Graph Algorithms

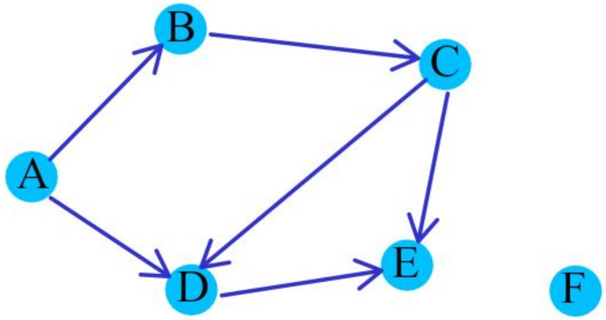
Graph

- A ***graph*** $G = (V, E)$ is an ordered pair consisting of
  - a set V of ***vertices*** (singular: ***vertex***),
  - a set $E \subseteq V \times V$ of ***edges***.

- E can be a set of ordered pair or unordered pairs
  - $G = (V, E)$ is ***directed graph*** if E consists of *ordered* pairs of vertices.
  - $G = (V, E)$ is an ***undirected graph*** if E consists of *unordered* pairs of vertices.
  - Number of vertices: $|V|$
  - Number of edges: $|E|$

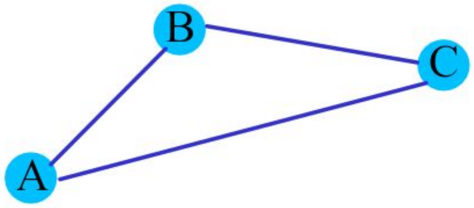
Graph Example

- Here is a graph G = (V, E)
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V



V = {A, B, C, D, E, F}

E = {(A,B),(A,D),(B,C),(C,D),(C,E),(D,E)}



V = {A, B, C}

E = {(A,B),(A,C),(B,C)}

Undirected Graph: Terminology

- u and v are **adjacent** in an undirected graph G if (u,v) is an edge in G
  - edge e = (u,v) is **incident** with vertex u and vertex v
- **degree** of a node: ***deg(v)***: the number of edges incident to v

Directed Graph

- Vertex u is **adjacent to** vertex v in a directed graph G if (u,v) is an edge in G
    - vertex u is the **initial** vertex of (u,v)
- Vertex v is **adjacent from** vertex u
    - vertex v is the terminal (or end) vertex of (u,v)
- the ***indegree*** of a node v, ***indeg(v),*** is the number of edges entering v
- the ***outdegree*** of a node v, ***outdeg(v)***, is the number of edges leaving v
- deg(u) = indeg(u) + outdeg(u)
- **Source**: a node **u** is a source if indeg(u) = 0
- **Sink**: a node **u** is a sink if outdeg(u) = 0

Terminologies

- **A Path of length k**:
  - In the graph $G=(V,E)$, a path from vertex **u** to vertex **v** is a sequence $(v_0, v_1, \cdots, v_k)$ of vertices such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for all i = 1,2,...,k. The length of the path is the number of edges in the path.
- The vertice v is **reachable** from u if there is a path from u to v
- **Simple path:** A path in which all the vertices are distinct
- **Cycle:** A special type of path where starting and ending vertices are the same $(v_0, v_1, \cdots, v_k)$ forms a cycle if $v_0 = v_k$, and the path contains at least one edge.
- A cycle is a special type of path starting and ending at the same vertex.
- A graph with no cycles is **acyclic**.
- **Subgraph**: graph ***H = (V', E')*** is subgraph of G = (V, E) if V' is subset of V, and E' is a subset of E.

Graph properties

In both undirected and directed graphs: **$|E| = O(|V|^2)$**

- **Proof**:
  - every edge connects two distinct vertices (G has no loops)
  - No two edges connect the same pair of vertices (G has no multi-edges)
  - G has at most $\binom{n}{2}$ edges in an undirected graph and 2x $\binom{n}{2}$ in a directed graph

$O(V + E) = O(V^2)$

- Which one is a better runtime? $O(v^2)$ or $O(V+E)$?
  - $O(V + E)$

Graph properties

- A graph is called **dense** if $E = \Theta(V^2)$
  - Most pairs of vertices are connected by an edge
- A graph is called **sparse** if it is not dense: $E \ll V^2$
  - There are very few edges in the graph
- **connected graph**: an undirected graph in which every vertex is reachable from all other vertices
- In an undirected graph, if the graph is connected:
  - There must be at least $|V| - 1$ edges $\rightarrow |E| \geq |V| - 1$
  - Proof by induction on the number of vertices
- $E = \Theta(V^2)$ and $|E| \geq |V| - 1 \rightarrow \log |E| = \Theta(\log V)$
- **Strongly connected graph**: a *directed* graph in which every two vertices are reachable from each other

Graph

- Tree:
  - A connected (undirected) graph without any cycle
  - Tree is a graph with exactly one path between any pair of vertices
  - Yet another definition: a tree is a connected graph with $|V|-1$ edges, i.e., $|E|=|V|-1$
    - Proof by induction

Storing Graphs

There are two ways to store a graph:

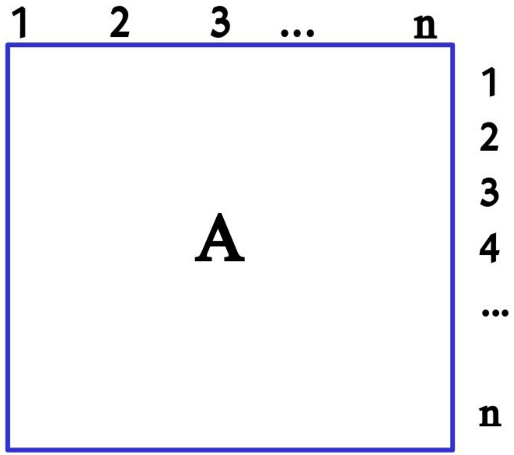
- Adjacency matrix
- Adjacency list

Storing Graphs: Adjacency-matrix

The adjacency matrix of a graph G = (V, E)

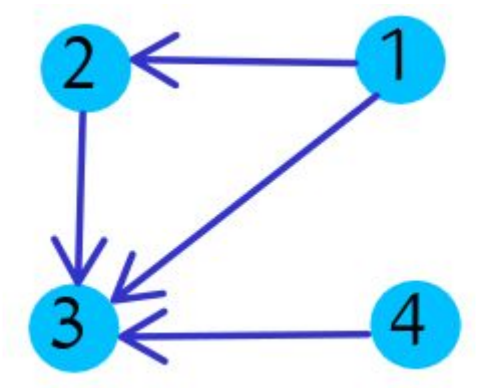
- V = {1, 2, …, n} is the the set of vertices of the graphs
- is an **n** x **n** matrix A

$$A[i,j] = \begin{cases} 0 & \text{if } (i,j) \notin \mathrm{E} \\ 1 & \text{if } (i,j) \in \mathrm{E} \end{cases}$$



- Space : $O(n^2)$

Storing Graphs: Adjacency-matrix: Example



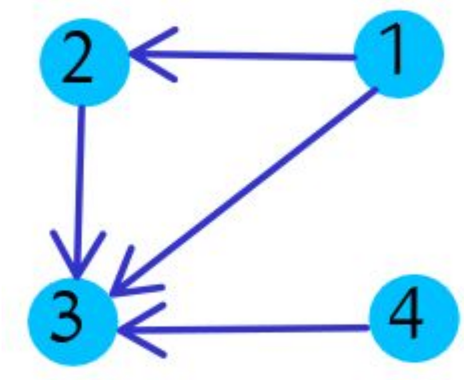
$$
\begin{array}{c|cccc}
A & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0
\end{array}
$$

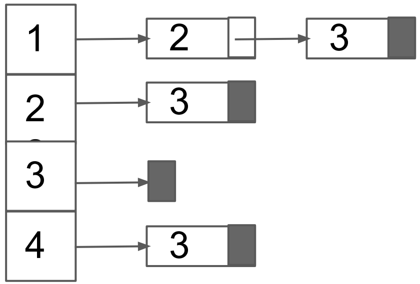
$\Theta(V^2)$ storage

Storing Graphs: Adjacency list

- An adjacency list of a vertex **v ∈ V** is the list **Adj[v]** of vertices adjacent to v
- For undirected graphs, **|Adj[v]| = degree(v)**
- For directed graphs, **|Adj[v]| = out-degree(v)**
- adjacency lists use **Θ(V + E)** storage
- **Handshaking Lemma:** for undirected graphs $\sum_{v \in V} deg(v) = 2|E|$

$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$





Storing Graphs

- Adjacency Matrix vs Adjacency List

	Adjacency matrix	Adjacency list
$(u, v) \in E$	$\Theta(1)$	$O(\deg(u))$
Time to list u's neighbor	$\Theta(n)$	$\Theta(\deg(u))$
Time to list all edges	$\Theta(n^2)$	$\Theta(n+m)$
Space complexity	$\Theta(n^2)$	$\Theta(n+m)$

Storing Graphs

Adjacency Matrix

- Advantage:
  - O(1) test for presence or absence of edges
- Disadvantage:
  - Inefficient for sparse graphs
  - Storage not efficient
  - Accessing edges not efficient

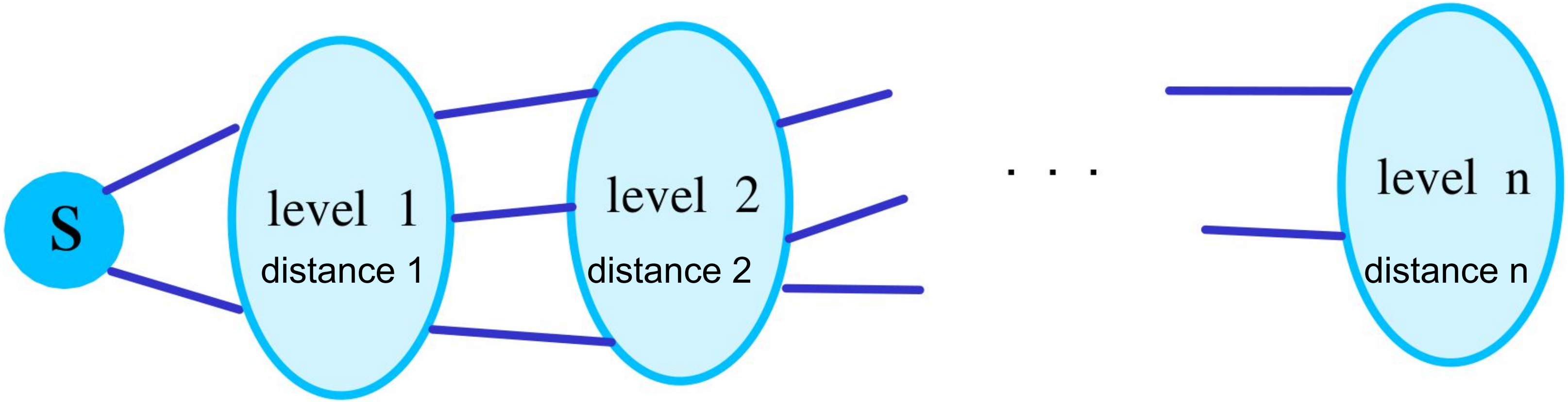
Adjacency List

- Advantage:
  - Good for sparse graphs
  - Accessing edges are easy
- Disadvantage:
  - More complex data structure
  - Not possible to access an edge in O(1)

BFS

Breadth-First search

- A graph traversal algorithm
- **Input**: A graph G=(V, E) and a source vertex s
- **Output**:
  - Visits the vertices in order of their distance from s
  - Find the shortest distance from s to each reachable vertex

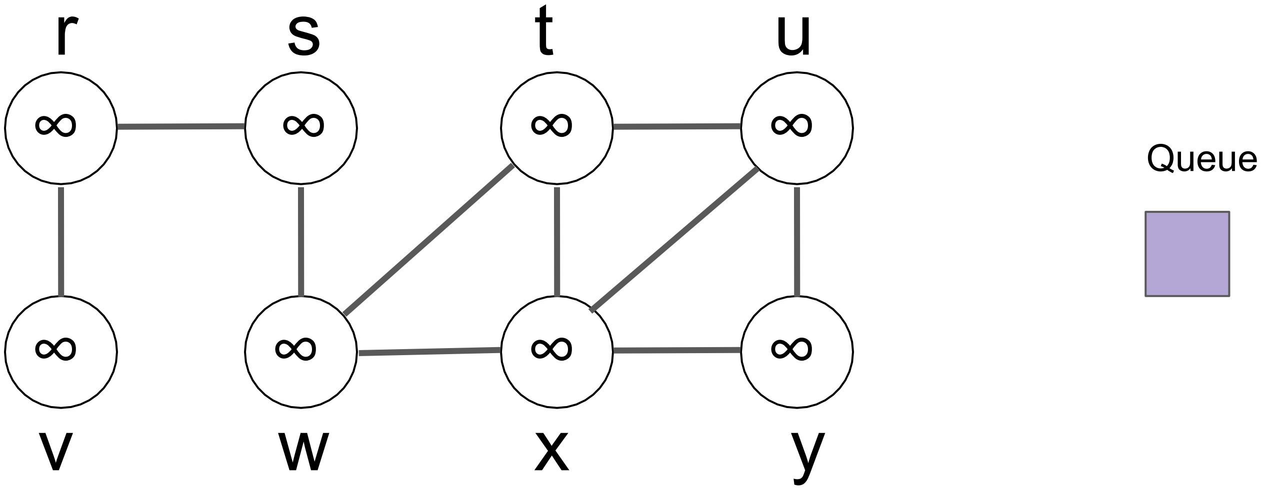


Breadth-First search: pseudocode: Basic version

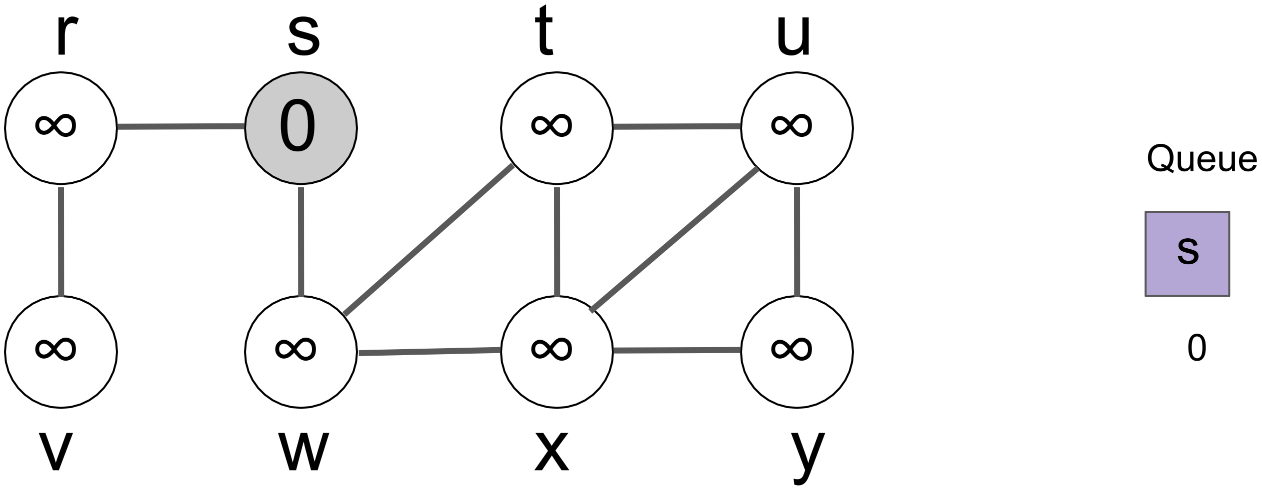
- During execution of the algorithm, the vertices are in one of the three following states:
  - Undiscovered (White)
  - Discovered (Gray)
  - Fully-explored (Black)

```
Initialization: mark all vertices undiscovered(white)
BFS(G, s)
    mark s "discovered"
    Q = [s]
    while Q not empty
        u = dequeue(Q)
        for each neighbor v of u:
            if (v is undiscovered):
                mark v discovered
                enqueue(Q, v)
        mark u fully-explored
```

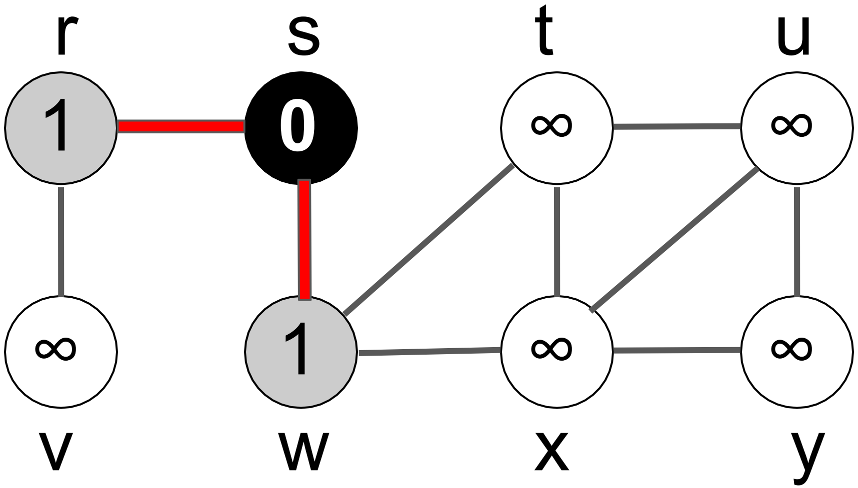
BFS Example



BFS Example



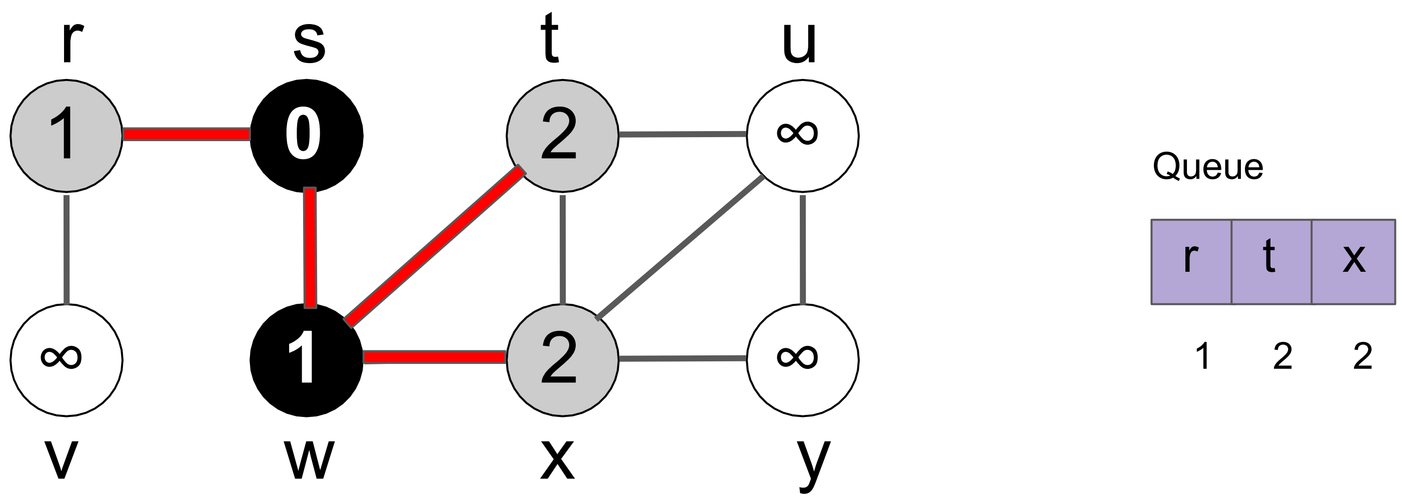
BFS Example



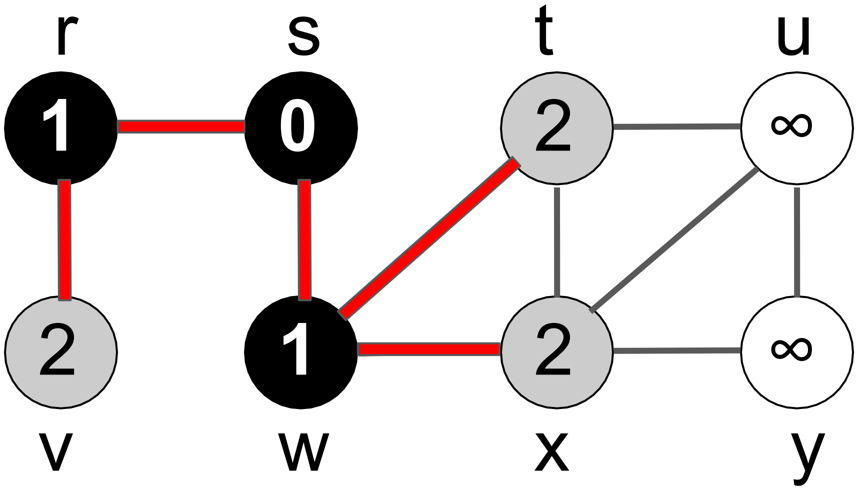
Queue

w	r
1	1

BFS Example



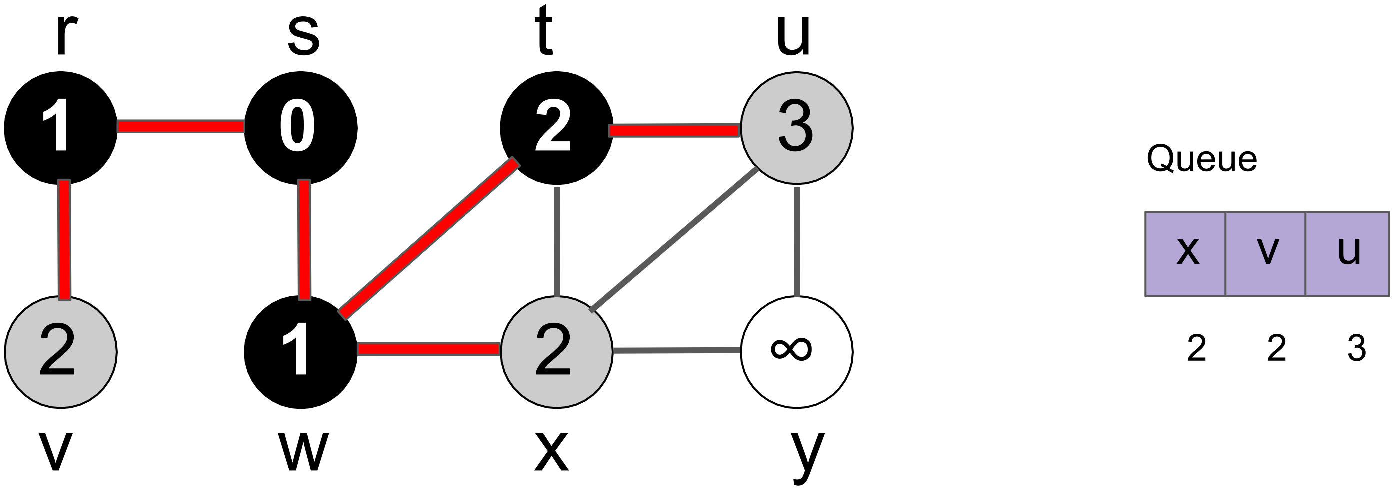
BFS Example



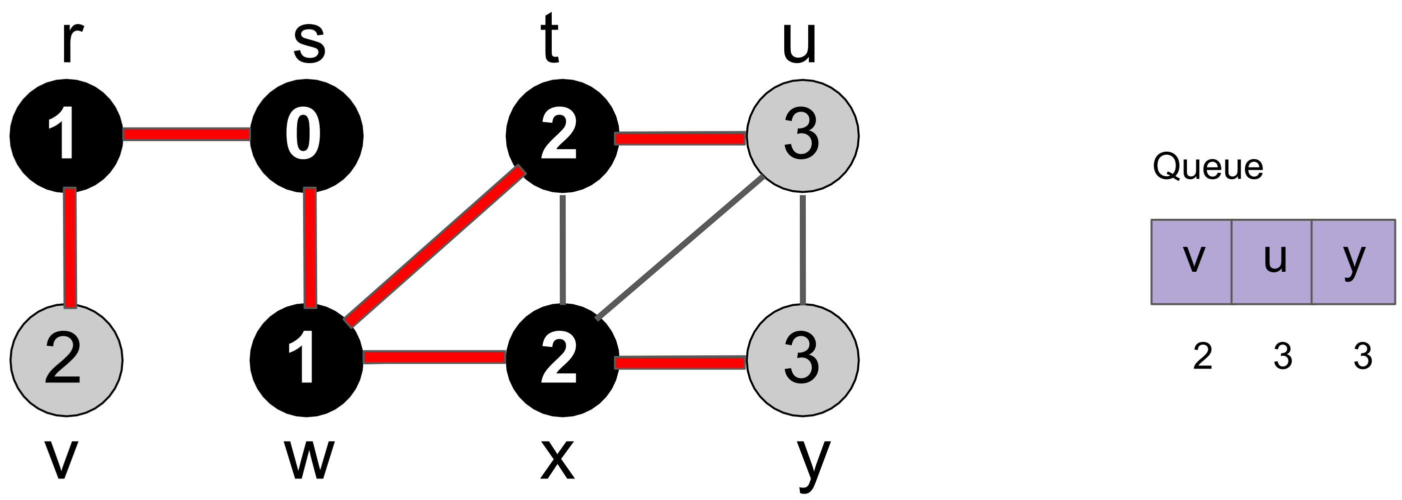
Queue

t	x	v
2	2	2

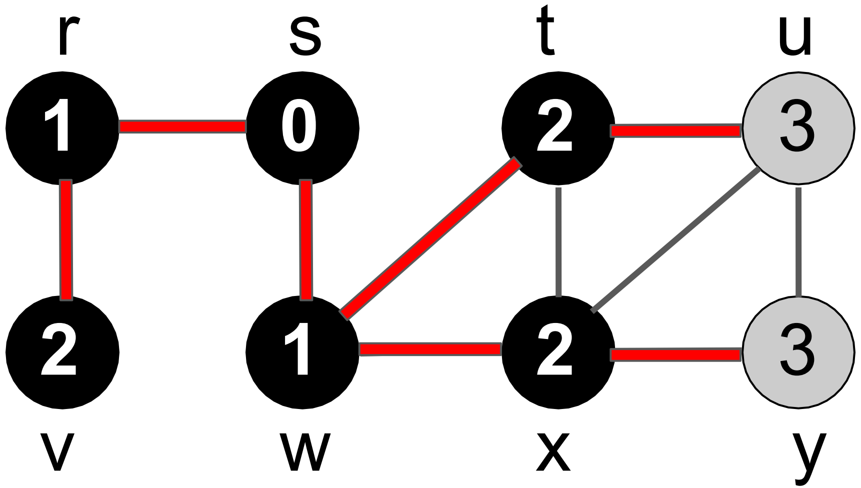
BFS Example

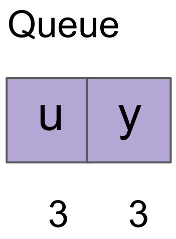


BFS Example

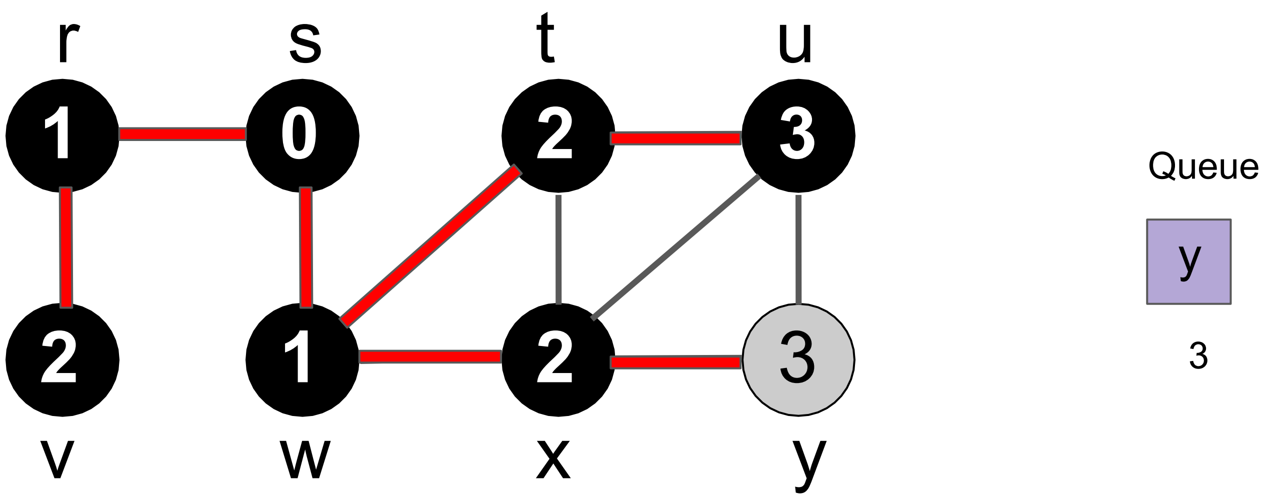


BFS Example

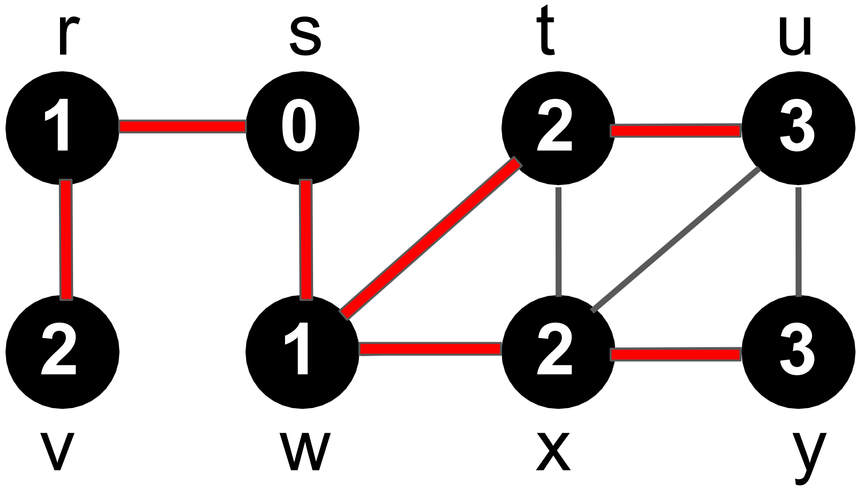




BFS Example



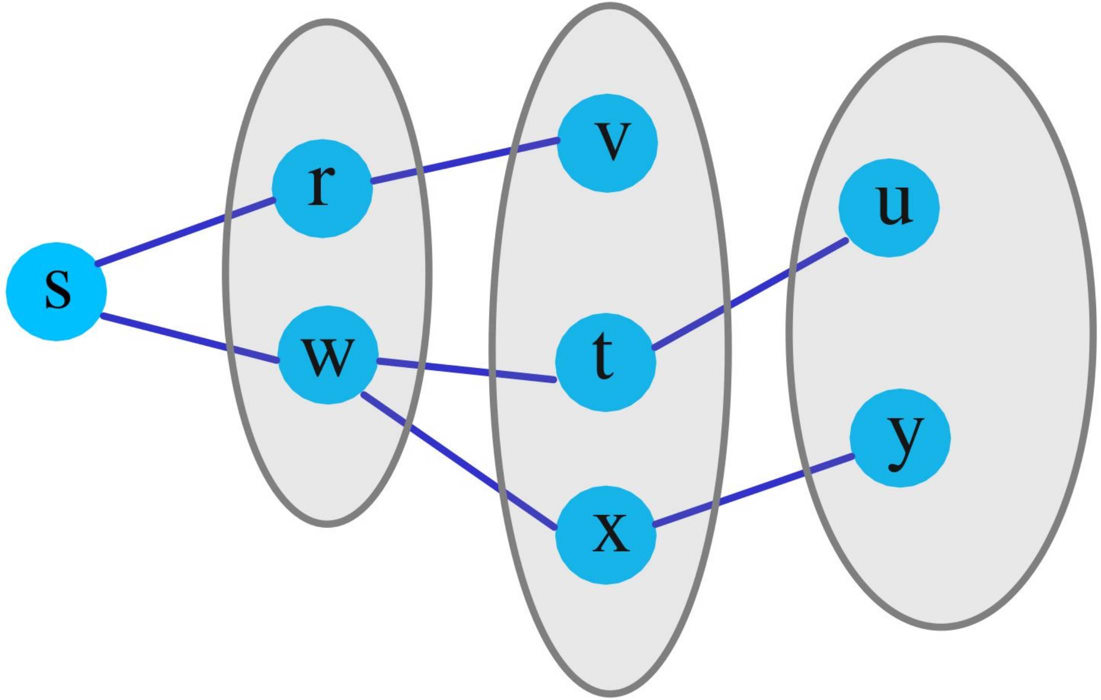
BFS Example



Queue

∅

BFS Example



Distance 1 Distance 2 Distance 3

BFS: Basic version: Runtime Analysis: Naive Analysis

```
Initialization: mark all vertices undiscovered
BFS(G, s)
    mark s "discovered"
    Q = [s]
    while Q not empty
        u = dequeue(Q)
        for each neighbor v of u:
            if (v is undiscovered):
                mark v discovered
                enqueue(Q, v)
        mark u fully-explored
```

O(V): for every vertex in G

O(V)

Runtime: $O(V^2)$

BFS: Basic version: Runtime Analysis: Aggregate Analysis

```
Initialization: mark all vertices undiscovered
BFS(G, s)
    mark s "discovered"
    Q = [s]
    while Q not empty
        u = dequeue(Q)
        for each neighbor v of u:
            if (v is undiscovered):
                mark v discovered
                enqueue(Q, v)
        mark u fully-explored
```

O(V): for every vertex in G

O(deg(u))

- Each vertex is enqueued at most once (when it is discovered)
- When a vertex u is dequeued, the for loop is executed for **deg(u)** iterations
- So the total time complexity is $O(n + \sum_{u \in V} deg(u))$ = O(n+m)

BFS: Finding the path from s to v

- **How to trace back a path from s to v**
  - We can add an array **parent[v]**
  - When a vertex v is discovered within the for loop of vertex u, then we set parent[v] = u.
  - Now to trace out a path v to s, we just need to write a for loop that starts from v and keep going to its parent until we reach vertex s
  - For all vertices v reachable from s, the edges (v, parent[v]) form a tree, called the **BFS tree**
- Also useful to store level (distance)

BFS: Detailed Version

```
BFS(G, s)
    Q = ∅
    for each node u in G:
        dist[u] = ∞
        color[u] = WHITE
        pred[u] = NULL
    dist[s] = 0
    color[s] = GRAY  #mark s "discovered"
    enqueue(Q, s)
    while Q not empty
        u = dequeue(Q)
        for each neighbor v of u:
            if color[v] == WHITE:  #(u is undiscovered)
                color[v] = GRAY  #mark u discovered
                distance[v]= dist[u] + 1
                pred[v] = u
                enqueue(Q, v)
        color[u] = BLACK #mark u fully-explored
```

BFS: Shortest paths

- $\delta(s, v)$ = shortest path distance from s to v
    - Minimum number of edges in any path from vertex s to vertex v
- $\delta(s, v) = \infty$
    - If there is no path from s to v
- A shortest path path from s to v has length $\delta(s, v)$
- **BFS correctly computes the shortest path distances**

BFS: Proof of correctness

- To prove correctness of BFS,
  - Upon termination of BFS, dist[v] = $\delta(s, v)$ for all v in V
  - *Therefore, all the vertices that are reachable from s must be discovered otherwise dist[v]* $= \infty > \delta(s, v)$.

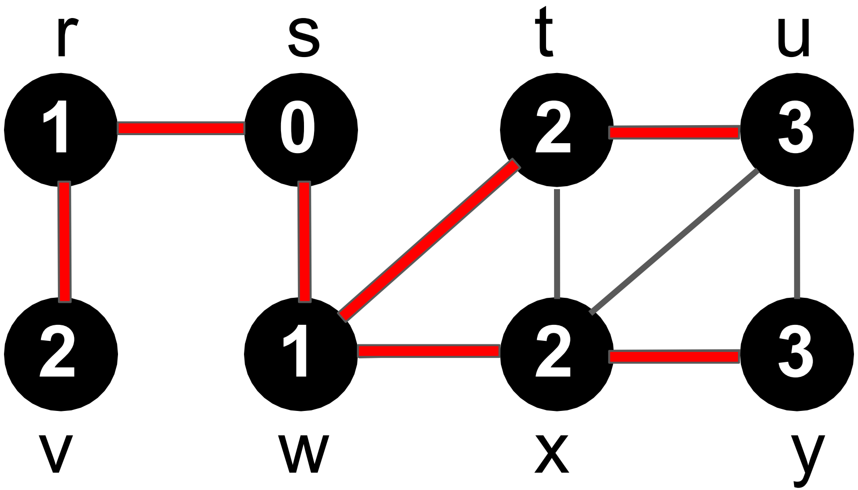
BFS Application

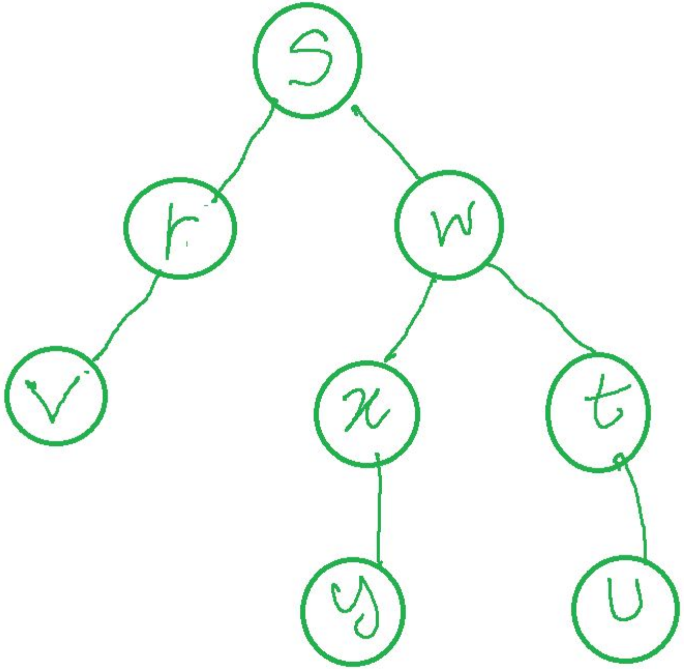
- BFS can be used to check whether a graph is connected or not
  - Checking whether all vertices are marked discovered
- The connected component containing s
  - by returning all the vertices which are discovered
- Whether there is a path from s to v
  - Checking whether v is discovered

- Exercises:
  - Enhance BFS to find all connected components in time $O(n + m)$
    - If a graph is not connected
  - Use BFS to find if a connected graph has a cycle.
  - Prove that if (u, v) in E then $level(u)$, $level(v)$ differ by 0 or 1.

Breadth-First Tree

- BSF builds a breadth-first tree:
  - A tree where the path from s to every node is the shortest path





Breadth-First Tree

- Subgraph G'=(v', E') is generated after running the algorithm BFS
  - V' = {v ∈ V: pred[v] ≠ NULL} U {s}
    - V' consists of vertices in V which are reachable from s, since the algorithm sets pred[v]=u if and only if (u,v) is an edge in E and v is reachable from s → G' is a connected graph
  - E' = { (pred[v], v): v ∈ V' - {s} }
- G' is a Tree since it is connected and |E'| = |V'| - 1
- Since G' is a tree there is a unique simple path from s to every vertex in V'
- According to Theorem 2, this path is the shortest path.
  - How? You can prove that using induction and using Theorem 2