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”
test-bipartiteness(\(s\))

1: \(head \leftarrow 1, \, tail \leftarrow 1, \, queue[1] \leftarrow s\)
2: mark \(s\) as “visited” and all other vertices as “unvisited”
3: \(color[s] \leftarrow 0\)
4: \(\text{while } head \leq tail \text{ do}\)
5: \(v \leftarrow queue[head], \, head \leftarrow head + 1\)
6: \(\text{for all neighbors } u \text{ of } v \text{ do}\)
7: \(\text{if } u \text{ is “unvisited” then}\)
8: \(\text{tail } \leftarrow \text{tail } + 1, \, queue[\text{tail}] = u\)
9: \(\text{mark } u \text{ as “visited”}\)
10: \(\text{else if } color[u] = color[v] \text{ then}\)
11: \(\text{print(“G is not bipartite”) and exit}\)
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3: \hspace{1em} if $v$ is “unvisited” then
4: \hspace{2em} test-bipartiteness($v$)
5: print(“$G$ is bipartite”)
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3: if $v$ is “unvisited” then
4: test-bipartiteness($v$)
5: print(“$G$ is bipartite”)

Obs. Running time of algorithm $= O(n + m)$
Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

1: mark all vertices as “unvisited”
2: recursive-test-DFS(s)

recursive-test-DFS(v)

1: mark v as “visited”
2: for all neighbors u of v do
3: if u is unvisited then , recursive-test-DFS(u)
Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)
1: mark all vertices as “unvisited”
2: color[s] ← 0
3: recursive-test-DFS(s)

recursive-test-DFS(v)
1: mark v as “visited”
2: for all neighbors u of v do
3: if u is unvisited then
4: color[u] ← 1 − color[v], recursive-test-DFS(u)
5: else if color[u] = color[v] then
6: print(“G is not bipartite”) and exit
Testing Bipartiteness using DFS

1: mark all vertices as “unvisited”
2: for each vertex \( v \in V \) do
3:   if \( v \) is “unvisited” then
4:     test-bipartiteness-DFS(\( v \))
5: print(“\( G \) is bipartite”)
Testing Bipartiteness using DFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3: \hspace{1em} if $v$ is “unvisited” then
4: \hspace{2em} test-bipartiteness-DFS($v$)
5: \hspace{1em} print(“$G$ is bipartite”)

Obs. Running time of algorithm $= O(n + m)$
**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 $v \in L, u \in R$. 
**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 $v \in L, u \in R$.

**Obs.** Bipartite graph may contain cycles.
**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 $v \in L, u \in R$.

**Obs.** Bipartite graph may contain cycles.

**Obs.** If a graph is a tree, then it is also a bipartite graph.
BFS and DFS

**Obs.** BFS and DFS naturally induce a tree.
**Obs.** BFS and DFS naturally induce a tree.

**Obs.** If $G$ is a tree, then BFS tree $=$ DFS tree.
BFS and DFS

**Obs.** BFS and DFS naturally induce a tree.

**Obs.** If $G$ is a tree, then BFS tree = DFS tree.

**Obs.** If BFS tree = DFS tree, then $G$ is a tree.
BFS and DFS

Observation: If BFS tree $\equiv$ DFS tree, then $G$ is a tree.

- True: simple, undirected graph
- Not True: directed graph
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
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 Problem

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

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

- if \((u, v) \in E\) then \( \pi(u) < \pi(v) \)
**Algorithm:** each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
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.
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.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
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.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
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.
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.
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.
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.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
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?
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\((G)\)

1: let \(d_v \leftarrow 0\) for every \(v \in V\)
2: 
   for every \(v \in V\) do
3:     for every \(u\) such that \((v, u) \in E\) do
4:       \(d_u \leftarrow d_u + 1\)
5:   \(S \leftarrow \{v : d_v = 0\}, i \leftarrow 0\)
6:   while \(S \neq \emptyset\) do
7:     \(v \leftarrow\) arbitrary vertex in \(S, S \leftarrow S \setminus \{v\}\)
8:     \(i \leftarrow i + 1, \pi(v) \leftarrow i\)
9:   for every \(u\) such that \((v, u) \in E\) do
10:      \(d_u \leftarrow d_u - 1\)
11:      if \(d_u = 0\) then add \(u\) to \(S\)
12:     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

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$head \leftarrow 1$, $tail \leftarrow 0$</td>
<td>$top \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$head \leq tail$</td>
<td>$top &gt; 0$</td>
</tr>
<tr>
<td>Add($v$)</td>
<td>$tail \leftarrow tail + 1$</td>
<td>$top \leftarrow top + 1$</td>
</tr>
<tr>
<td></td>
<td>$S[tail] \leftarrow v$</td>
<td>$S[top] \leftarrow v$</td>
</tr>
<tr>
<td>Retrieve $v$</td>
<td>$v \leftarrow S[head]$</td>
<td>$v \leftarrow S[top]$</td>
</tr>
<tr>
<td></td>
<td>$head \leftarrow head + 1$</td>
<td>$top \leftarrow top - 1$</td>
</tr>
</tbody>
</table>
Example

Queue:

\[
\begin{array}{c}
\text{head} \\
\text{tail}
\end{array}
\]

degree

\[
\begin{array}{ccccccc}
a & b & c & d & e & f & g \\
0 & 1 & 1 & 1 & 2 & 1 & 3
\end{array}
\]
Example

A directed graph with nodes labeled $a$, $b$, $c$, $d$, $e$, $f$, and $g$. The graph shows the following edges:

- $a$ to $b$
- $a$ to $c$
- $b$ to $d$
- $b$ to $e$
- $c$ to $f$
- $c$ to $g$
- $d$ to $f$
- $d$ to $g$
- $e$ to $g$

A queue with elements $a$, $b$, $c$, $d$, $e$, $f$, and $g$, labeled as follows:

- Head: $a$
- Tail: $g$

A table showing the degree of each node:

<table>
<thead>
<tr>
<th>Degree</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$2$</td>
<td>$1$</td>
<td></td>
<td>$3$</td>
</tr>
</tbody>
</table>
Example

 Queue: 

\[
\begin{array}{c}
tail \\
head \\
\end{array}
\]

\[
\begin{array}{cccccccc}
|\text{queue:}| & a & b & c & d & e & f & g \\
|degree  | & 0 & 0 & 0 & 1 & 2 & 1 & 3 \\
\end{array}
\]

Diagram:

- Nodes: e, b, c, d, f, g
- Edges: e→b, e→f, b→g, c→d, d→g, f→g

Degree:
- e: 0
- b: 0
- c: 0
- d: 1
- f: 1
- g: 3
Example
Example

![Diagram of a graph with nodes b, c, d, e, f, and g connected by arrows.]

A queue is shown with the following elements: a, b, c, d, e, f, and g. The queue has a head and a tail. The degree of each node is as follows:

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

\[ a \quad b \quad c \quad d \quad f \quad g \]

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

tail

head
Example

Queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

tail

head

degree

Graph:

- Nodes: e, d, f, c, g
- Edges: e→d, d→f, f→g, c→d, c→f, e→g
Example

**Queue:**

```
queue: a  b  c
```

**Degree:**

```
degree | a | b | c | d | e | f | g
--------|---|---|---|---|---|---|---
       | 0 | 0 | 0 | 0 | 1 | 0 | 3
```
Example

Queue:

$$\begin{array}{cccccc}
\text{head} & \text{tail} \\
\downarrow & \\
\text{queue:} & a & b & c & d & f & \\
\downarrow & \\
\text{degree} & 0 & 0 & 0 & 0 & 1 & 0 & 3
\end{array}$$

Diagram:

- $e$ to $d$
- $d$ to $f$
- $g$ to $d$
- $e$ to $g$
- $d$ to $g$
- $f$ to $g$

Graph representation:

- Node $e$ is connected to $d$, $g$.
- Node $d$ is connected to $e$, $f$, $g$.
- Node $f$ is connected to $d$, $g$.
- Node $g$ is connected to $e$, $d$, $f$. 
Example

Queue:

\[
\begin{array}{ccccccc}
\text{head} & & & \text{tail} & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 3 \\
\end{array}
\]

Degree:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th></th>
</tr>
</thead>
</table>

degree  

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue: \[\begin{array}{cccccc}
a & b & c & d & f & e \\
\end{array}\]

degree: \[
\begin{array}{cccccccc}
a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 0 & 0 & 2 \\
\end{array}
\]
Example

queue: \[ \begin{array}{c|c|c|c|c|c|c} a & b & c & d & f & e \\ \hline \text{degree} & 0 & 0 & 0 & 0 & 0 & 2 \end{array} \]
Example

queue:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>e</th>
</tr>
</thead>
</table>

degree

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

Queue: $a$, $b$, $c$, $d$, $f$, $e$

Degree:

| degree | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
Example

queue: \[a \ b \ c \ d \ f \ e\]

g

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

tail

head
Example

Queue: [a, b, c, d, f, e, g]

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

queue: \begin{array}{cccccccc}
a & b & c & d & f & e & g \\
\end{array}

\begin{array}{cccccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Types of Graphs

3. Bipartite Graphs
   - Testing Bipartiteness

4. Topological Ordering
   - Applications: Word Ladder
**Def.** Word: A string formed by letters.

**Def.** Adjacency words: Word $A$ and $B$ are adjacent if they differ in exactly one letter.

e.g. word and work; tell and tall; askbe and askee.
Def. Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.
**Def.** Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.

- The objective is to make the change in the smallest number of steps, with each step involving changing a **single letter** of the word to create a new valid word.
Word Ladder Problem

**Input:** Two words $S$ and $T$, a list of words $A = \{W_1, W_2, ..., W_k\}$.

**Output:** “The smallest word ladder” if we can change $S$ to $T$ by moving between adjacency words in $A \cup \{S, T\}$; Otherwise, “No word ladder”.

Example:

- $S=“a\ e\ f\ g\ h”,\ T=“d\ l\ m\ i\ h”$
- $W_1=“a\ e\ f\ i\ h”,\ W_2=“a\ e\ m\ g\ h”,\ W_3=“d\ l\ f\ i\ h”$
- $W_4=“s\ e\ f\ i\ h”,\ W_5=“a\ d\ f\ g\ h”,\ W_6=“d\ e\ m\ i\ h”$
- $W_7=“d\ e\ f\ i\ h”,\ W_8=“d\ e\ m\ g\ h”,\ W_9=“s\ e\ m\ i\ h”$
Example:

- \( S = \text{"a e f g h"}, \ T = \text{"d l m i h"} \)
- \( W_1 = \text{"a e f i h"}, \ W_2 = \text{"a e m g h"}, \ W_3 = \text{"d l f i h"} \)
- \( W_4 = \text{"s e f i h"}, \ W_5 = \text{"a d f g h"}, \ W_6 = \text{"d e m i h"} \)
- \( W_7 = \text{"d e f i h"}, \ W_8 = \text{"d e m g h"}, \ W_9 = \text{"s e m i h"} \)

Each vertex corresponds to a word.

Two vertices are adjacent if the corresponding words are adjacent.
Each vertex corresponds to a word.

Two vertices are adjacent if the corresponding words are adjacent.

Hints: Given vertex \( v \), check its nearest neighbor.