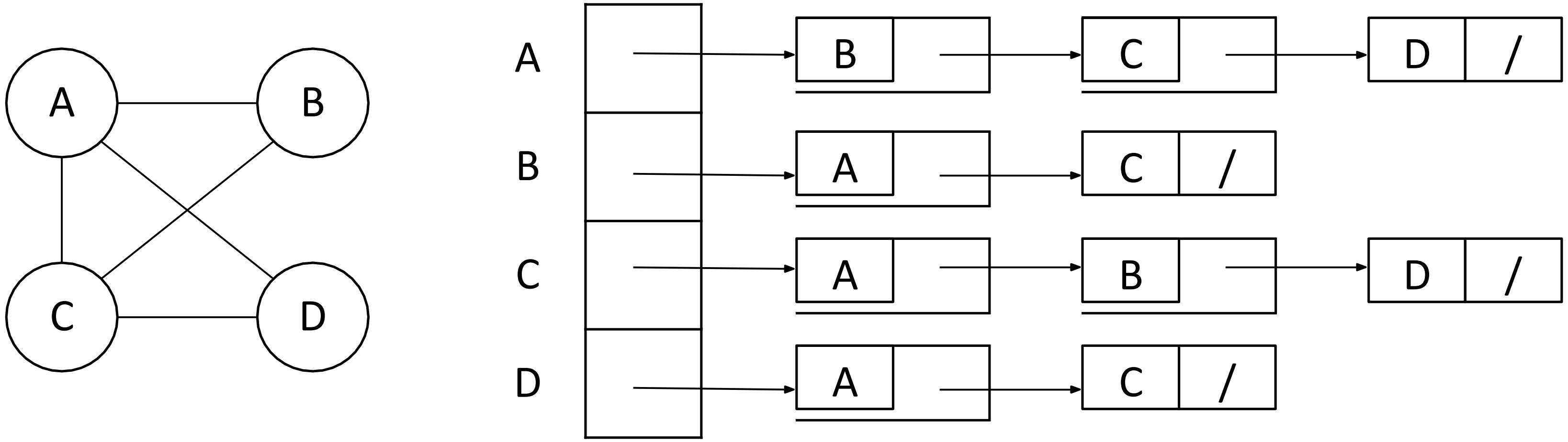
	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0

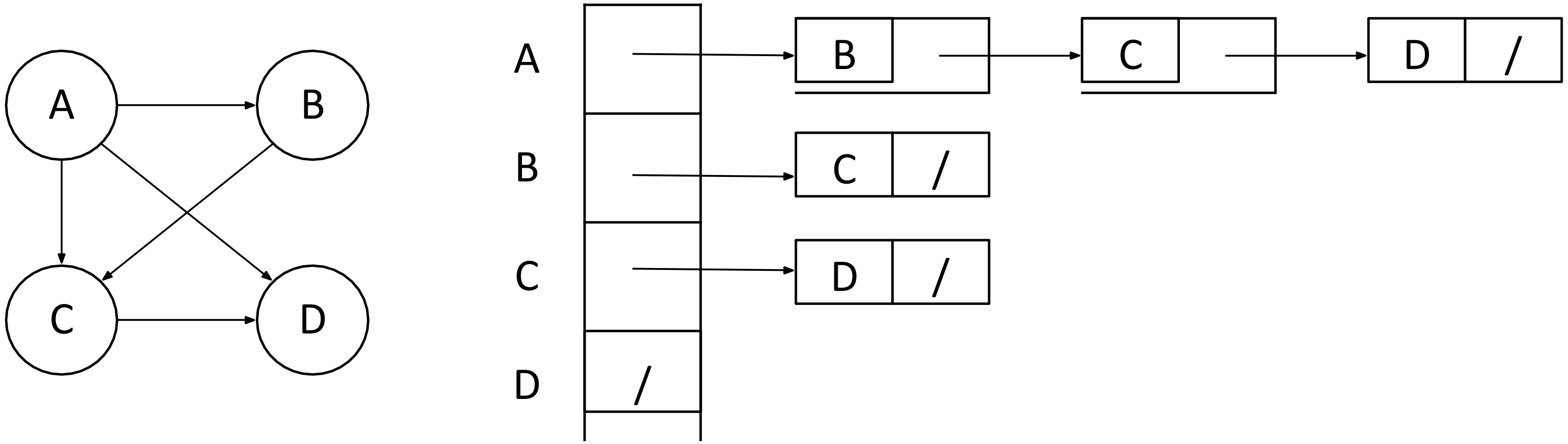
- A = AT for undirected graphs.

Adjacency Lists

- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, *Adj[u]* consists of all vertices adjacent to *u.*
- If weighted, store weights also in adjacency lists.
- Used for sparse graph
- For undirected the space complexity is θ $(n+2^{e})$
- For directed the space complexity is θ $(n+e)$

Adjacency Lists





Graph Traversal

- Graph traversal means **visiting every vertex and edge exactly once** in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.
- The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.
- During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Traversing / Searching Graphs

- Systematically follow the edges of a graph to visit the vertices of the graph.
  - Used to discover the structure of a graph.
- Standard tree-traversal algorithms.
  - Pre-order [root-left-right]
  - In-order [left-root-right]
  - Post-order [left-right-root]
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

Breadth-first Search

- A standard BFS implementation puts each vertex of the graph into one of two categories:
- Visited
- Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. BFS follows graph traversal technique.
- The algorithm works as follows:
  1. Select any vertex as starting vertex and insert that into the Queue and mark the status as visited.
  2. While Queue is non empty
     a. DELETE(dequeue) a vertex from the Queue and DISPLAY it.
     b. INSERT(enqueue) all the unvisited adjacent vertices into the Queue and mark the status as visited.
  3. Repeat step 2 until the Queue gets empty.

Breadth-first Search

- ## Algorithm

```
/* Array visited[] is initialized to O */
/* BFS traversal on the graph G is carried out beginning at vertex V */
Void BFS(int V)
{
    q: a queue type variable;
    initialize q;
    visited[v] = 1; //mark v as visited
    add the vertex V to queue q;
    while(q is not empty)
    {
    v <- delete an element from the queue;
    for all vertices w adjacent from V
    {
        if(!visited[w])
        {
            visited[w]=1;
            add the vertex w to queue q;
        }
    }
}
```

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "**discovered**" the first time it is encountered during the search.
  - A vertex is "**finished**" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.

Breadth-first Search

- Algorithm:
  - Graph $G = (V, E)$, either directed or undirected, and source vertex $s \in V$.
- Output:
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$, such that (u, v) is last edge on shortest path s to v. [u is v's predecessor.]
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-first Search

- Input:
  - Graph $G = (V, E)$, either directed or undirected, and source vertex $s \in V$.
- Output:
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$, such that (u, v) is last edge on shortest path s to v. [u is v's predecessor.]
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-first Search

BFS(G, s)

1. **for** each vertex u in V[G] – {s} **do**
2. color[u] $\leftarrow$ white
3. d[u] $\leftarrow \infty$
4. π[u] $\leftarrow$ nil
5. color[s] $\leftarrow$ gray
6. d[s] $\leftarrow$ 0
7. π[s] $\leftarrow$ nil
8. Q $\leftarrow \Phi$
9. enqueue(Q, s)

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

Breadth-first Search

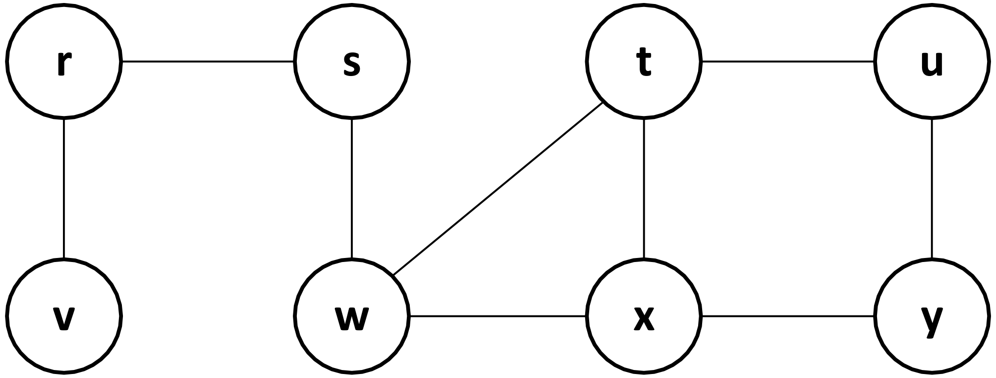
```
while Q ≠ Φ do
    u ← dequeue(Q)
    for each v in Adj[u] do
        if color[v] = white 14.
          then  color[v] ← gray
                d[v] ← d[u] + 1
                π[v] ← u
                enqueue(Q, v)
    color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

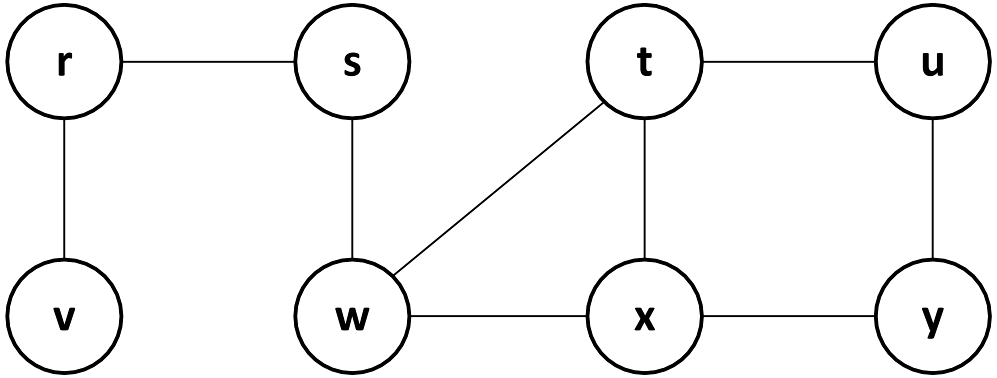
Breadth-first Search

- Given Graph:



- Source *s*

Breadth-first Search



V[G]	d[v]	π[v]
r		
s		
t		
u		
v		
w		
x		
y		

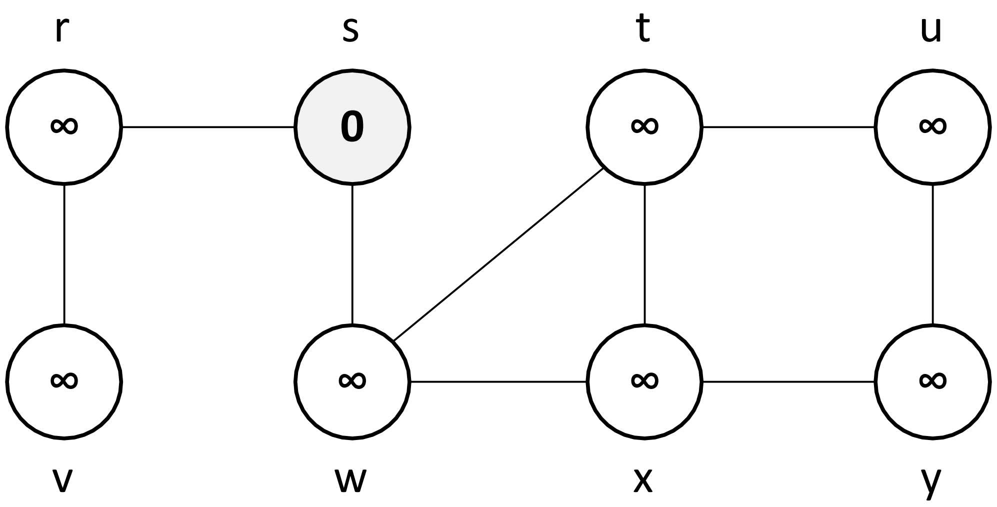
Q:

u:

v:

white: undiscovered
gray: discovered
black: finished
Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

Breadth-first Search



Q: s

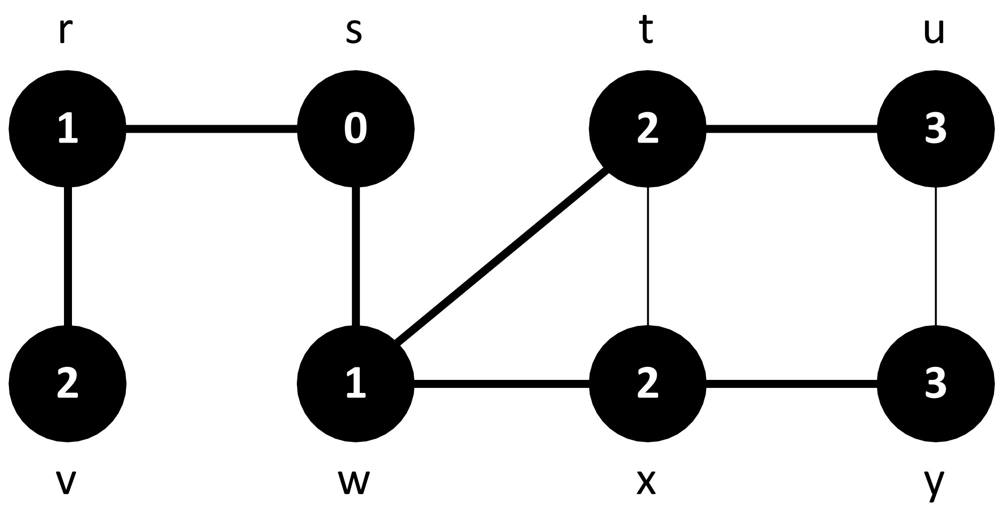
u:

v:

V[G]	d[v]	π[v]
r	∞	-
s	0	-
t	∞	-
u	∞	-
v	∞	-
w	∞	-
x	∞	-
y	∞	-

white: undiscovered
gray: discovered
black: finished
Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

Breadth-first Search



Q:

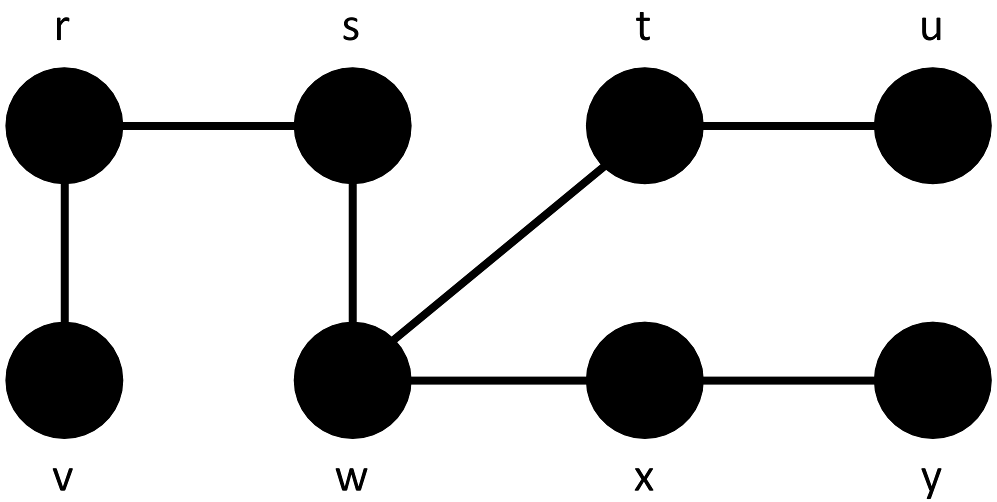
u:

v:

V[G]	d[v]	π[v]
r	1	s
s	0	-
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x

white: undiscovered
gray: discovered
black: finished
Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

Breadth-first Search

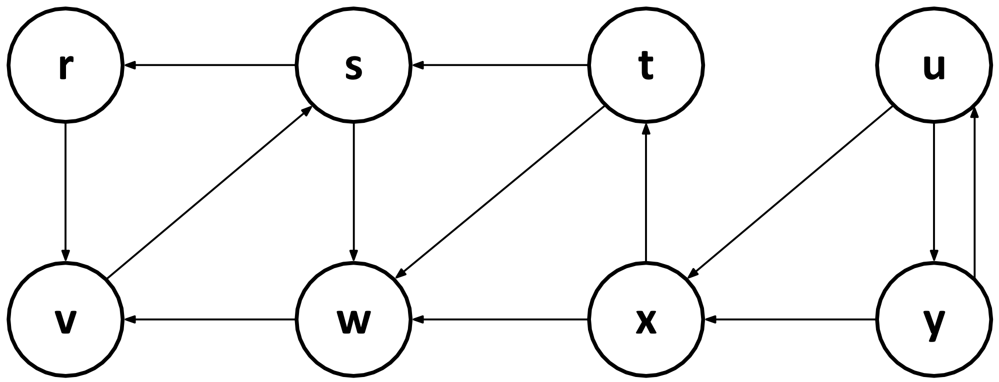


Breadth First Tree

V[G]	d[v]	π[v]
r	1	s
s	0	-
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x

Example

- Given Graph:



- Source *u*

Breadth-first Search Time Analysis

```
BFS(G, s)
for each vertex u in V[G] – {s} do
        color[u] ← white
        d[u] ← ∞
        π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q, s)
while Q ≠ Φ do
        u ← dequeue(Q)
        for each v in Adj[u] do
                if color[v] = white
                then color[v] ← gray
                        d[v] ← d[u] + 1
                        π[v] ← u
                        enqueue(Q, v)
        color[u] ← black
```

$O(V)$

$O(V)$

$\Theta(E)$

$O(V+E)$

Complexity of Breadth-first Search

- The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

- The space complexity of the algorithm is O(V).

Applications of Breadth-first Search

- **Web crawlers** - To build index by search index
- **For GPS navigation** - To find neighboring location from source location.
- **Network broadcasting** – packets to find and reach all node addresses.
- **Un-weighted graphs -** easily create the shortest path and a minimum spanning tree
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph

Depth-first Search

- Explore edges going out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its predecessor).
- **"Search as deep as possible first."**
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- A standard DFS implementation puts each vertex of the graph into one of two categories:
- Visited
- Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The DFS algorithm works as follows:

1. Select any vertex as starting vertex and insert it in the top of stack and mark the status as visited.
2. While stack is not empty
   a. Select any one unvisited adjacent vertex of the corresponding vertex and **insert (push)** into stack and mark status as visited.
   b. If there is no unvisited adjacent vertex then perform backtracking and perform **delete (pop)** element from stack.
3. Repeat step 2 until the stack gets empty.

Depth-first Search

- Algorithm

```
N <- number of nodes
Initialize visited[] to false (0)
for(i=0; i<n; i++)
    Visited[i]=0;
Void DFS(vertex i) //DFS starting from i
{
    Visited[i] =1;
    For each w adjacent to I
        If(!visited[w])
            DFS(w)
}
```

Depth-first Search

- Input:
  - G = (V, E), directed or undirected. No source vertex given!
- Output:
  - 2 timestamps on each vertex. Integers between 1 and 2|V|.
    - d[v] = discovery time (v turns from white to gray)
    - f [v] = finishing time (v turns from gray to black)
  - π[v] : predecessor of v = u, such that v was discovered during the scan of u's adjacency list.
- Uses the same colouring scheme for vertices as BFS.

Depth-first Search

DFS(G)

1. for each vertex u ∈ V[G] **do**

2. color[u] ← white

3. π[u] ← NIL

4. time ← 0

5. for each vertex u ∈ V[G] **do**

6. if color[u] = white

7. then DFS-Visit(u)

white: undiscovered
gray: discovered
black: finished

color[v]: color of v
π[u]: predecessor of v
d[v]: discovery time of v
f[v]: finishing time of v
Uses a global timestamp time.

Depth-first Search

DFS-Visit(u)

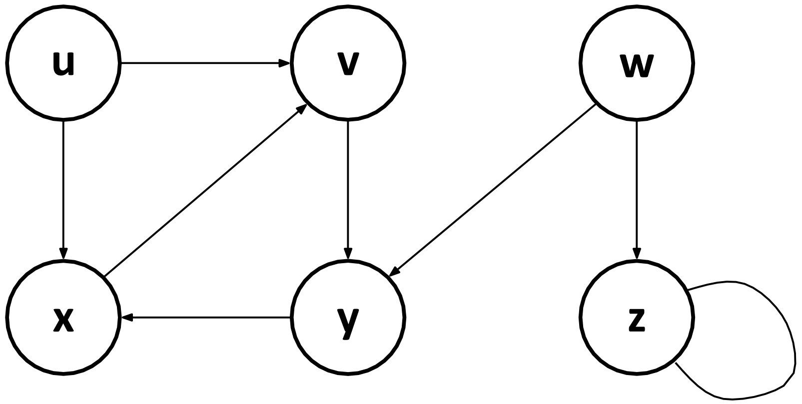
1. color[u] ← GRAY
2. time ← time + 1
3. d[u] ← time
4. **for each** v ∈ Adj[u] **do**
5. **if** color[v] = WHITE
6. **then** π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK
9. f[u] ← time ← time + 1

white: undiscovered
gray: discovered
black: finished

color[v]: color of v
π[u]: predecessor of v
d[v]: discovery time of v
f[v]: finishing time of v
Uses a global timestamp time.

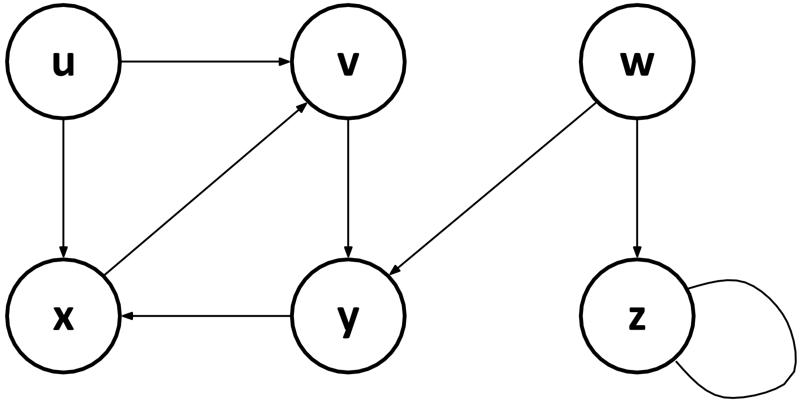
Depth-first Search

- Given Graph:



- Source: Any

Depth-first Search



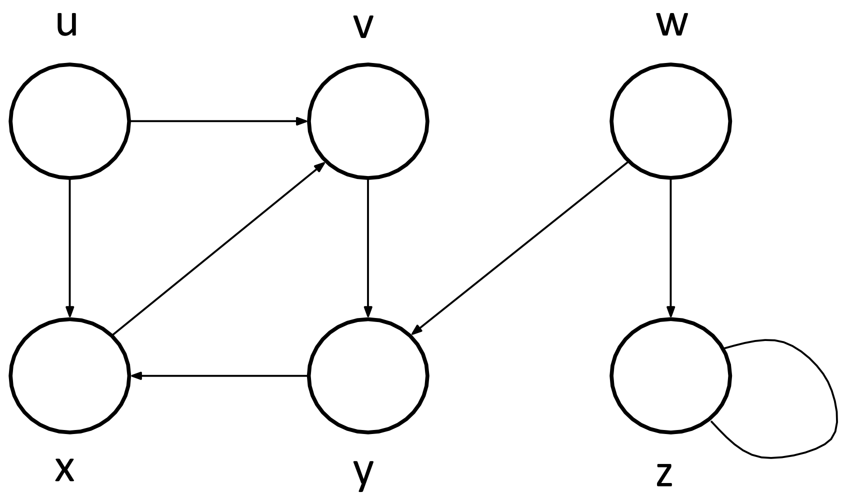
V[G]	π[v]	d[v]	f[v]
u			
v			
w			
x			
y			
z			

white: undiscovered
gray: discovered
black: finished

Time:

color[v]: color of v
π[u]: predecessor of v
d[v]: discovery time of v
f[v]: finishing time of v
Uses a global timestamp time.

Depth-first Search



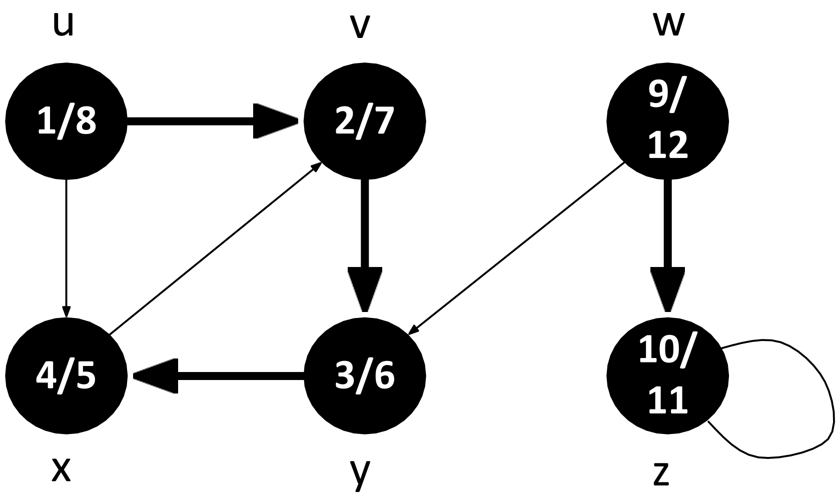
V[G]	π[v]	d[v]	f[v]
u	-		
v	-		
w	-		
x	-		
y	-		
z	-		

white: undiscovered
gray: discovered
black: finished

Time: 0

color[v]: color of v
π[u]: predecessor of v
d[v]: discovery time of v
f[v]: finishing time of v
Uses a global timestamp time.

Depth-first Search



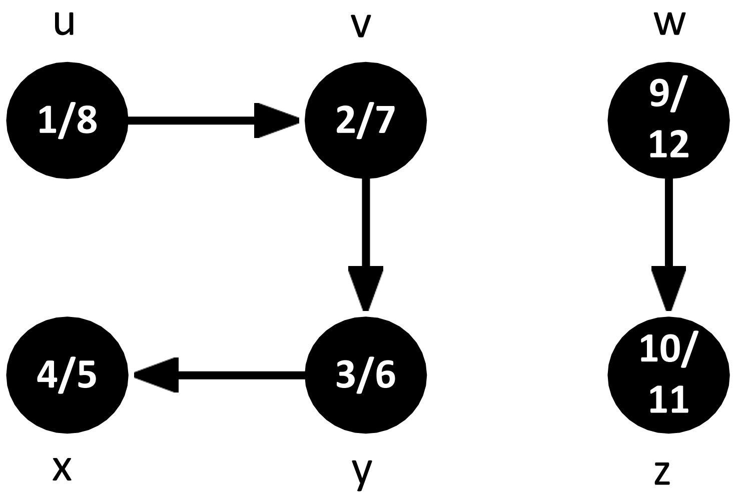
Time: 12

V[G]	π[v]	d[v]	f[v]
u	-	1	8
v	u	2	7
w	-	9	12
x	y	4	5
y	v	3	6
z	w	10	11

white: undiscovered
gray: discovered
black: finished

color[v]: color of v
π[u]: predecessor of v
d[v]: discovery time of v
f[v]: finishing time of v
Uses a global timestamp time.

Depth-first Search

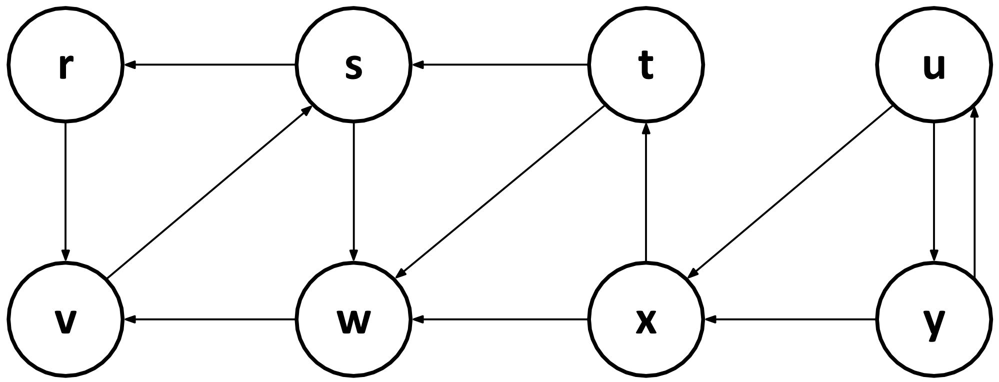


Depth-first Forest

V[G]	π[v]	d[v]	f[v]
u	-	1	8
v	u	2	7
w	-	9	12
x	y	4	5
y	v	3	6
z	w	10	11

Example

- Given Graph:



DFS(G)

1. **for each** vertex u ∈ V[G] **do**
2. color[u] ← white
3. π[u] ← NIL
4. time ← 0
5. **for each** vertex u ∈ V[G] **do**
6. **if** color[u] = white
7. **then** DFS-Visit(u)

$\Theta(v)$

DFS-Visit(u)

1. color[u] ← GRAY
2. time ← time + 1
3. d[u] ← time
4. **for each** v ∈ Adj[u] **do**
5. **if** color[v] = WHITE
6. **then** π[v] ← u
7. DFS-Visit(v)
8. color[u] ← BLACK
9. f[u] ← time ← time + 1

$\Theta(E)$

O(V+E)

Complexity of Depth-first Search

- The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is O(V).

Applications of Depth-first Search

- **Weighted Graph** - graph traversal generates the shortest path tree and minimum spanning tree.
- **Detecting a Cycle in a Graph** - A graph has a cycle if we found a back edge during DFS.
- **Path Finding** - search a path between two vertices.
- **Topological Sorting** - It is primarily used for scheduling jobs from the given dependencies among the group of jobs.

Till Now

- Trees
- **Graphs**

- Graphs and their understanding
- Matrix representations of a given graph
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Minimum Spanning Trees Algorithms (Prims, Kruskal, Dijkstra)
- Path Matrix
- Warshall's Algorithm

Up Next

- Trees
- **Graphs**

- Graphs and their understanding
- Matrix representations of a given graph
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Minimum Spanning Trees Algorithms (Prims, Kruskal, Dijkstra)
- Path Matrix
- Warshall's Algorithm

Thank You.