# CSE 431/531: Algorithm Analysis and Design (Spring 2021)

# Graph Basics

Lecturer: Shi Li

*Department of Computer Science and Engineering*
*University at Buffalo*

# Outline

1. Graphs
2. Connectivity and Graph Traversal
   - Testing Bipartiteness
3. Topological Ordering

# Examples of Graphs

Figure: Road Networks

Figure: Internet

Figure: Social Networks

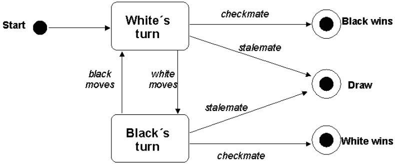
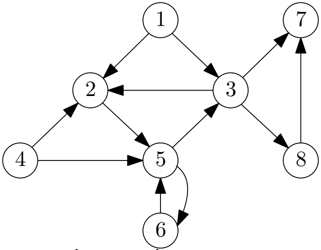



Figure: Transition Graphs

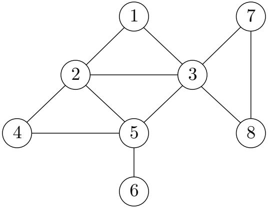
# (Undirected) Graph $G = (V, E)$




- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - (undirected) graphs: relationship is symmetric, $E$ contains subsets of size 2
  - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$

# Abuse of Notations

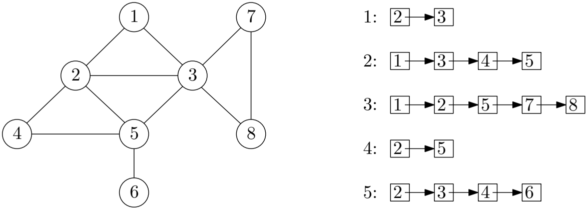
- For (undirected) graphs, we often use $(i, j)$ to denote the set $\{i, j\}$.
- We call $(i, j)$ an unordered pair; in this case $(i, j) = (j, i)$.




- $E = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6), (7,8)\}$

- Social Network : Undirected
- Transition Graph : Directed
- Road Network : Directed or Undirected
- Internet : Directed or Undirected

# Representation of Graphs




- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected
- Linked lists
  - For every vertex $v$, there is a linked list containing all neighbours of $v$.

# Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \le m \le n(n-1)/2$
- $d_v$: number of neighbors of $v$

| | Matrix | Linked Lists |
|---|---|---|
| memory usage | $O(n^2)$ | $O(m)$ |
| time to check $(u, v) \in E$ | $O(1)$ | $O(d_u)$ |
| time to list all neighbours of $v$ | $O(n)$ | $O(d_v)$ |

# Outline

## Connectivity Problem

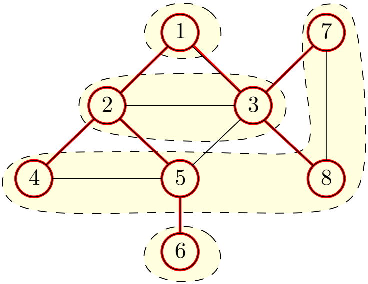
**Input:** graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

# Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$



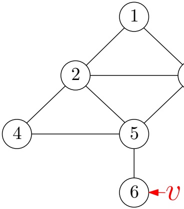
# Implementing BFS using a Queue

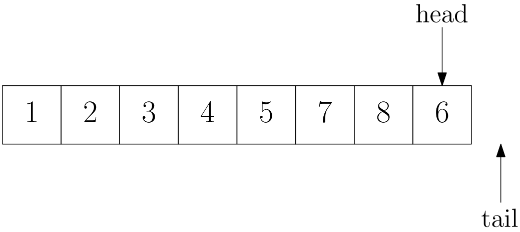
**BFS($s$)**

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as "visited" and all other vertices as "unvisited"
**while** head $\geq$ tail **do**
    $v \leftarrow queue[tail], tail \leftarrow tail + 1$
    **for** all neighbours $u$ of $v$ **do**
        **if** $u$ is "unvisited" **then**
            $head \leftarrow head + 1, queue[head] = u$
            mark $u$ as "visited"

- Running time: $O(n + m)$.

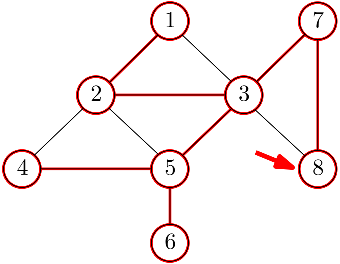
# Example of BFS via Queue



# Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



# Implementing DFS using Recurrsion

**DFS($s$)**

```
mark all vertices as "unvisited"
recursive-DFS(s)
```

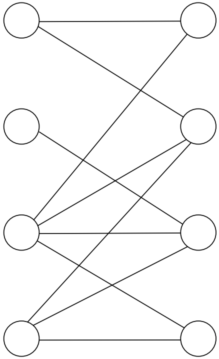
**recursive-DFS($v$)**

```
mark v as "visited"
for all neighbours u of v do
    if u is unvisited then recursive-DFS(u)
```

# Outline

# Testing Bipartiteness: Applications of BFS

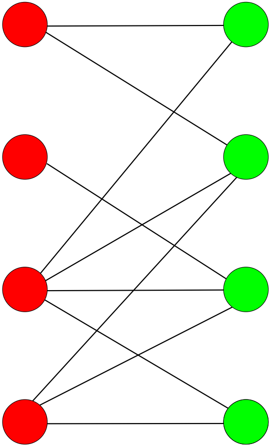
**Def.** A graph $G = (V, E)$ is a bipartite graph if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$.

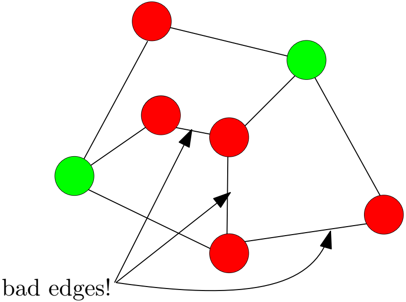


# Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- $\cdots$
- Report "not a bipartite graph" if contradiction was found
- If $G$ contains multiple connected components, repeat above algorithm for each component

# Test Bipartiteness





# Testing Bipartiteness using BFS

**BFS($s$)**

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
**while** head $\geq$ tail **do**
  $v \leftarrow queue[tail], tail \leftarrow tail + 1$
  **for** all neighbours $u$ of $v$ **do**
    **if** $u$ is "unvisited" **then**
      $head \leftarrow head + 1, queue[head] = u$
      mark $u$ as "visited"
      $color[u] \leftarrow 1 - color[v]$
    **else if** $color[u] = color[v]$ **then**
      print("$G$ is not bipartite") and exit

# Testing Bipartiteness using BFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness(v)
print("G is bipartite")
```

**Obs.** Running time of algorithm $= O(n+m)$

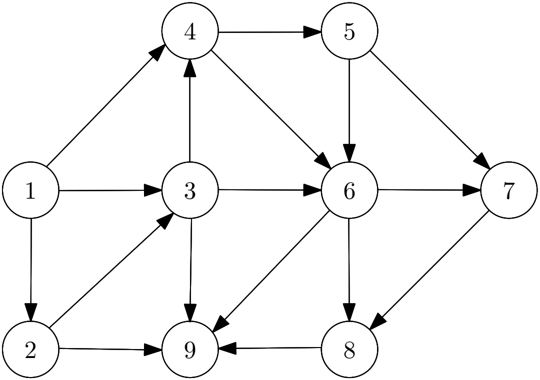
# Outline

## Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

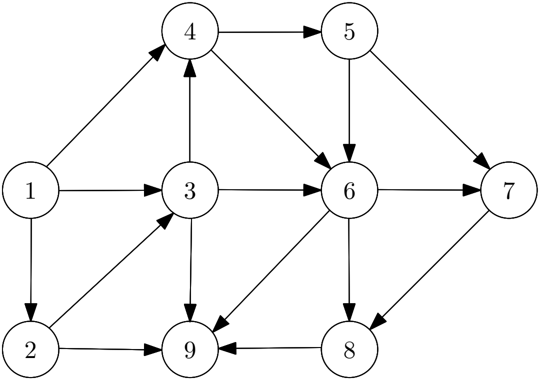
**Output:** 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



# Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



# Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

**Q:** How to make the algorithm as efficient as possible?

**A:**

- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$

## topological-sort($G$)

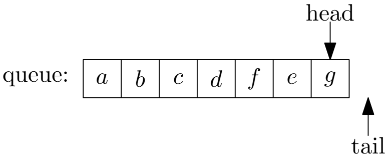
```
let d_v ← 0 for every v ∈ V
for every v ∈ V do
    for every u such that (v, u) ∈ E do
        d_u ← d_u + 1
S ← {v : d_v = 0}, i ← 0
while S ≠ ∅ do
    v ← arbitrary vertex in S, S ← S \ {v}
    i ← i + 1, π(v) ← i
    for every u such that (v, u) ∈ E do
        d_u ← d_u − 1
        if d_u = 0 then add u to S
if i < n then output "not a DAG"
```

- $S$ can be represented using a queue or a stack
- Running time $= O(n + m)$

# $S$ as a Queue or a Stack

| DS | Queue | Stack |
|---|---|---|
| Initialization | $head \leftarrow 0$, $tail \leftarrow 1$ | $top \leftarrow 0$ |
| Non-Empty? | $head \geq tail$ | $top > 0$ |
| Add$(v)$ | $head \leftarrow head + 1$<br>$S[head] \leftarrow v$ | $top \leftarrow top + 1$<br>$S[top] \leftarrow v$ |
| Retrieve $v$ | $v \leftarrow S[tail]$<br>$tail \leftarrow tail + 1$ | $v \leftarrow S[top]$<br>$top \leftarrow top - 1$ |

# Example





| | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|
| degree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |