Graphs

$G = (V, E)$

V is a set of vertices (often called nodes) — $|V|$ = number of vertices

E is a set of edges between vertices — $|E|$ = number of edges

Undirected Graph

Each edge specifies a connection between two vertices without any predecessor (source) / successor (sink) relationship

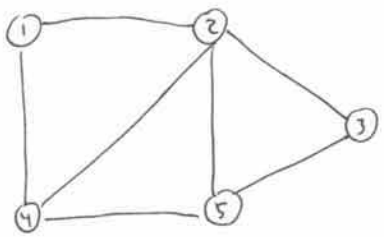
$edge(a, v) \equiv edge(v, a)$

Directed Graph

Each edge specifies a directed connection from one vertex (source) to another vertex (sink)

$edge(u, v) \not\equiv edge(v, u)$

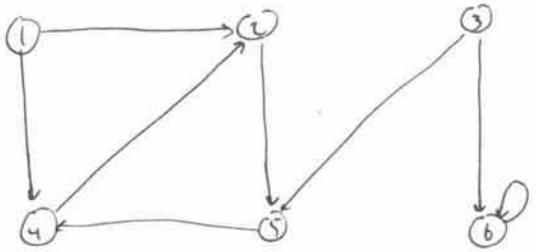
Example (undirected)



$V = \{1, 2, 3, 4, 5\}$

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

Example (directed)



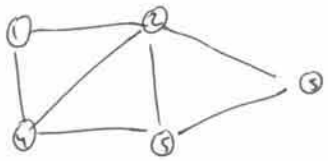
$V = \{1, 2, 3, 4, 5, 6\}$

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

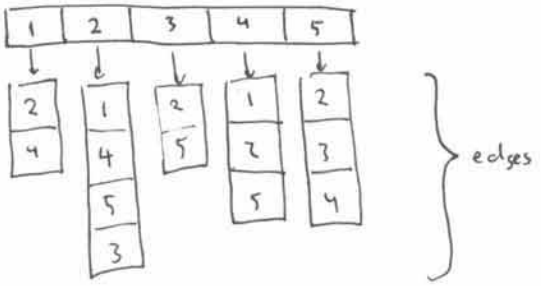
Graph Representation

Adjacency-list Representation: list (array) of vertices where each vertex contains a list (array) of vertices to which the vertex connects (i.e, edges)

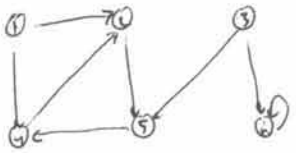
Example: undirected



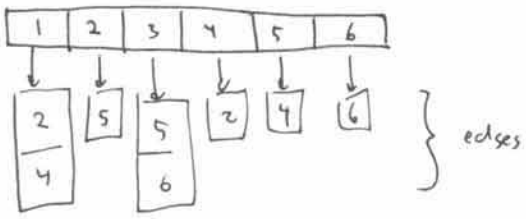
Vertices:



Example: directed



Vertices:



Implementation Note: Can be implemented as arrays of with pointers

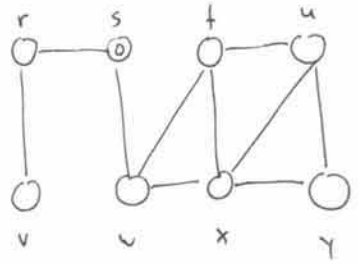
Algorithms:

Breadth-first search
- Given a graph $G = (V, E)$ and start vertex s
- Computes distant to all vertices $v \in V[G] - \{s\}$ from a vertex s

```
BFS(G, s):
    for each vertex u ∈ V[G] - {s}
        color[u] = white
        d[u] = ∞

    color[s] = gray
    d[s] = 0
    Q = ∅
    Q.push(s)
    while Q ≠ ∅
        u = Q.pop()
        for each v ∈ Adj[u]
            if color[v] == white
                color[v] = gray
                d[v] = d[u] + 1
                Q.push(v)
        color[u] = black
```

Note: Can also create a breadth-first tree

Example:



Depth-first search

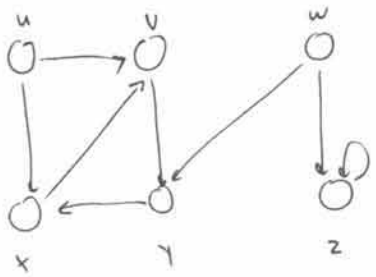
- Searches deeper in the graph before backtracking
- Given graph $G = (V, E)$ and start vertex s
- Compute discovery and finishing time for each vertex $v \in V[G]$.

```
DFS(G):
  for each vertex u ∈ V[G]
      color[u] = white

  time = 0
  for each vertex u ∈ V[G]
      if color[u] == white
          DFS-Visit(u)
```

```
DFS-Visit(u):
    color[u] = gray
    time = time + 1
    d[u] = time
    for each v ∈ Adj[u]
        if color[v] == white
            DFS-Visit[v]

    color[u] = black
    f[u] = time
    time = time + 1
```

Note: can also create a depth-first forest of depth-first trees

Example:



Direct Acyclic Graph: directed graph that contains no cycles

- often used to represent precedence relations (e.g. inputs and outputs of connected components such as logic circuits)

Topological Sort:

- Computes a linear ordering of all vertices in G such that if G contains an edge (u,v) then u appears before v in the ordering

Topological-Sort(G):

- Call DFS(G)
- as each vertex is finished, insert into front of list
- return list of vertices

Can also use stack:

```
Topological_Sort (G):
    DFS(G, stack)
    while stack ≠ ∅
        u = stack.pop
        output u
```

Modified DFS-visit

```
DFS_visit(u, stack)
    color[u] = gray
    time = time + 1
    d[u] = time
    for each vertex v ∈ Adj[u]
        if color[v] == white
            DFS_visit(v, stack)
    color = black
    f[u] = time
    stack.push(u)
    time = time + 1
```

Shortest Path

Many variate exist : single-destination, single-pair, all pairs

consider only single-source

path $p = \langle v_0, v_1, \dots v_k \rangle$

Single-source Shortest Path

- Given a graph $G = (V, E)$, compute the shortest path from source vertex $s \in V$ to each vertex $v \in V$
- Assumes weighted edges such that for each edge (u,v) a weight $w(u,v)$ is specified

$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

```
Initialize-Single-Source (G, s):
    for each vertex v ∈ V[G]
        d[v] = ∞
        π[v] = NIL
    d[s] = 0
```

$\pi[v]$ is the predecessor of node v

```
Relax (u, v, w):
    if d[v] > d[u] + w(u,v)
        d[v] = d[u] + w(u,v)
        π[v] = u
```

```
Dijkstra (G, w, s):
    Initialize-Single-Source(G, s)
    S = ∅
    Q = V[G]
    while Q ≠ ∅
        u = Q.extract_min
        S = S ∪ {u}
        for each vertex v ∈ Adj[u]
            Relax (u, v, w)
```

Q is a min-priority queue keyed by d[v]

Shortest paths for DAGs

```
DAG-Shortest-Path (G, w, s)

    Topological Sort Vertices (G)
    Initialize single source (G, s)
    for each vertex u (taken in topological sort order)
        for each vertex v ∈ Adj[u]
            Relax(u, v, w)
```

≠

```
while stack ≠ ∅
    u = stack.pop()
```