# Graphs:
# Breadth First Search

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 8

# Announcements

- Midterm marks are released in Markus.

  Midterm solutions are available in Piazza.

- A3 handout and A2 marks will be posted this week.
- Tutorial on Friday: Problems on amortized cost and BFS.

# Today

- Graphs
  - Applications
  - Formal definitions
  - Graph representations
- Graph Traversal
  - Breadth-First Search, BFS
    - ✓ BFS algorithm
    - ✓ BFS run time analysis
    - ✓ Proof of correctness
  - Depth-First Search, DFS

# Reading Assignments 

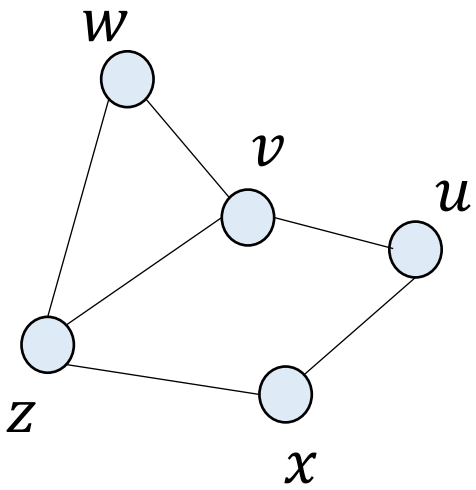
22.1, 22.2

# Graphs

- Very important ADT that is used to model relationships between objects.

**Graph G = $(V, E)$**

- Objects: vertices/nodes $V$
- Connections: edges/arcs $E$

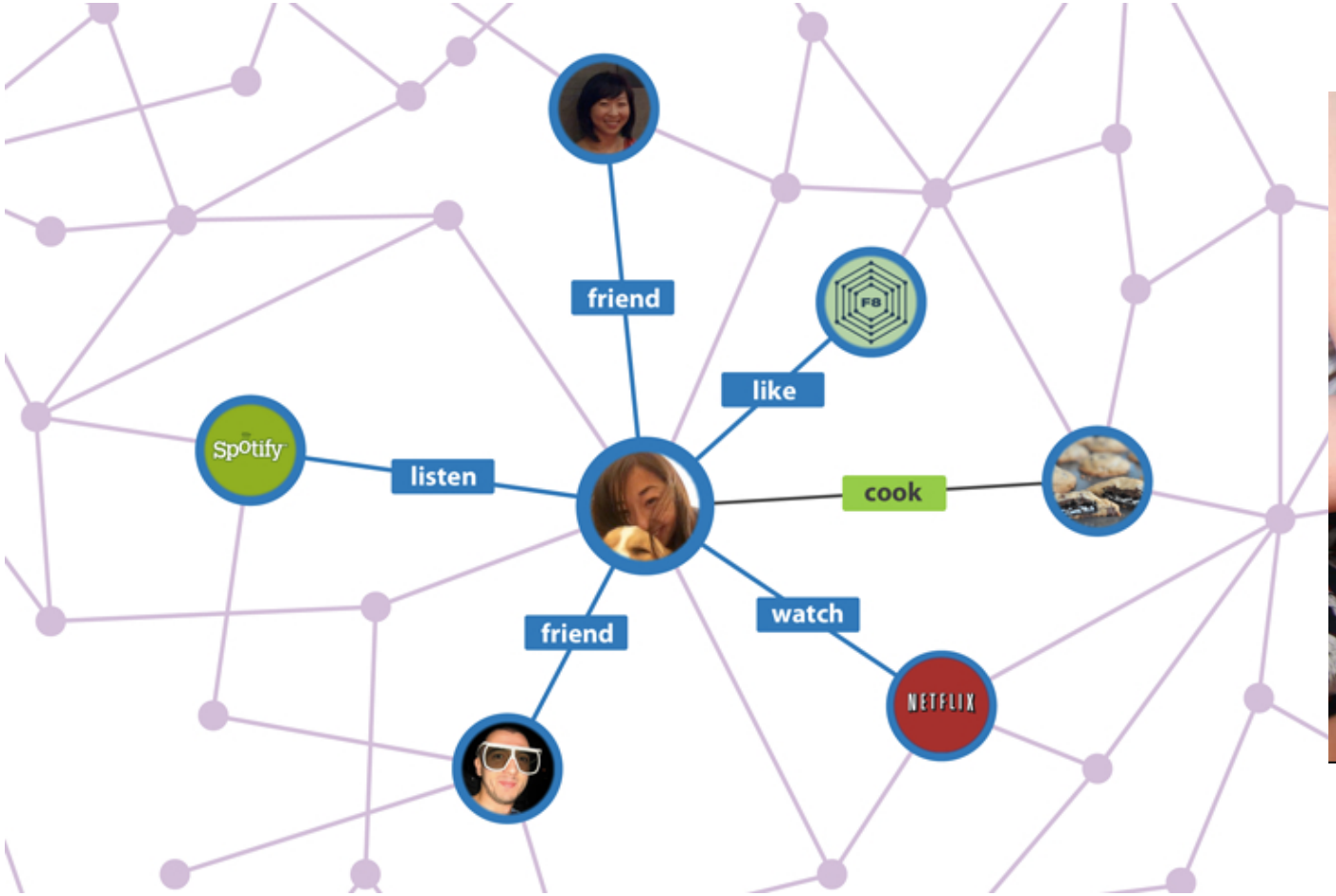


# Graphs – Transportation network

- Maps and GPS

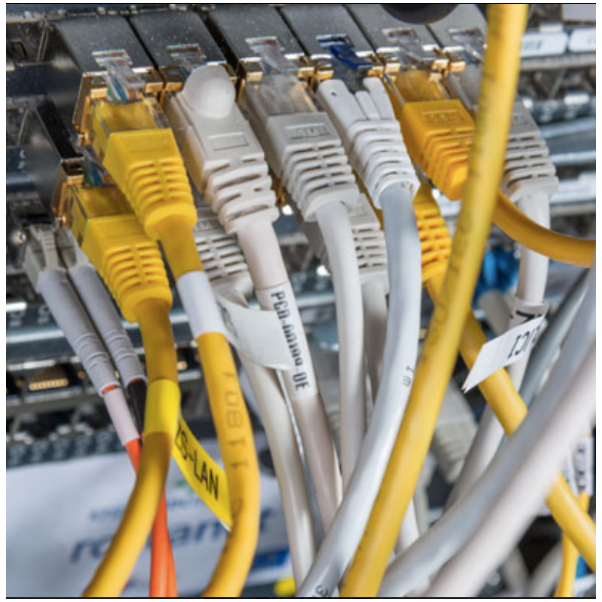
# Graphs – Social networks

- LinkedIn, Facebook, Tweeter



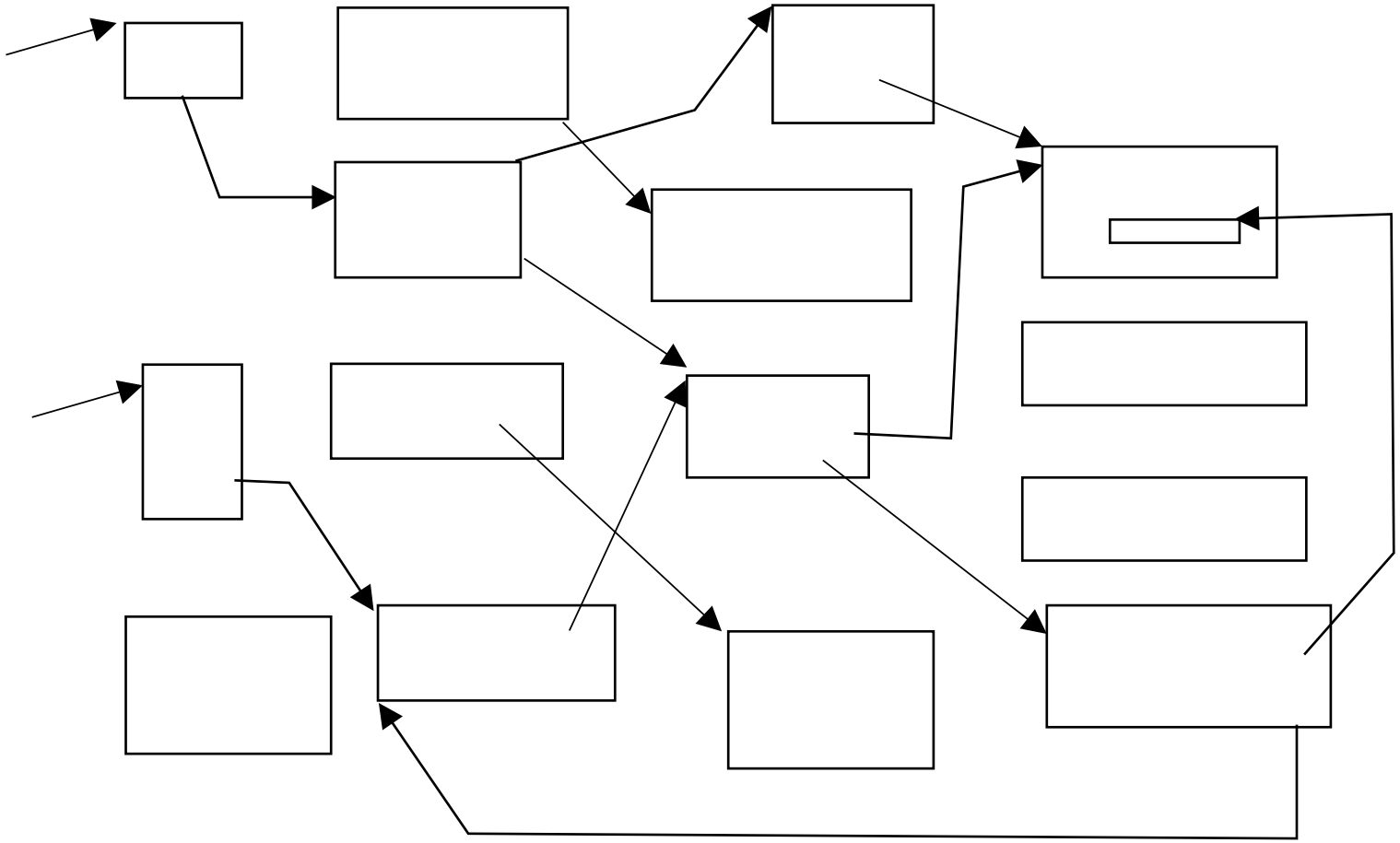

# Graphs – Computer Network

- Web



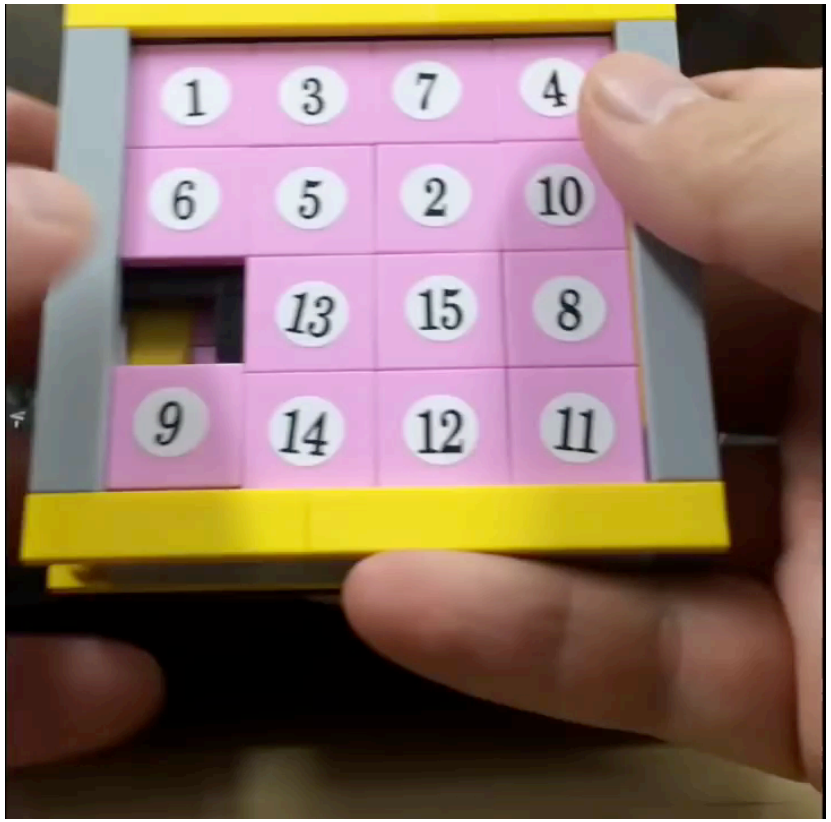
# Graphs – Compiler

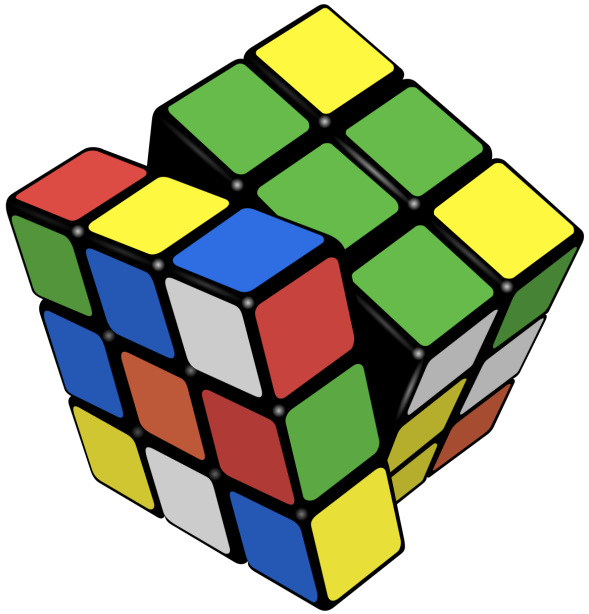
- garbage collection



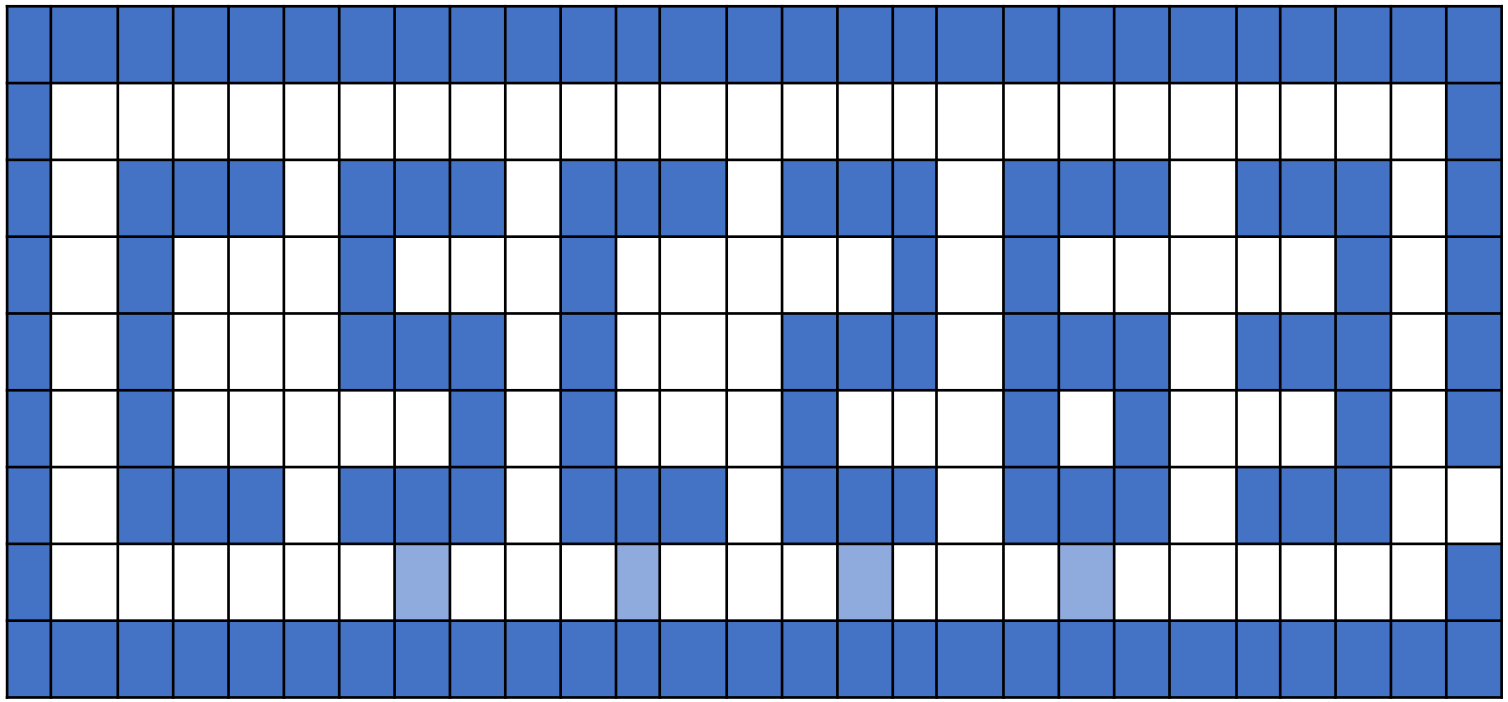
# Graphs – Games and puzzles

- Sliding puzzle
- Rubik's cube



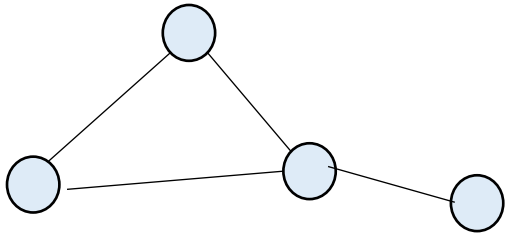



- Solving a maze!

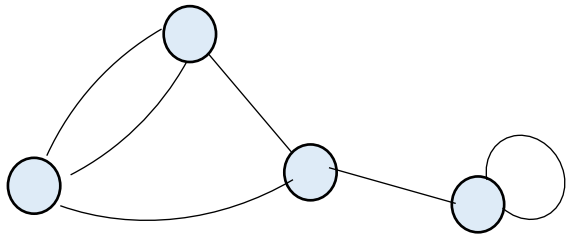



# Graph – definitions

- **Simple graph:** No multiple edge, no self-loop.



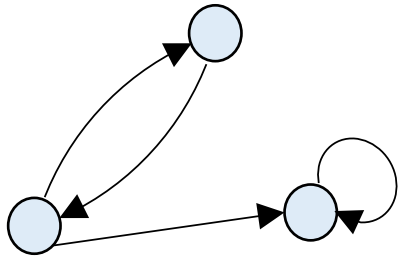
- **Non-simple graph**



# Graph – definitions

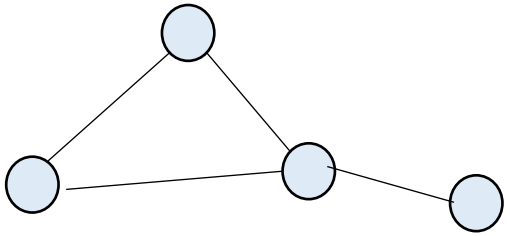
- **Directed graph**: $E$ contains ordered pairs $(u, v)$ in $V^2$

  $(u, v) \neq (v, u)$ and $(v, v)$ (self-loops) allowed.



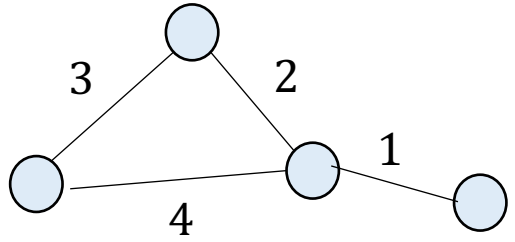
- **Undirected graph:** $E$ contains unordered pairs $\{u, v\}$

  V: $\{u, v\} = \{v, u\}$ and $\{v, v\} = \{v\}$ disallowed.

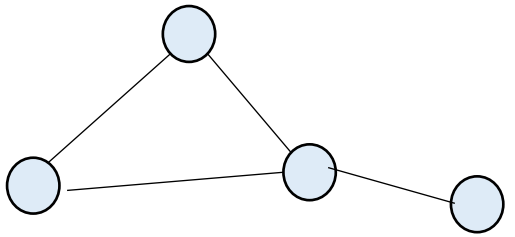


# Graph – definitions

- **Weighted graph:** all edges in $E$ are associated with some weight/cost $w(e) \in \mathbb{R}$.




- **Unweighted graph:** no weight/cost for edges.



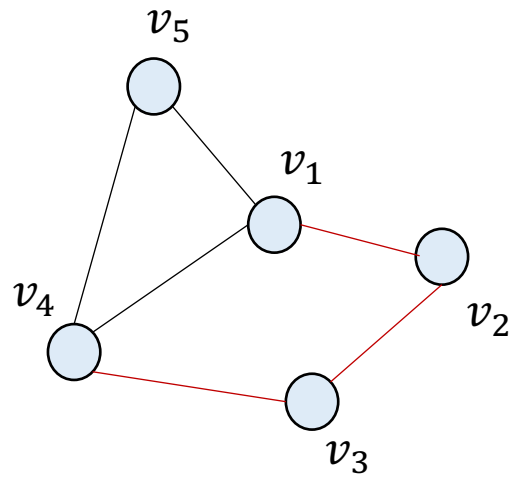
# Graph – definitions

- **Path:** sequence of edges connected to each other:

  $(v_1, v_2), (v_2, v_3), \dots, (v_k, v_{K+1})$

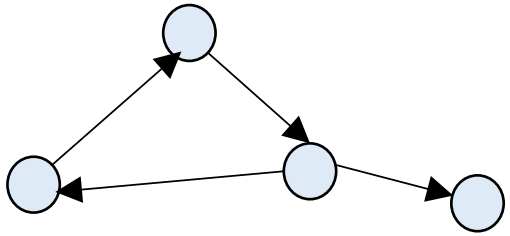
- **Length of path:** number of edges in the path.

- **Simple path:** no repeated edge or vertex.

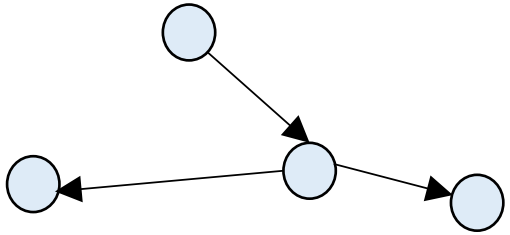


# Graph – definitions

- **Cycle:** path with end vertex = start vertex.
- **Cyclic graph:** A graph with at least one cycle.

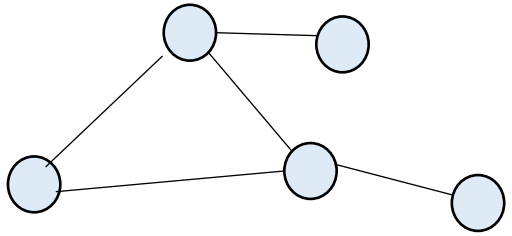


- **Acyclic graph:** A graph without any cycle.

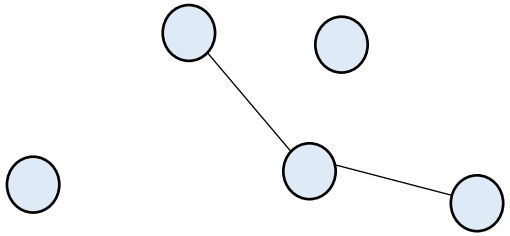


# Graph – definitions

- **Connected graph:** contains path between any two vertices.

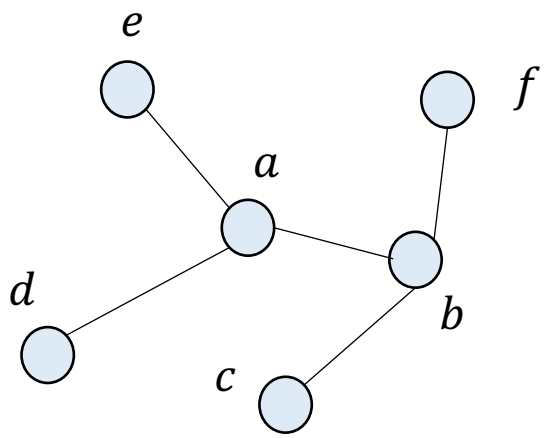


- **Disconnected graph:** has a pair of vertices with no any path between them.

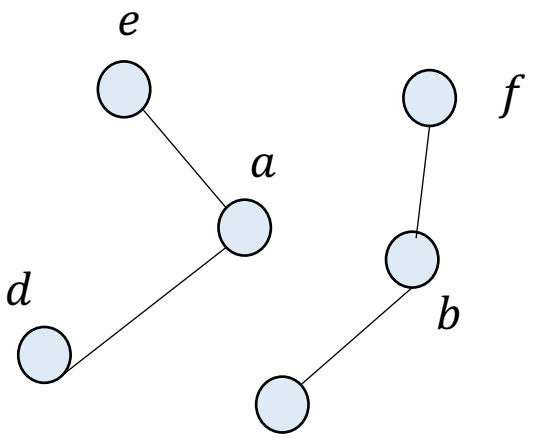


# Graph – definitions

- **Tree:** graph that is connected and acyclic.

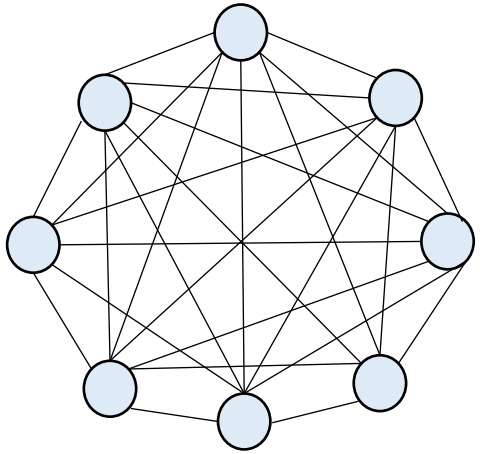



- **Forest:** acyclic graph (collection of disjoint trees, each one a "connected component")



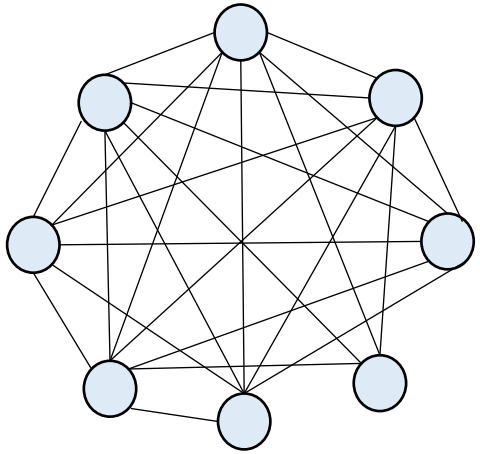
# Graph – definitions

- **Complete graph:** a graph in which there is an edge between all the pair of vertices.

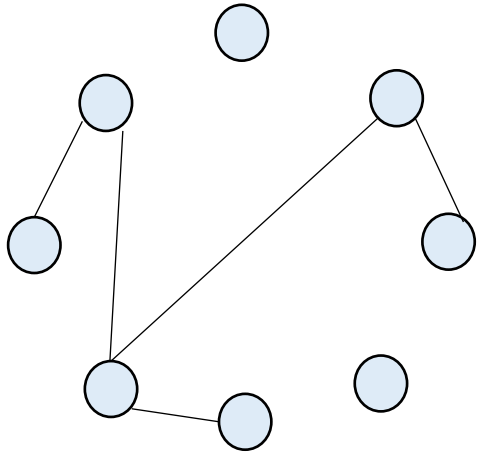


# Graph – definitions

- **Dense graph:** a graph in which the number of edges is close to the maximal number of edges.



- **Sparse graph:** a graph with only a few edges

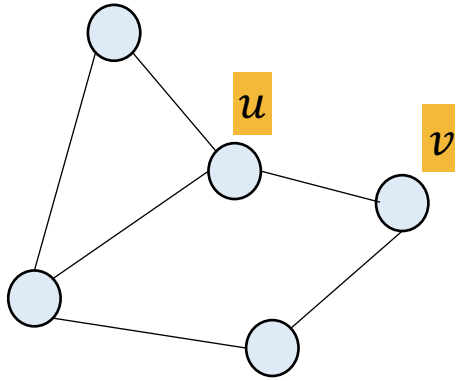


# review

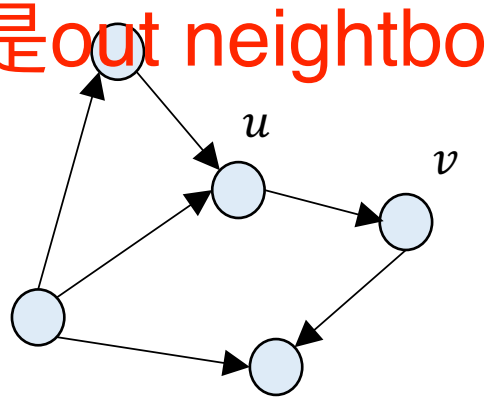
- Simple graph
- Directed graph
- Weighted graph
- Path, length, simple path
- Cycle, Cyclic graph, Acyclic graph
- Connected graph
- Tree
- Forest
- Complete graph
- Dense, sparce graph

# Standard operations on graphs:

- **Add or remove a vertex \ Add or remove an edge**
- **Edge Query:** given vertices $u, v$, does $G$ contain edge $(u, v)$ or $\{u, v\}$?
- **Neighbourhood** (undirected graph)**:** given vertex u, return set of vertices $v$ such that $\{u, v\} \in E$.
- **In-neighbourhood/out-neighbourhood** (directed graph): given vertex u, return set of vertices $v$ such that $(v, u)/(u, v) \in E$.
- **Degree/in-degree/out-degree:** size of neighbourhood/ in-neighbourhood/ out-neighbourhood.
- **Traversal:** visit each vertex of a graph to perform some task.

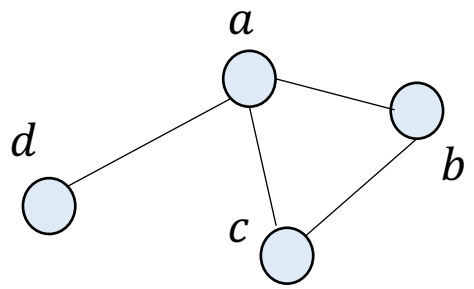



V 是out neightbour



# How to implement graph?

Data structures for the graph ADT




- Adjacency matrix: $|V| \times |V|$ matrix $A$

  Let $V = \{v_1, v_2, \ldots, v_n\}$

$$A[i,j] = \begin{cases} 1 & if\ (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

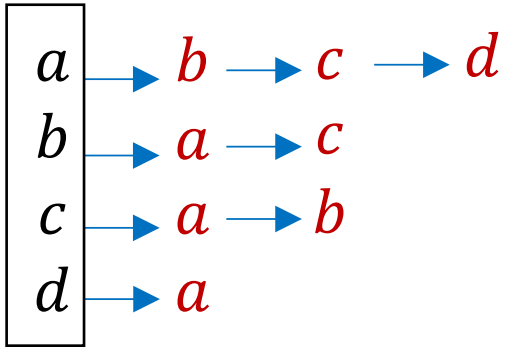
Space: $|V|^2$

| | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| $a$ | 0 | 1 | 1 | 1 |
| $b$ | 1 | 0 | 1 | 0 |
| $c$ | 1 | 1 | 0 | 0 |
| $d$ | 1 | 0 | 0 | 0 |

- Adjacency list:

We have an array of lists $A$. $A[i]$ belongs to vertex $v_i$ and stores a list of $v_j$ that satisfy $(v_i, v_j) \in E$.

Space: $|V| + 2|E|$



# Matrix vs List

In term of space complexity

- adjacency matrix is $\Theta(|V|^2)$
- adjacency list is $\Theta(|V| + |E|)$

Which one is more space-efficient?

- Adjacency list, if $|E| \ll |V|^2$ , i.e., the graph is not very dense.

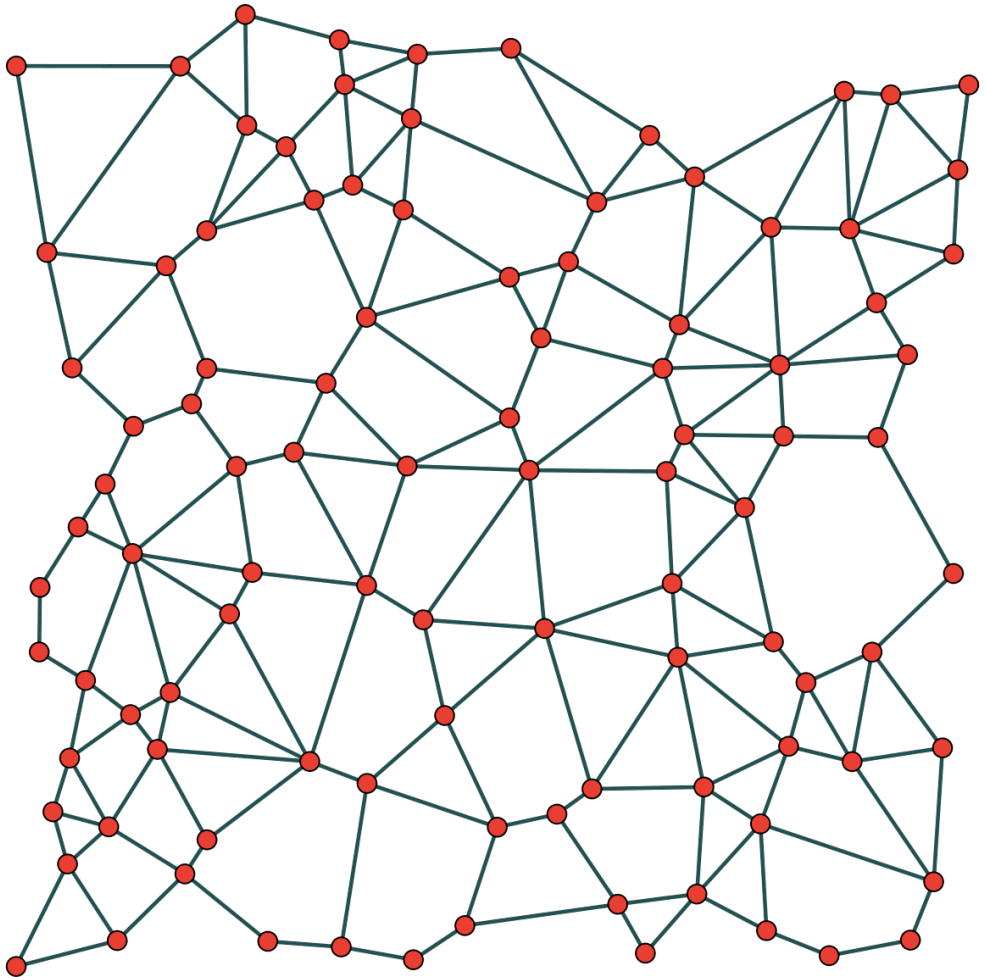
**Anything that Matrix does better than List?**

Edge query: Check whether edge $(v_i, v_j)$ is in $E$

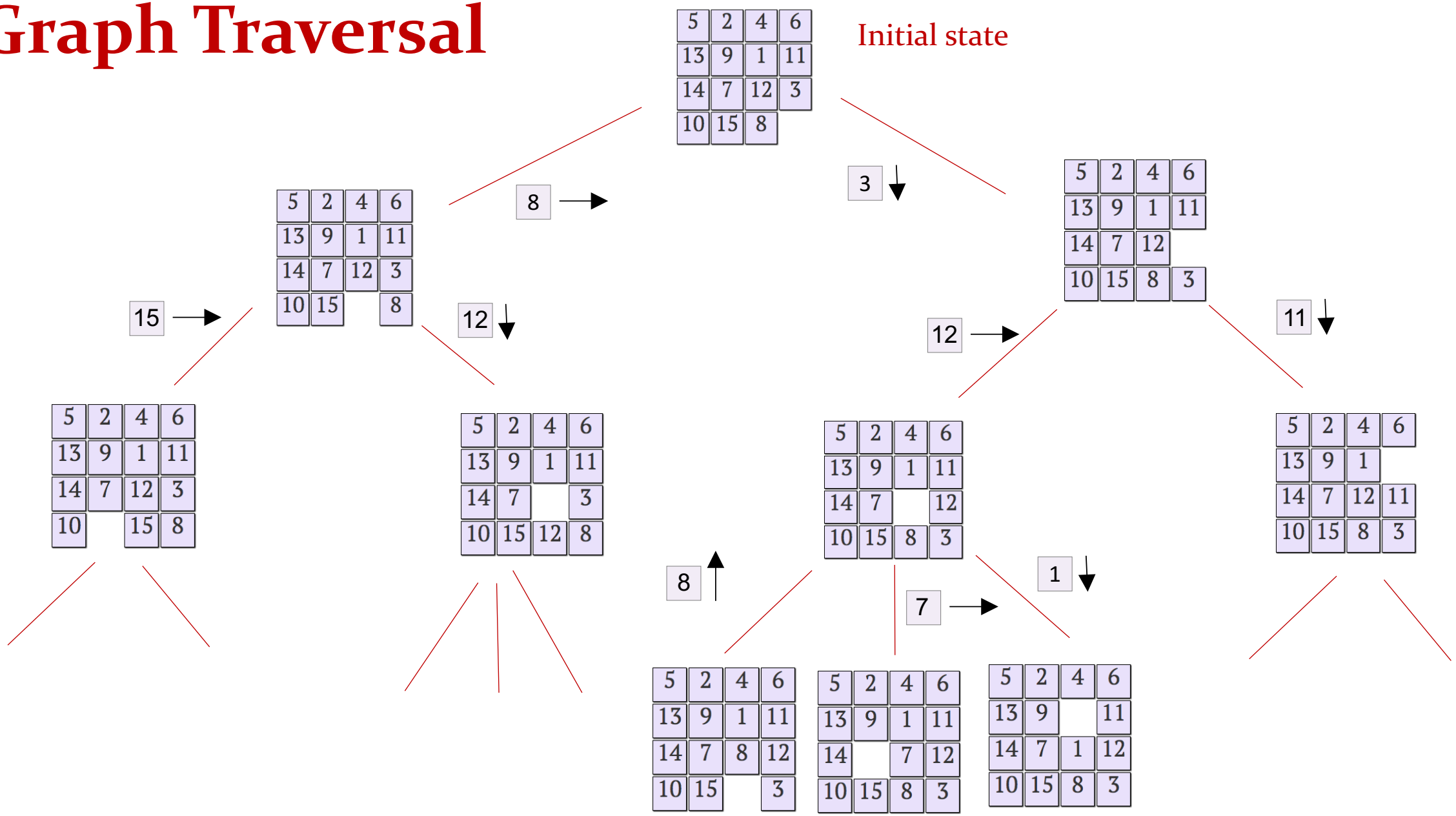
- Matrix: just check if $A[i,j] = 1$, $O(1)$
- List: go through list $A[i]$ see if $j$ is in there, $O(length\ of\ list)$

# Graph Traversal

- BFS: Breadth First Search
- DFS: Depth First Search

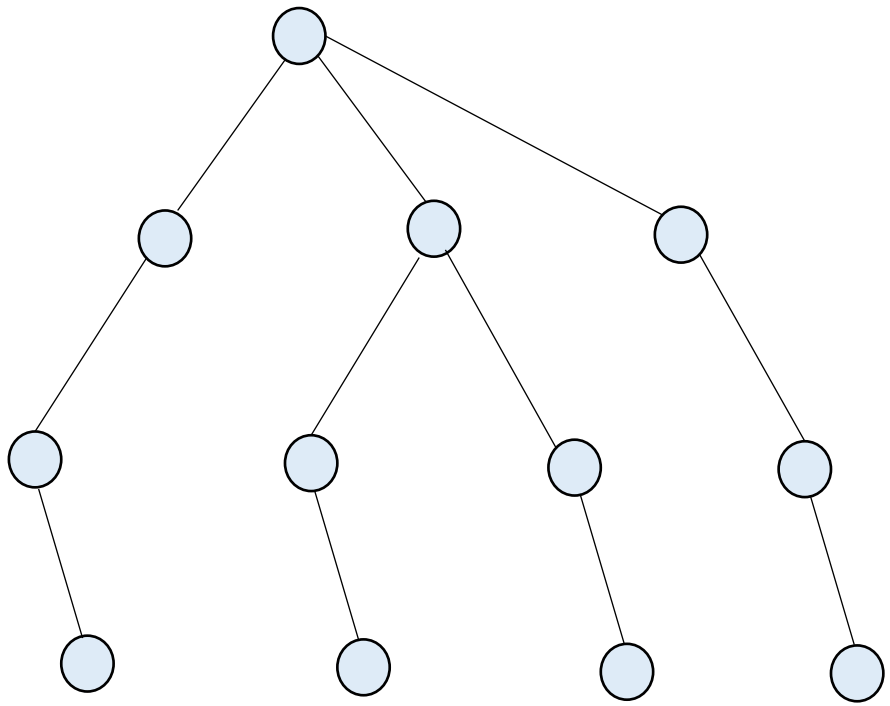


# Graph Traversal

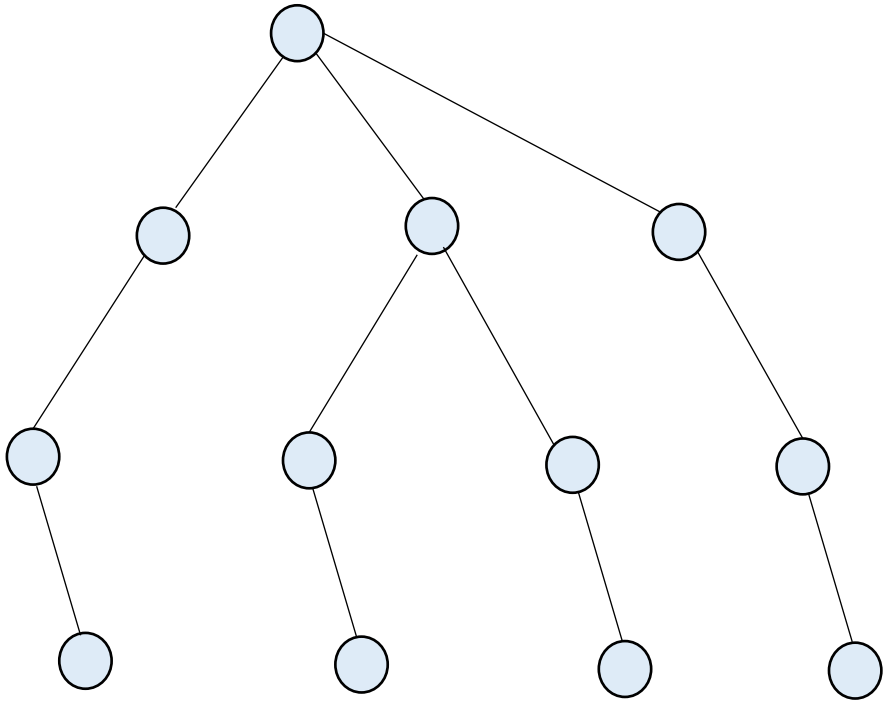


# BFS vs DFS

**BFS:**

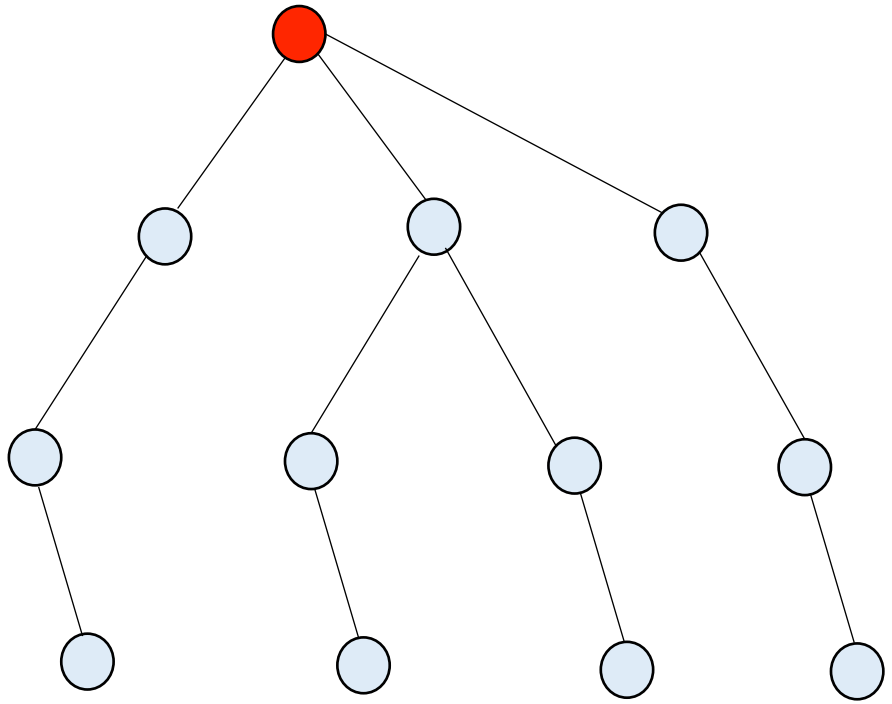


**DFS:**

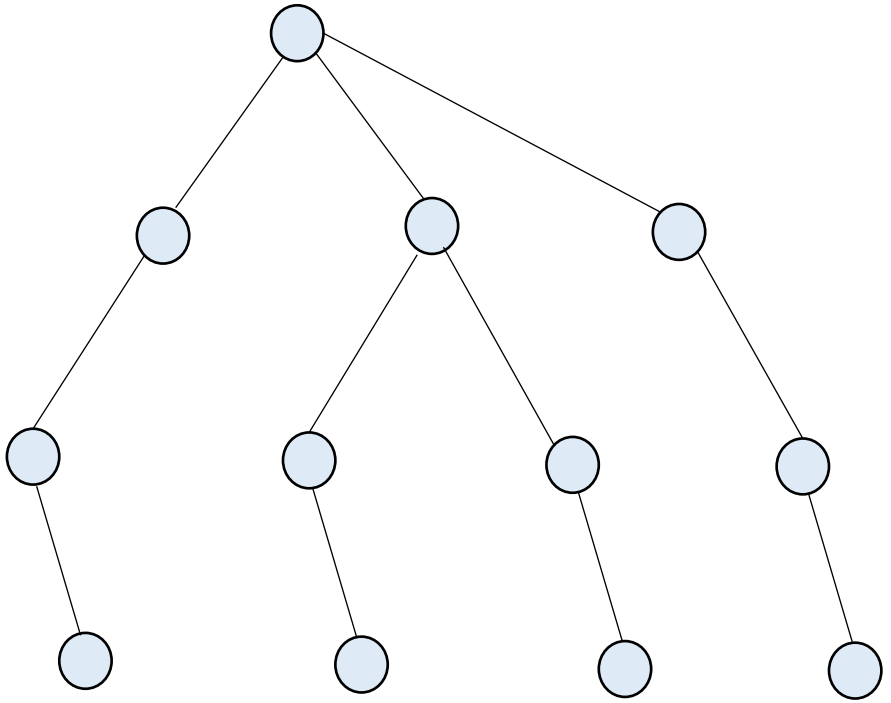


# BFS vs DFS

**BFS:**

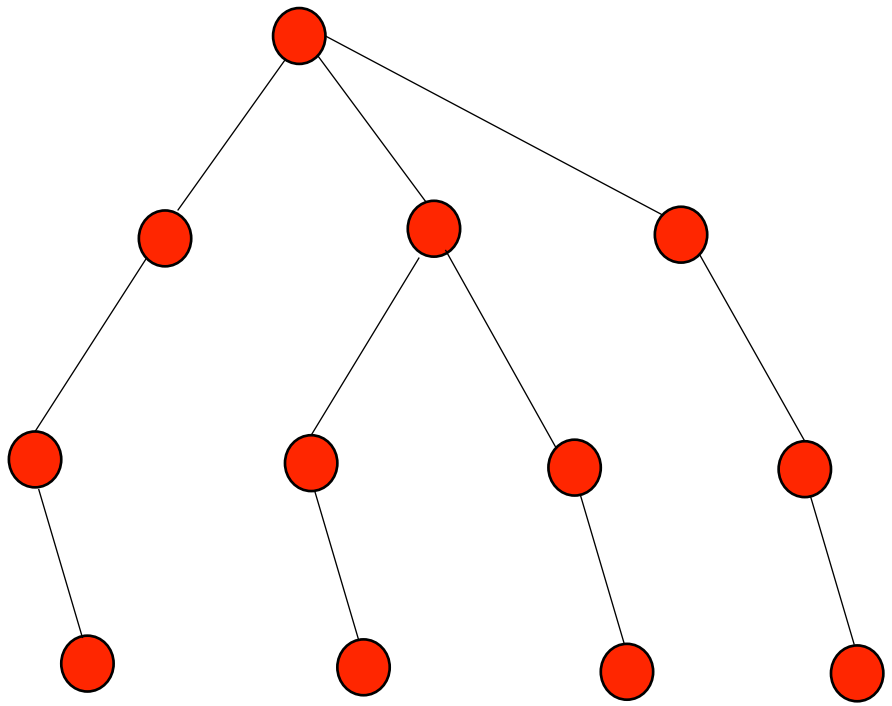


**DFS:**

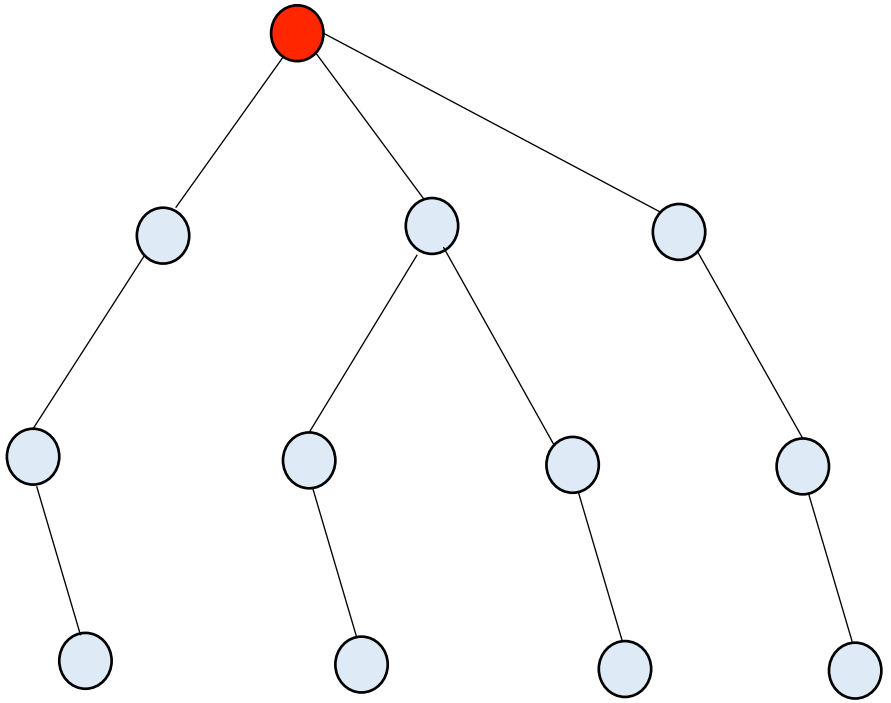


# BFS vs DFS

**BFS:**

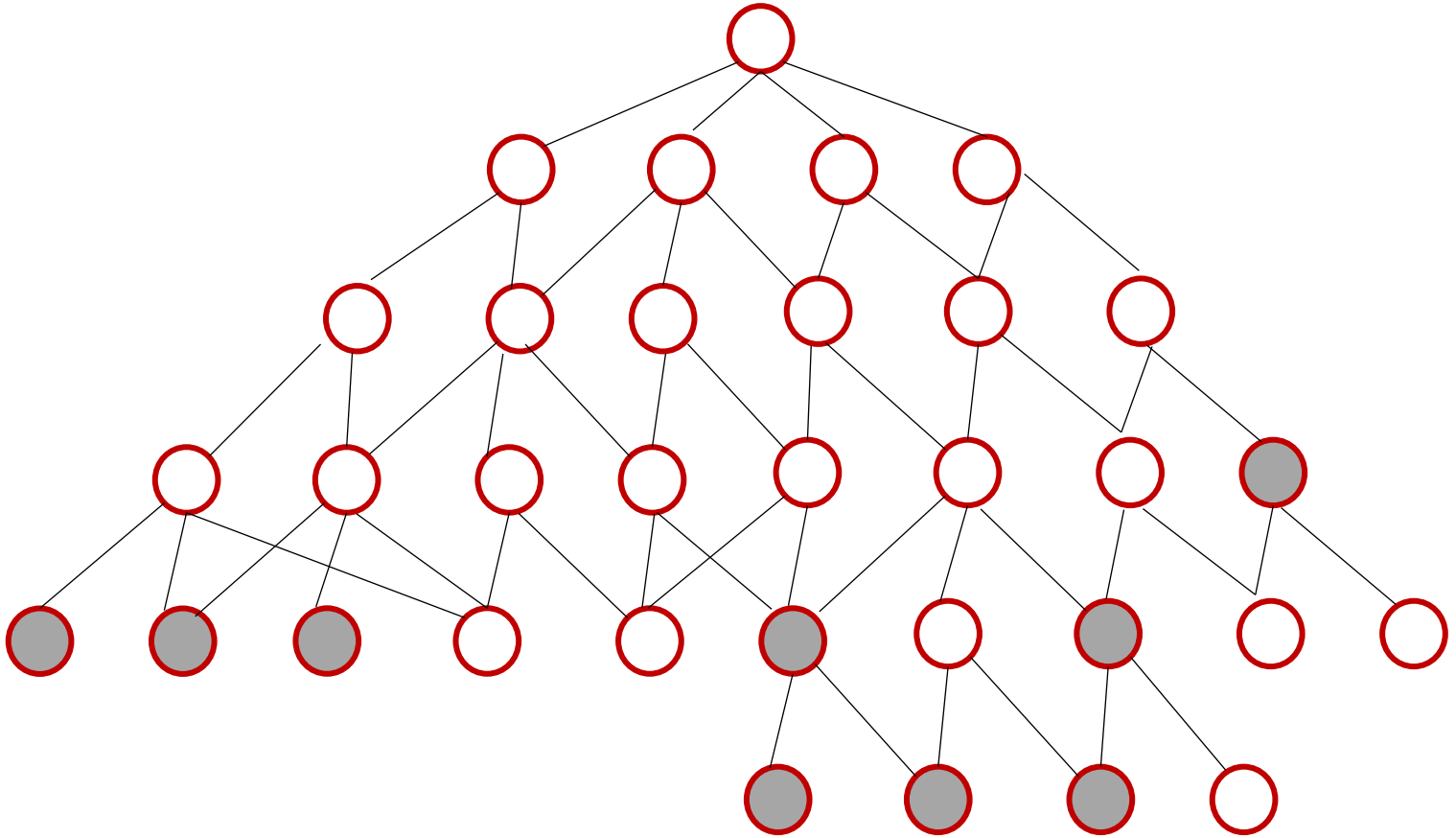


**DFS:**



# Which one is a better choice? BFS or DFS?

- Using BFS and DFS depends on the problem.
- If you expect to find the answer in deeper nodes DFS would be a better choice.



# Breadth First Search

Algorithm

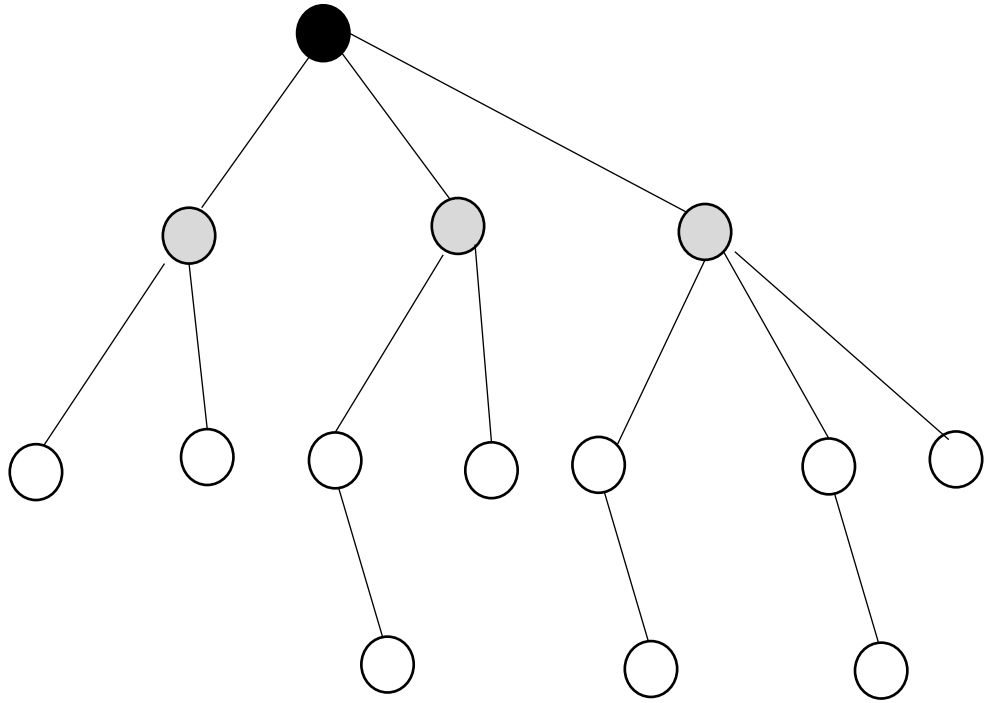
# BFS in a Graph

There is a cycle: How avoid visiting a vertex twice

Remember you visited it by

labelling it using colors.

- White: “unvisited”
- Gray: “encountered”
- Black: “explored”

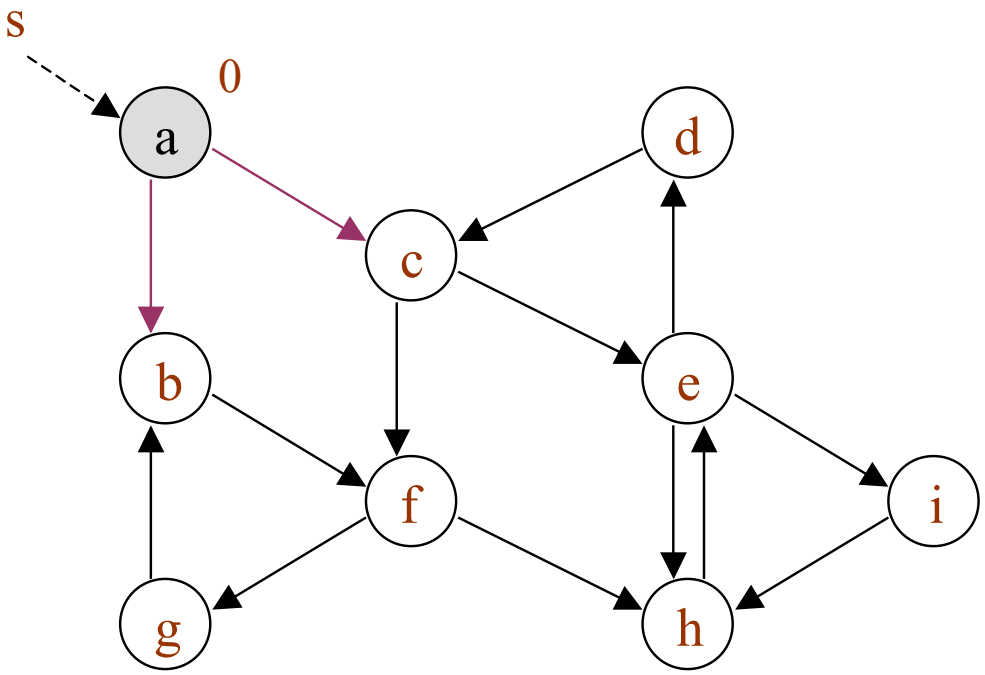


# Main approach

- Initially all vertices are white.
- Color a vertex gray the **first time** visiting it.
- Color a vertex black when all its neighbour have been encountered
- Avoid visiting gray or black vertices.
- In the end, all vertices are black.

- Some other values we want to remember during the traversal.
  - $v.\pi$ the vertex from which $v$ is encountered
  - $v.d$: the distance value

# BFS – Pseudocode (from CLRS)




Q: a

$BFS(G, s)$

for **each** vertex $u \in V - \{s\}$

$u.color = WHITE$

$u.d = \infty$

$u.\pi = NIL$

$s.color = GRAY$

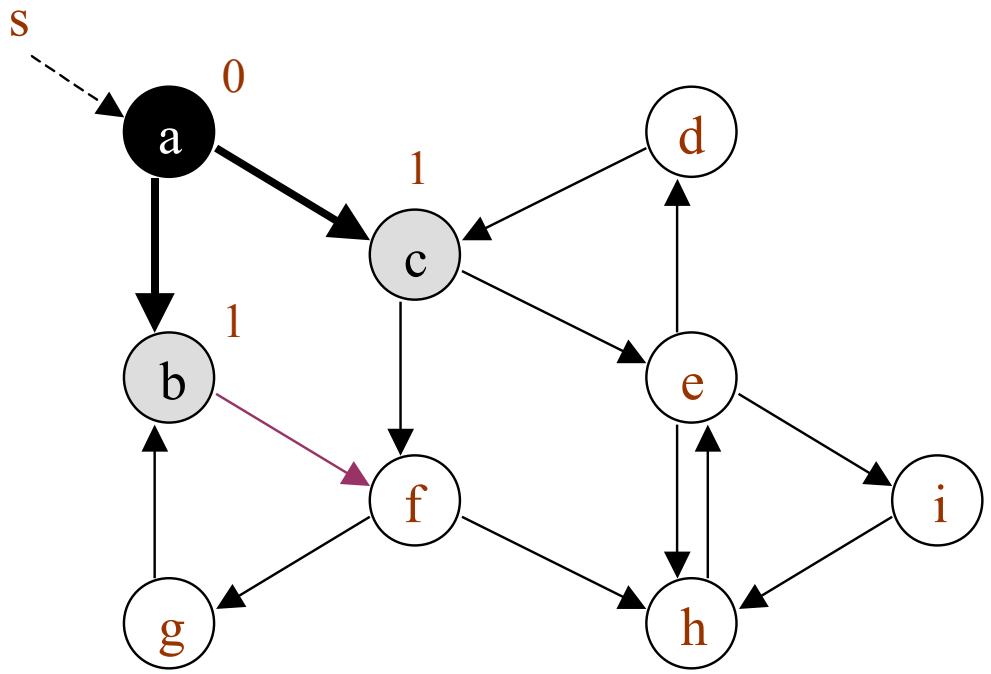
$s.d = 0$

$s.\pi = NIL$

$Q = \emptyset$

ENQUEUE$(Q, s)$

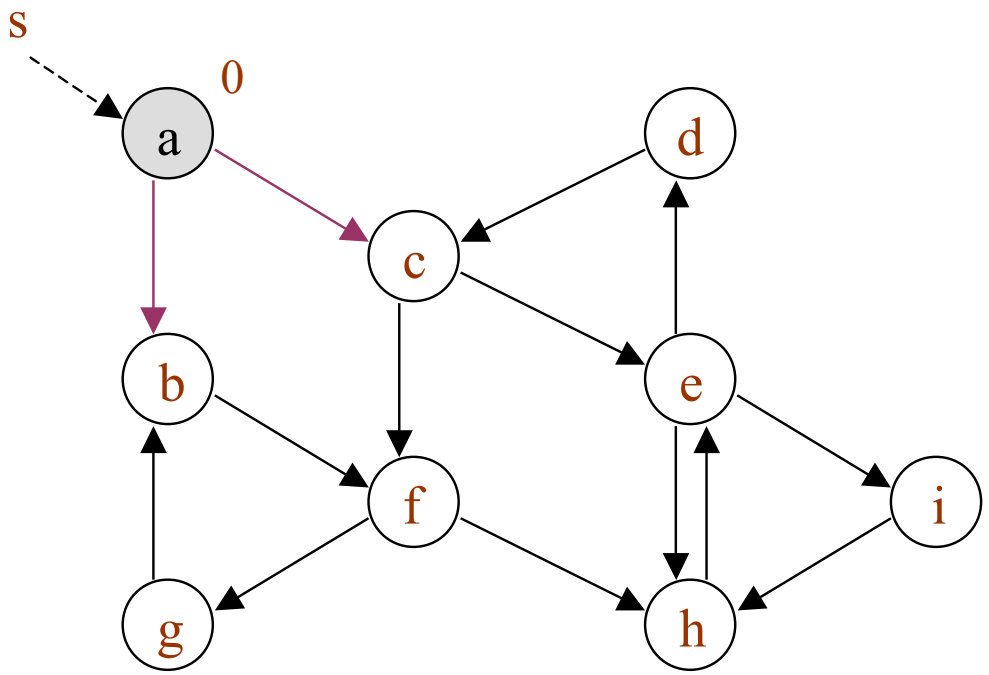
# BFS – Pseudocode (Cont'd)




while $(Q \neq \emptyset)$
    $u = DEQUEUE(Q)$
    **for** each $v \in G.Adj[u]$
        if $v.color == WHITE$
            $v.color = GRAY$
            $v.d = u.d + 1$
            $v.\pi = u$
            $ENQUEUE(Q, v)$
    $u.color = BLACK$

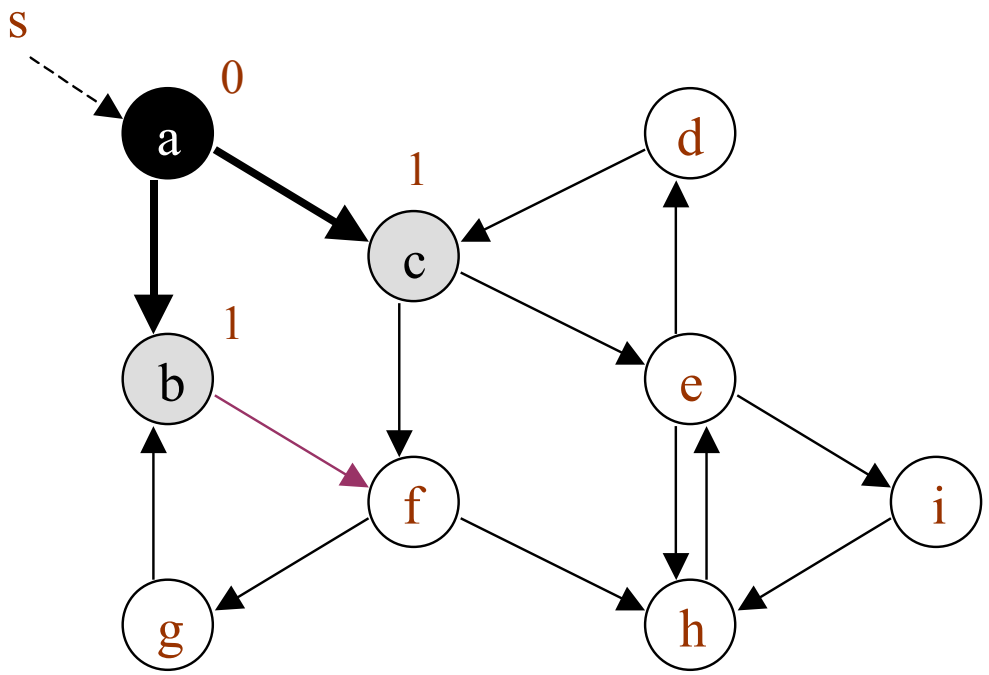
# Breadth-First Search

Sample Graph:



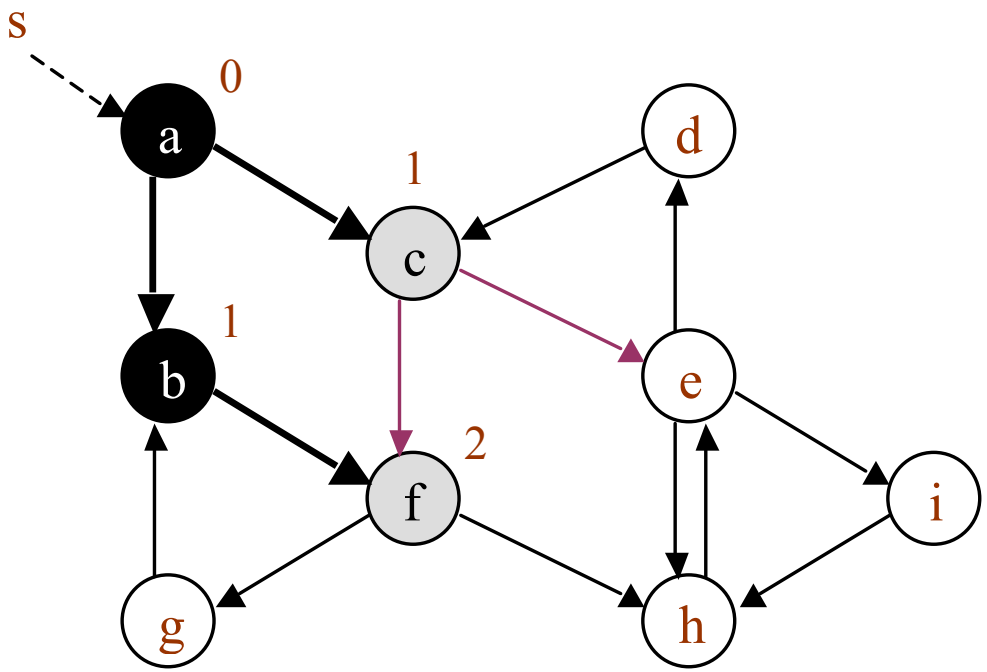

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ ↑ | - |

# Breadth-First Search



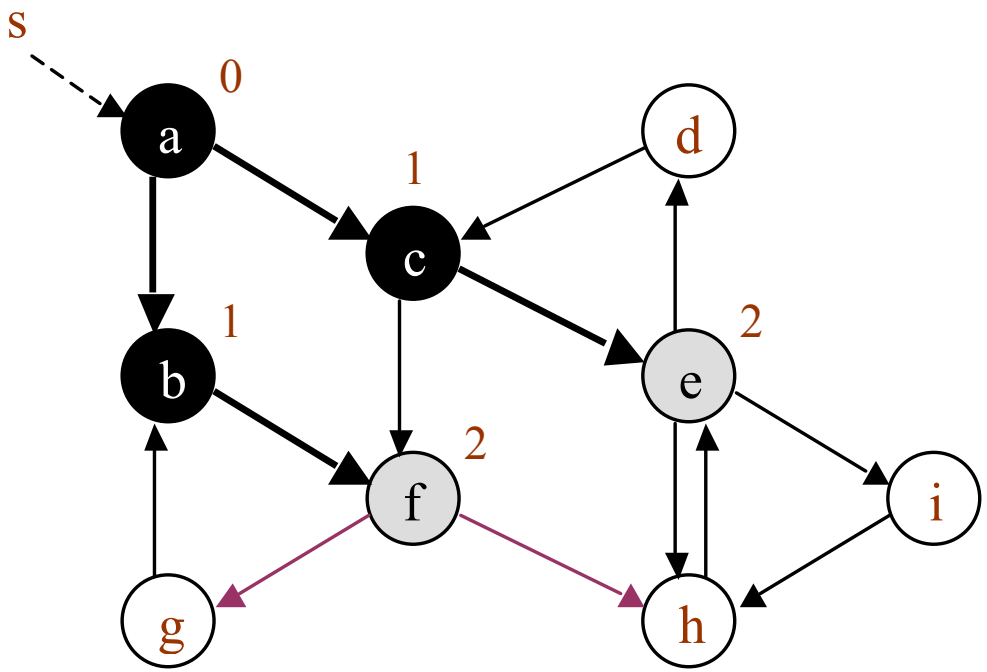

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |

# Breadth-First Search



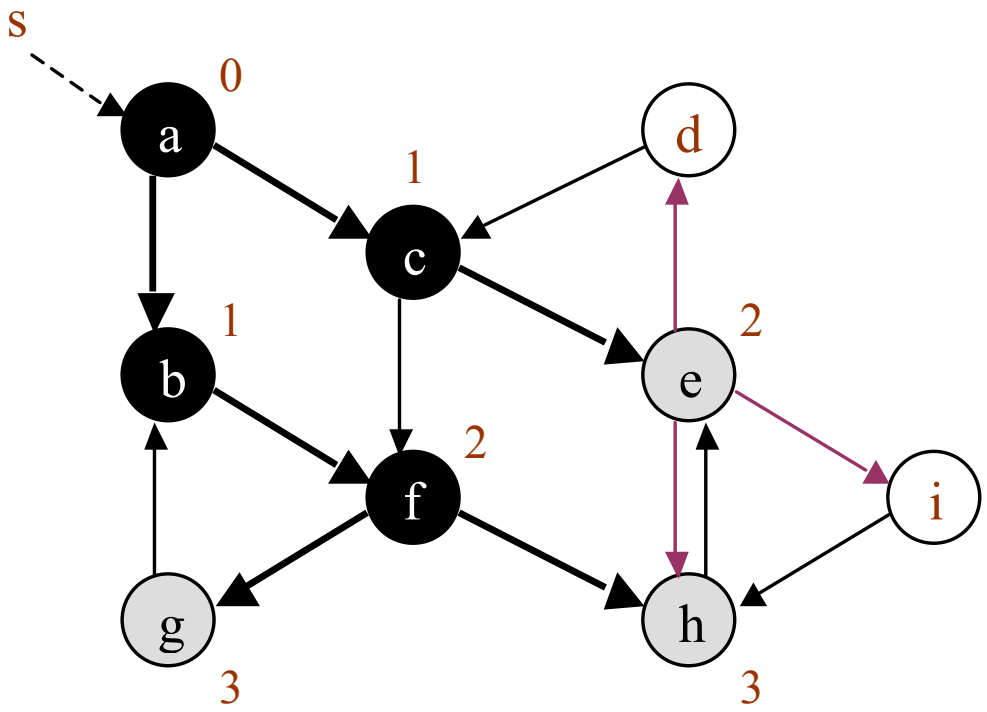

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |

# Breadth-First Search



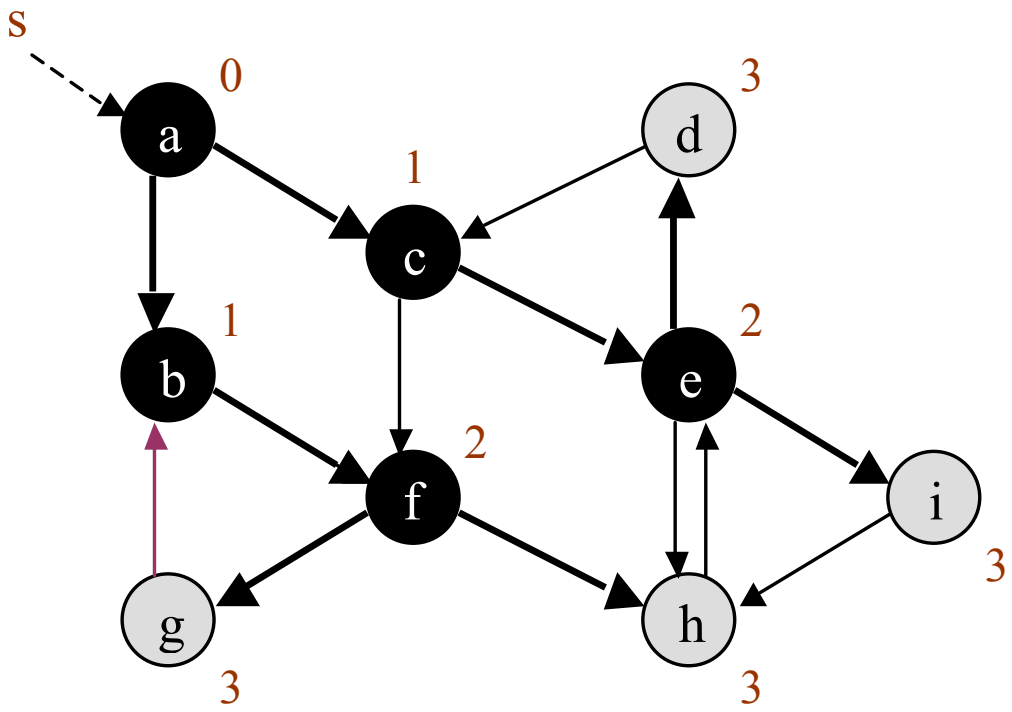

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |

# Breadth-First Search




| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |

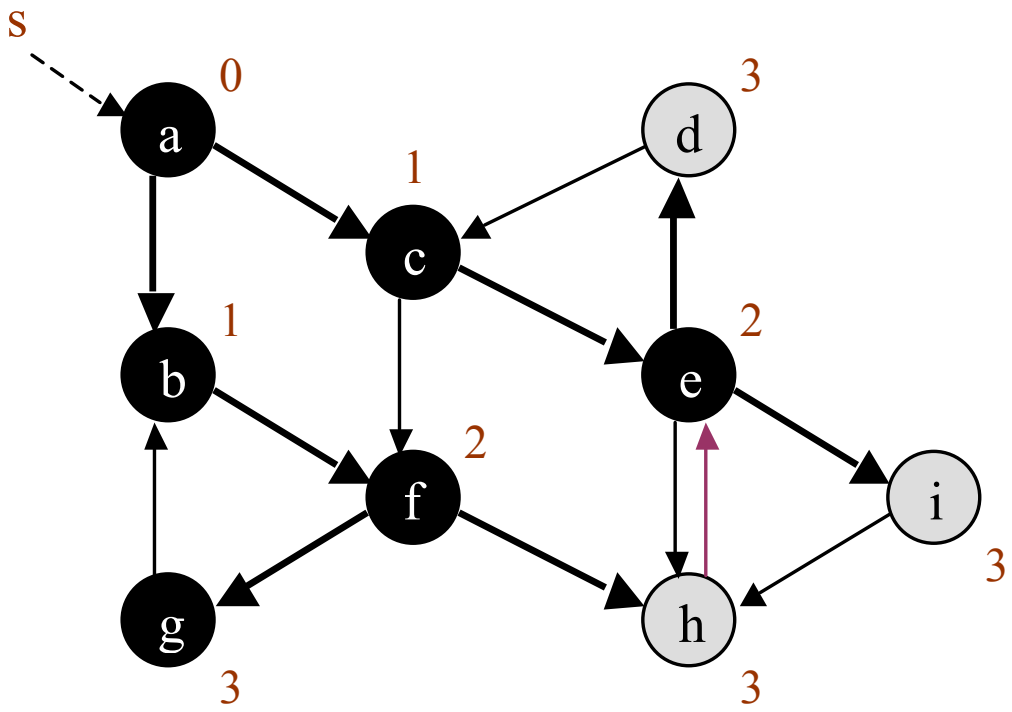
# Breadth-First Search




| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |
| ⟨ g,h,d,i⟩ | e |

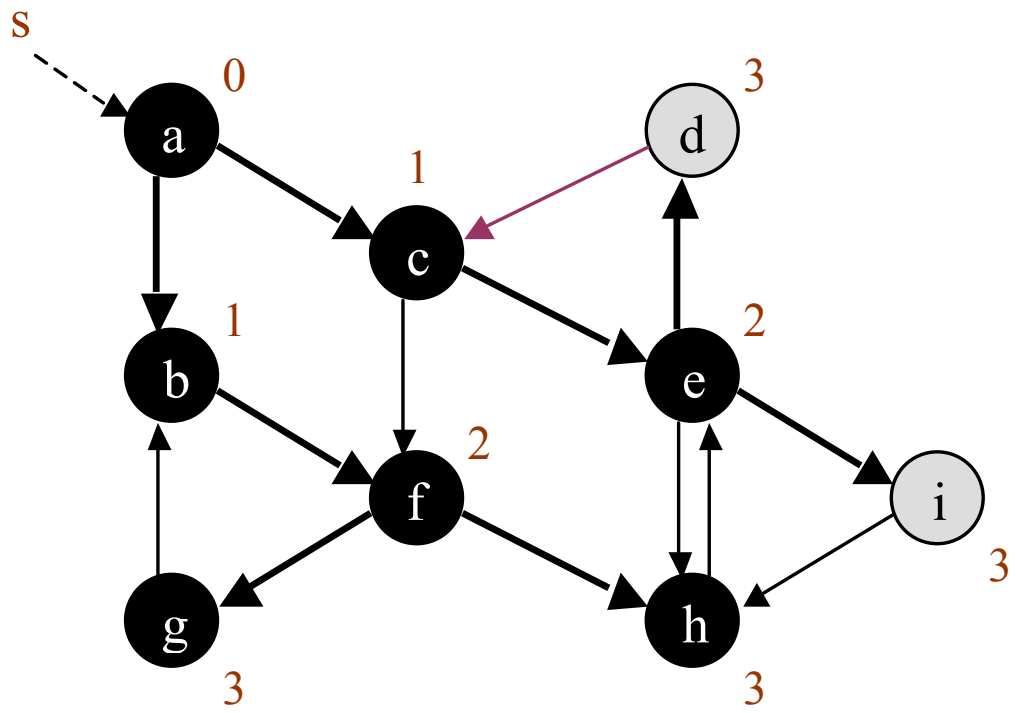
all distances are filled in after processing *e*

# Breadth-First Search



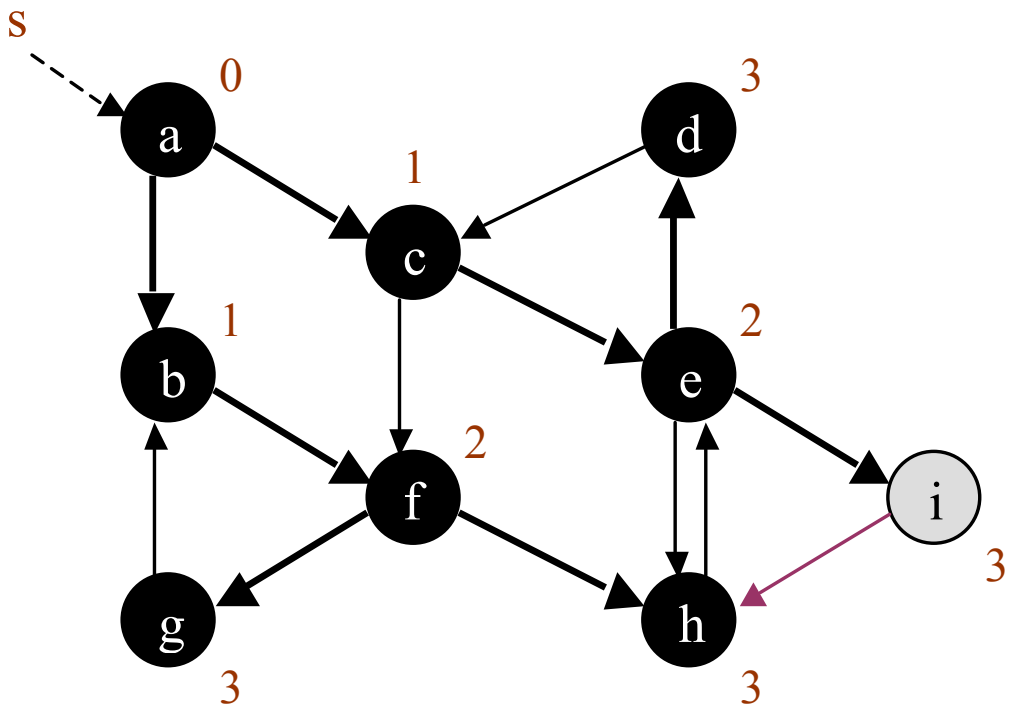

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |
| ⟨ h,d,i⟩ | g |

# Breadth-First Search



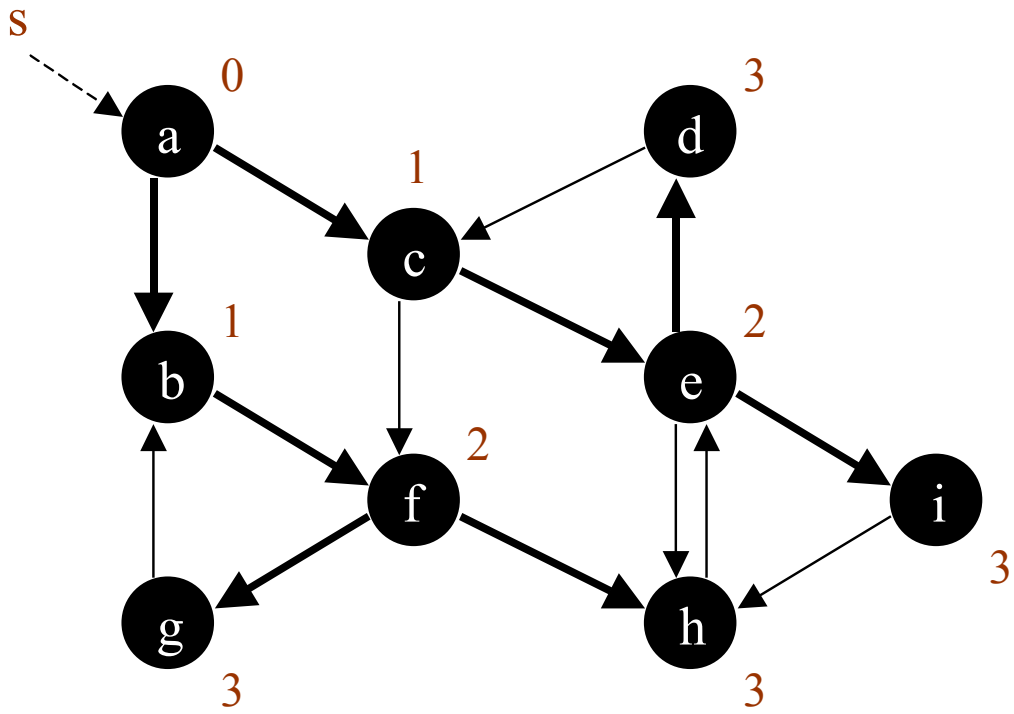

| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |
| ⟨ d,i⟩ | h |

# Breadth-First Search




| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |
| ⟨ i⟩ | d |

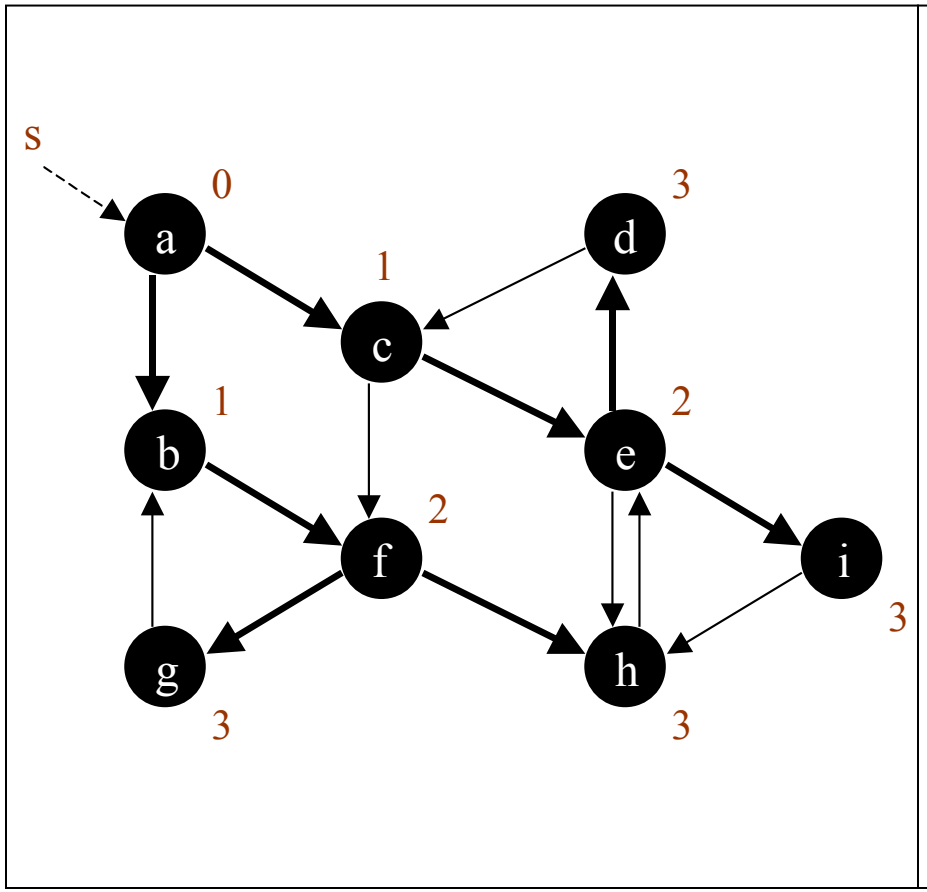
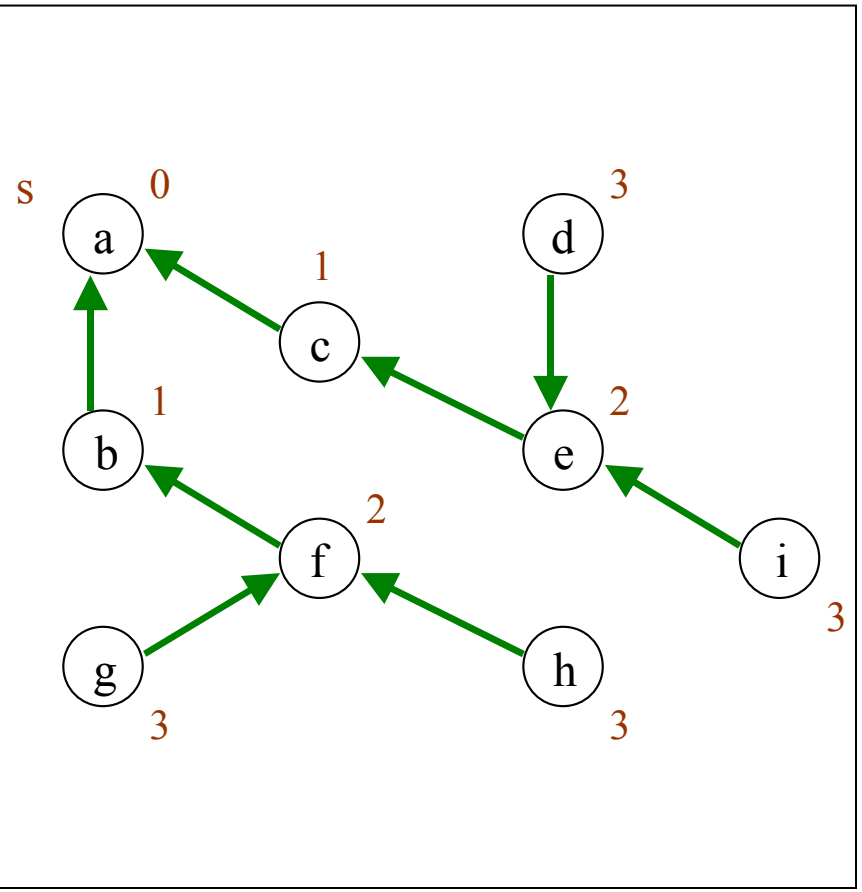
# Breadth-First Search




| FIFO queue *Q* | just after processing vertex |
|---|---|
| ⟨a⟩ | - |
| ⟨ b,c⟩ | a |
| ⟨ c,f⟩ | b |
| ⟨ f,e⟩ | c |
| ⟨ e,g,h⟩ | f |
| ⟨ ⟩ | i |

algorithm terminates: all vertices are processed

# Breadth-First Tree Generated by BFS

| $BFS(G, a)$ terminated | BFT generated by $BFS(G, a)$ |
| --- | --- |
|  |  |

# BFS – Running time analysis

Data structures for the graph ADT: adjacency list

- Each vertex is enqueued at most once, and hence dequeued at most once.
- The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $\boldsymbol{O(V)}$.
- The procedure scans the adjacency list of each vertex only when the vertex is dequeued, so it scans each adjacency list at most once.
- Total lengths of the adjacency lists is $O(E)$, so the total time spent in scanning adjacency lists is $\boldsymbol{O(E)}$.
- The overhead for initialization is $O(V)$.

Thus the total running time of the BFS procedure is $\boldsymbol{O(V+E)}$.

# BFS – Correctness

**Claim 1.** For each vertex $v \in V$, the value $v.d$ computed by BFS satisfies

$$v.d \geq \delta(s, v)$$

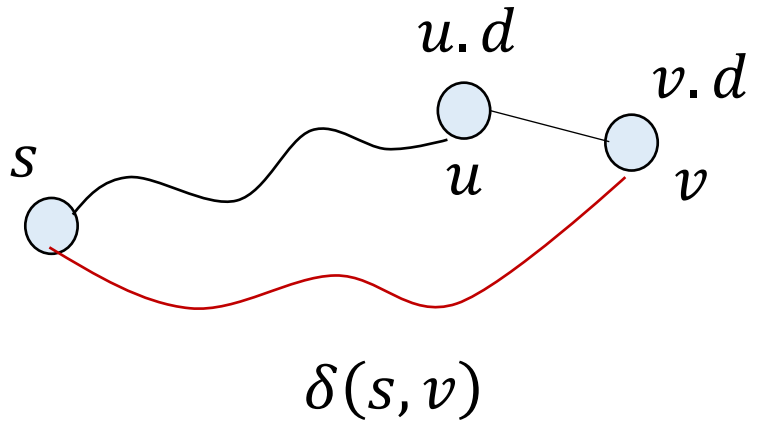
**Proof.** By induction on the number of enqueue operations.

$s.d = 0 = \delta(s, s)$

$v.d = u.d + 1$

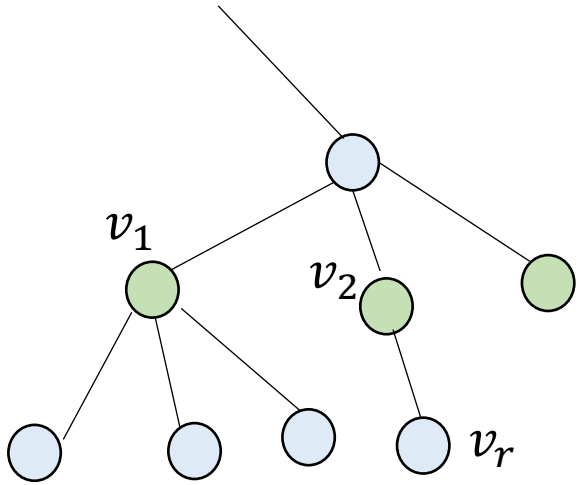
$\geq \delta(s, u) + 1$

$\geq \delta(s, v)$



# BFS – Correctness

**Claim 2.** Suppose that during BFS execution, the queue $Q$ contains the vertices $< v_1, v_2, \dots, v_r >$, where $v_1$ is the head of $Q$ and $v_r$ is the tail, then $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1,2, \dots, r-1$.




**Proof.** By induction on the number of enqueue operations.

Base: when the queue contains only s the lemma holds.

- $Q: < v_1, v_2, \dots, v_r >$
  - $v_1$ is dequeued, $v_2$ becomes the new head.
  - a new vertex is enqueued.

  In both cases $v_i.d$ is ascending and the difference of the depths cannot be more than one. (For more formal proof see CLRS page 599)

# BFS – Correctness

**Main result.**

- BFS discovers every vertex $v \in V$ that is reachable from $s$.
- $v.d = \delta(s, v)$

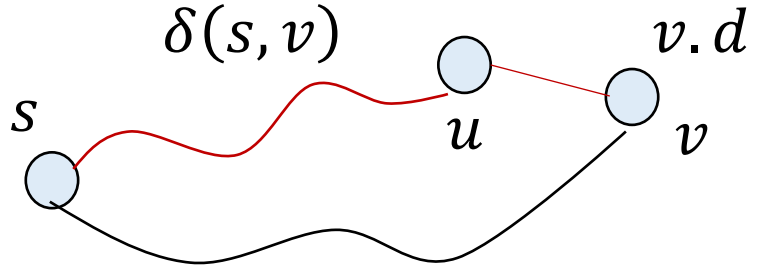
**Proof.** By contradiction

Let $v$ be the vertex with minimum $\delta(s, v)$ which $v.d > \delta(s, v)$

$v$ must be reachable from $s$, otherwise $\delta(s, v) = \infty$.

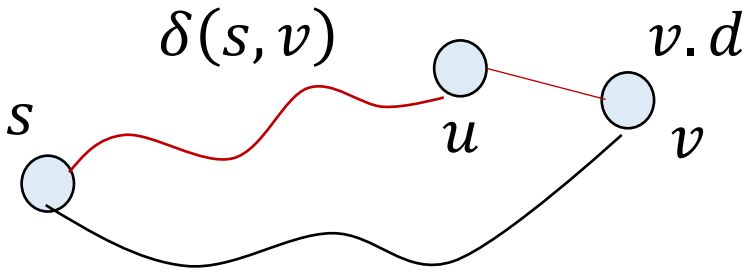
$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$

$\rightarrow \quad v.d > u.d + 1$



# BFS – Correctness (Cont'd)

$v.d > u.d + 1$




Three cases when $u$ is dequeued

- $v$ is white: it is not in the queue, then $v.d = u.d + 1$
- $v$ is gray: it is in the queue $< u, v, \ldots >$

  It means that there was a vertex $z$ which was removed once v is enqueued. ($z$ is $v.\pi$).

  $< z, \ldots, u, v, \ldots >$ So, by claim 2 $z.d = v.d - 1 \leq u.d \rightarrow v.d \leq u.d + 1$
- $v$ is black (it has been removed from the queue). $< v, u \ \ldots >$

  According to claim 2, $v.d \leq u.d$

# Questions