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 $L_j$
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 $L_j$
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 $L_j$
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 $L_j$
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($s$)**

1: $\text{head} \leftarrow 1$, $\text{tail} \leftarrow 1$, $\text{queue}[1] \leftarrow s$
2: mark $s$ as “visited” and all other vertices as “unvisited”
3: **while** $\text{head} \leq \text{tail}$ **do**
4: $v \leftarrow \text{queue}[\text{head}]$, $\text{head} \leftarrow \text{head} + 1$
5: **for** all neighbors $u$ of $v$ **do**
6: $\quad$ **if** $u$ is “unvisited” **then**
7: $\quad$ $\text{tail} \leftarrow \text{tail} + 1$, $\text{queue}[\text{tail}] = u$
8: $\quad$ mark $u$ as “visited”

- Running time: $O(n + m)$. 
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

```
Example of BFS via Queue
1
2
3
4
5
6
7
8
```

![Graph and Queue Diagram](image)

- **Graph Diagram**: Starting node `v` is connected to nodes 2, 3, 4, 5, 6, 7, and 8.
- **Queue Diagram**: Queue shows elements 1, 2, and 3, with `head` and `tail` markers.

This image illustrates how Breadth-First Search (BFS) is implemented using a queue, with nodes visited in a breadth-first manner.
Example of BFS via Queue

The diagram illustrates a breadth-first search (BFS) using a queue. The vertices 1, 2, 3, 4, 5, 6, 7, and 8 are arranged in a graph. The queue is depicted with elements 1, 2, 3, and 4, indicating the order of vertices visited in a BFS.

The vertex 2 is marked with an arrow, indicating the direction of traversal. The queue is shown with an arrow pointing to the tail and another arrow pointing to the head, indicating the queue's direction and order of elements.
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

1 2 3 4 5 7 8 6

14/38

tail
head

v
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
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
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
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
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
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
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
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
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
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
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
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 Recursion

<table>
<thead>
<tr>
<th><strong>DFS</strong>( (s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. mark all vertices as “unvisited”</td>
</tr>
<tr>
<td>2. recursive-DFS( (s) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>recursive-DFS</strong>( (v) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. mark ( v ) as “visited”</td>
</tr>
<tr>
<td>2. for all neighbors ( u ) of ( v ) do</td>
</tr>
<tr>
<td>3. if ( u ) is unvisited then recursive-DFS( (u) )</td>
</tr>
</tbody>
</table>
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
**Def.** An undirected graph \( G = (V, E) \) is a **path** if the vertices can be listed in an order \( \{v_1, v_2, \ldots, v_n\} \) such that the edges are the \( \{v_i, v_{i+1}\} \) where \( i = 1, 2, \ldots, n - 1 \).

- Path graphs are connected graphs.
**Def.** An undirected graph \( G = (V, E) \) is a **cycle** if its vertices can be listed in an order \( v_1, v_2, \ldots, v_n \) such that the edges are the \( \{v_i, v_{i+1}\} \) where \( i = 1, 2, \ldots, n - 1 \), plus the edge \( \{v_n, v_1\} \).

- The degree of all vertices is 2.
**Def.** An undirected graph $G = (V, E)$ is a **tree** if any two vertices are connected by exactly one path. Or the graph is a connected acyclic graph.

- Most important type of special graphs: most computational problems are easier to solve on trees or lines.
**Def.** An undirected graph $G = (V, E)$ is a complete graph if each pair of vertices is joined by an edge.

- A complete graph contains all possible edges.
Def. An undirected graph $G = (V, E)$ is a planar graph if its vertices and edges can be drawn in a plane such that no two of the edges intersect.

- Most computational problems have good solutions in a planar graph.
Def. A directed graph $G = (V, E)$ is a directed acyclic graph if it is a directed graph with no directed cycles.

- DAG is equivalent to a partial ordering of nodes.
Def. An undirected 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 $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$. 
Outline

1. Graphs
2. Connectivity and Graph Traversal
   - Types of Graphs
3. Bipartite Graphs
   - Testing Bipartiteness
4. Topological Ordering
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 $(u, v) \in E$, either $u \in L$, $v \in R$ or $v \in L$, $u \in R$. 
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$

If $G$ contains multiple connected components, repeat above algorithm for each component.
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 \( L \)

Report “not a bipartite graph” if contradiction was found

If \( G \) contains multiple connected components, repeat above algorithm for each component
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 $L$
- ...
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 $L$
- ... 
- Report “not a bipartite graph” if contradiction was found
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 \( L \)
- \( \ldots \)
- Report “not a bipartite graph” if contradiction was found
- If \( G \) contains multiple connected components, repeat above algorithm for each component
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness

bad edges!
Testing Bipartiteness using BFS

BFS(s)

1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as “visited” and all other vertices as “unvisited”
3: while head ≤ tail do
4:   v ← queue[head], head ← head + 1
5:   for all neighbors u of v do
6:     if u is “unvisited” then
7:        tail ← tail + 1, queue[tail] = u
8:     mark u as “visited”
Testing Bipartiteness using BFS

test-bipartiteness(s)

1: head ← 1, tail ← 1, queue[1] ← s 
2: mark s as “visited” and all other vertices as “unvisited”
3: color[s] ← 0
4: while head ≤ tail do
5: v ← queue[head], head ← head + 1
6: for all neighbors u of v do
7: if u is “unvisited” then
8: tail ← tail + 1, queue[tail] = u
9: mark u as “visited”
10: color[u] ← 1 − color[v]
11: else if color[u] = color[v] then
12: print(“G is not bipartite”) and exit