COP 4531
Complexity & Analysis of Data Structures & Algorithms

Overview of Graphs
Breadth First Search, and Depth First Search

Thanks to several people who contributed to these slides including Piyush Kumar and S-H Poon (BFS detailed example) and the text authors.
Outline

• What are Graphs?
• Terminology
• Representation (Adjacency matrices and Linked lists)
• Searching
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
Graphs

- A graph \( G = (V,E) \) is composed of:
  - \( V \): set of vertices
  - \( E \subseteq V \times V \): set of edges connecting the vertices

- An edge \( e = (u,v) \) is an ordered or unordered pair of vertices
  - Directed graphs ordered: \((u, v)\)
    - Edge incident from or leaves \( u \)
    - Edge incident to or enters \( v \)
  - Undirected graphs unordered: \( \{u, v\} \)
Graphs with edge weights

- The initial graph definition defines whether an edge exists or not.
- Often we want to associate a weight \( w(e) \) to the edge:
  \[
  w: E \rightarrow \mathbb{R}
  \]
- The weight can represent distance, time, etc.
Directed graphs

source

sink

(b) Graph2 is a directed graph.

V(Graph2) = {1, 3, 5, 7, 9, 11}
E(Graph2) = {{1, 3}, (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}
An undirected graph
A more complicated undirected graph
## Some Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
**Terminology**

- **a is adjacent to b iff** \((a,b) \in E\).
- **degree** \((a) =\) number of adjacent vertices (Self loop counted twice)
- **Self Loop**: \((a,a)\)
- **Could also define parallel edges**:
  - \(E = \{ \ldots(a,b), (a,b)\ldots\}\) is a multiset
• A Simple Graph is a graph with no self loops or parallel edges.
• Incidence: v is incident to e if v is an end vertex of e.
Degree of a vertex

- The number of edges incident on the vertex in an undirected graph
- For directed graphs we have out-degree and in-degree (edges leaving or entering the vertex). The degree is in-degree + out-degree
- Isolated vertex has degree 0
- Max degree vertex, min degree vertex
Example

- Max Degree = 4. Isolated vertices = 1.
- $|V| = 8$, $|E| = 8$
  
  Sum of degrees = 16 =
  
  $2|E| = \sum_{v \in V} \text{degree (v)}$
- Handshaking theorem
QUESTION

• How many edges are there in a graph with 100 vertices each of degree 4?
QUESTION

• How many edges are there in a graph with 100 vertices each of degree 4?
  - Total degree sum = 400 = 2 |E|
  - 200 edges by the handshaking theorem.
Handshaking Corollary

The number of vertices with odd degree is always even.

**Proof:** Let $V_1$ and $V_2$ be the set of vertices of even and odd degrees, respectively (Hence $V_1 \cap V_2 = \emptyset$, and $V_1 \cup V_2 = V$).

- Now we know that
  \[2|E| = \sum_{v \in V} \text{degree}(v) = \sum_{v \in V_1} \text{degree}(v) + \sum_{v \in V_2} \text{degree}(v)\]

- Since degree$(v)$ is odd for all $v \in V_2$, $|V_2|$ must be even.
Representation

• Two ways
  - Adjacency List
    • (as a linked list for each node in the graph to represent the edges)
  - Adjacency Matrix
    • (as a boolean matrix)
Representing Graphs

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>1, 4</td>
</tr>
<tr>
<td>3</td>
<td>1, 4</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Vertex</th>
<th>Terminal Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>
adjacency list

1
2
3
4

2
1
4
3
4
2
3
adjacency matrix

\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{pmatrix}
, \quad
\begin{pmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
\end{pmatrix}
, \quad
\begin{pmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{pmatrix}
\]
Another example (directed graph)

1. Adjacency Matrix

2. Adjacency List
Another example
(undirected graph)
AL Vs AM

- AL: Takes $O(|V| + |E|)$ space
- AM: Takes $O(|V|^*|V|)$ space

Question: How much time does it take to find out if $(v_i,v_j)$ belongs to $E$?

- AM?
- AL?
AL Vs AM

- AL: Takes $O(|V| + |E|)$ space
- AM: Takes $O(|V|^*|V|)$ space
- Question: How much time does it take to find out if $(v_i, v_j)$ belongs to $E$?
  - AM : $O(1)$
  - AL : $O(|V|)$ in the worst case.
Connectivity

• s-t connectivity problem. Given two node s and t, is there a path between s and t?

• s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

• Applications.
  - Maze traversal.
  - Kevin Bacon number / Erdos number
  - Fewest number of hops in a communication network.
  - Friendster.
BFS/DFS

• Breadth-first search (BFS) and depth-first search (DFS) are two distinct orders in which to visit the vertices and edges of a graph.
• BFS: radiates out from a root to visit vertices in order of their distance from the root. Thus closer nodes get visited first.
Breadth first search

- Question: Given $G$ in AM form, how do we say if there is a path between nodes $a$ and $b$?

- Note: Using AM or AL its easy to answer if there is an edge $(a,b)$ in the graph, but not path questions. This is one of the reasons to learn BFS/DFS.
BFS

- A Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree.
Algorithm \(BFS(s)\)

**Input:** \(s\) is the source vertex

**Output:** Mark all vertices that can be visited from \(s\).

1. for each vertex \(v\)
2. \hspace{1cm} do \(\text{flag}[v] := \text{false};\)
3. \(Q = \text{empty queue};\)
4. \(\text{flag}[s] := \text{true};\)
5. \(\text{enqueue}(Q, s);\)
6. while \(Q\) is not empty
7. \hspace{1cm} do \(v := \text{dequeue}(Q);\)
8. \hspace{1cm} for each \(w\) adjacent to \(v\)
9. \hspace{2cm} do if \(\text{flag}[w] = \text{false}\)
10. \hspace{3cm} then \(\text{flag}[w] := \text{true};\)
11. \hspace{3cm} \text{enqueue}(Q, w)\)
Example

\[ Q = \{ \} \]

Initialize \( Q \) to be empty
Example

\[ Q = \{ 2 \} \]

Place source 2 on the queue.
Example

Q = \{2\} \rightarrow \{8, 1, 4\}

Dequeue 2.
Place all unvisited neighbors of 2 on the queue
Example

Q = \{ 8, 1, 4 \} \rightarrow \{ 1, 4, 0, 9 \}

Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!
Example

\[ Q = \{ 1, 4, 0, 9 \} \rightarrow \{ 4, 0, 9, 3, 7 \} \]

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven’t been visited yet.
Example

\( Q = \{ 4, 0, 9, 3, 7 \} \rightarrow \{ 0, 9, 3, 7 \} \)

Dequeue 4.
-- 4 has no unvisited neighbors!
Example

Q = \{ 0, 9, 3, 7 \} \rightarrow \{ 9, 3, 7 \}

Dequeue 0.
-- 0 has no unvisited neighbors!
Example

\[ Q = \{ 9, 3, 7 \} \rightarrow \{ 3, 7 \} \]

Deque 9.
-- 9 has no unvisited neighbors!
Example

Q = \{ 3, 7 \} \rightarrow \{ 7, 5 \}

Dequeue 3.
-- place neighbor 5 on the queue.
Example

\[ Q = \{7, 5\} \rightarrow \{5, 6\} \]

Deque 7.
-- place neighbor 6 on the queue.
Example

\[ Q = \{5, 6\} \rightarrow \{6\} \]

Deque 5.
-- no unvisited neighbors of 5.
Example

Adjacency List

Visited Table (T/F)

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Q = { 6 } → { }

Dequeue 6.
-- no unvisited neighbors of 6.
Example

Q = { } STOP!!! Q is empty!!!

What did we discover?

Look at “visited” tables.

There exist a path from source vertex 2 to all vertices in the graph!
Time Complexity of BFS
(Using adjacency list)

- Assume adjacency list
  - $n =$ number of vertices $m =$ number of edges

Algorithm $BFS(s)$

**Input:** $s$ is the source vertex

**Output:** Mark all vertices that can be visited from $s$.

1. for each vertex $v$
2. do $\text{flag}[v] := \text{false};$
3. $Q =$ empty queue;
4. $\text{flag}[s] := \text{true};$
5. $\text{enqueue}(Q, s);$
6. while $Q$ is not empty
7. do $v := \text{dequeue}(Q);$  
   No more than $n$ vertices are ever put on the queue.
8. for each $w$ adjacent to $v$
9. do if $\text{flag}[w] =$ false
10. then $\text{flag}[w] := \text{true};$
11. $\text{enqueue}(Q, w)$

How many adjacent nodes will we ever visit. This is related to the number of edges. How many edges are there?

$$\sum_{\text{vertex } v \in V} \text{deg}(v) = 2m^*$$

*Note: this is not per iteration of the while loop. This is the sum over all the while loops!
Time Complexity of BFS
(Using adjacency matrix)

- Assume adjacency matrix
  - \( n = \) number of vertices \( m = \) number of edges

Algorithm \( BFS(s) \)

Input: \( s \) is the source vertex
Output: Mark all vertices that can be visited from \( s \).

1. for each vertex \( v \)
2.   do \( flag[v] := \) false;
3. \( Q = \) empty queue;
4. \( flag[s] := \) true;
5. \( enqueue(Q, s) \);
6. while \( Q \) is not empty
7.   do \( v := dequeue(Q) \);
8.   for each \( w \) adjacent to \( v \)
9.     do if \( flag[w] = \) false
10.    then \( flag[w] := \) true;
11. \( enqueue(Q, w) \)

\( O(n^2) \)

So, adjacency matrix is not good for BFS!!!

No more than \( n \) vertices are ever put on the queue. \( O(n) \)

Using an adjacency matrix. To find the neighbors we have to visit all elements in the row of \( v \). That takes constant time \( O(n) \)!
Path Recording

• BFS only tells us if a path exists from source $s$, to other vertices $v$.
  - It doesn’t tell us the path!
  - We need to modify the algorithm to record the path.

• Not difficult
  - Use an additional predecessor array $\text{pred}[0..n-1]$
  - $\text{Pred}[w] = v$
    • Means that vertex $w$ was visited by $v$
BFS + Path Finding

Algorithm \( BFS(s) \)
1. for each vertex \( v \)
2. \( \text{do } \) \( flag(v) \) := false;
3. \( \text{pred}[v] := -1; \)
4. \( Q = \text{empty queue;} \)
5. \( flag[s] := \text{true;} \)
6. \( \text{enqueue}(Q, s); \)
7. \( \text{while } Q \text{ is not empty} \)
8. \( \text{do } v := \text{dequeue}(Q); \)
9. \( \text{for each } w \text{ adjacent to } v \)
10. \( \text{do if } flag[w] = \text{false} \)
11. \( \text{then } flag[w] := \text{true}; \)
12. \( \text{pred}[w] := v; \)
13. \( \text{enqueue}(Q, w) \)

Set \( \text{pred}[v] \) to -1 (let -1 means no path to any vertex)

Record who visited \( w \)
Adjacency List

Visited Table (T/F)

Q = {} Initialize Q to be empty

Initialize visited table (all empty F)

Initialize Pred to -1
Example

Adjacency List

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td>0</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Visited Table (T/F)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Flag that 2 has been visited.

\[ Q = \{ 2 \} \]

Place source 2 on the queue.
Example

Dequeue 2.
Place all unvisited neighbors of 2 on the queue

\[ Q = \{2\} \rightarrow \{8, 1, 4\} \]

Mark neighbors as visited.

Record in Pred who was visited by 2.
Example

Adjacency List

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Visit Table (T/F)

Q = \{ 8, 1, 4 \} → \{ 1, 4, 0, 9 \}

Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!
Example

Q = \{ 1, 4, 0, 9 \} \rightarrow \{ 4, 0, 9, 3, 7 \}

Dequeue 1.
-- Place all unvisited neighbors of 1 on the queue.
-- Only nodes 3 and 7 haven’t been visited yet.
Example

\[ Q = \{ 4, 0, 9, 3, 7 \} \rightarrow \{ 0, 9, 3, 7 \} \]

Dequeue 4.
-- 4 has no unvisited neighbors!
Example

Q = \{ 0, 9, 3, 7 \} \rightarrow \{ 9, 3, 7 \}

Dequeue 0.
-- 0 has no unvisited neighbors!
Example

Q = \{ 9, 3, 7 \} \rightarrow \{ 3, 7 \}

Dequeue 9.
-- 9 has no unvisited neighbors!
Example

\[ Q = \{ 3, 7 \} \rightarrow \{ 7, 5 \} \]

Dequeue 3.
-- place neighbor 5 on the queue.
Example

\[ Q = \{7, 5\} \rightarrow \{5, 6\} \]

Deque 7.
-- place neighbor 6 on the queue.

Mark new visited Vertex 6.

Record in Pred who was visited by 7.
Example

\[ Q = \{5, 6\} \rightarrow \{6\} \]

Dequeue 5.
-- no unvisited neighbors of 5.
Example

Q = \{ 6 \} → \{ \}  

Deque 6.  
-- no unvisited neighbors of 6.
**Example**

Adjacency List

```
0: 8
1: 3 7 9 2
2: 8 1 4
3: 4 5 1
4: 2 3
5: 3 6
6: 7 5
7: 1 6
8: 2 0 9
9: 1 8
```

Visited Table (T/F)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**Pred**

```
5 8
```

**Neighbors**

```
2
-
7
1
2
3
7
1
2
8
```

**Q** = \{ \}  
STOP!!!  Q is empty!!!

**Pred now stores the path!**
Pred array represents paths

Try some examples.
P
Path(0) ->
Path(6) ->
Path(1) ->

Algorithm $Path(w)$
1. if $pred[w] \neq -1$
2. then
3. $Path(pred[w])$
4. output $w$
BFS tree

• We often draw the BFS paths as a m-ary tree, where $s$ is the root.

Question: What would a “level” order traversal tell you?
Can also keep track of the distance from the source

• Assumption: if an edge \((u, v)\) exists, then distance is 1 from \(u\) to \(v\)

• In the next example code, each vertex initially has a color (white). When it is enqueued, color is set to gray. When it is dequeued color is set to black.
BFS(G, s)

for each vertex $u \in G, V - \{s\}$
  $u.color = \text{white}; u.d = \infty; \quad u.\pi = \text{nil}$

$s.color = \text{gray}; s.d = 0; s.\pi = \text{nil}, Q = \emptyset; \text{Enqueue}(Q, s)$

while $Q \neq \emptyset$
  $u = \text{Dequeue}(Q)$
  for each $v \in G.\text{Adj}[u]$
    if $v.color == \text{white}$
      $v.color = \text{gray}$
      $v.d = u.d + 1$
      $v.\pi = u$
      $\text{Enqueue}(Q, v)$

$u.color = \text{black}$
BFS

- Breadth first search is a basis for many algorithms including spanning trees and single source shortest path
- Next, we will explore depth first search and applications
Another example BFS
Depth First Search

• Recall that in breadth first search, the discovery of nodes proceeds as a wavefront hitting all nodes 1 distance from the source, then 2, etc.

• In depth first search we instead search deeper into the graph whenever possible
  - dfs explores the edges out of the most recently discovered vertex first
  - dfs creates a forest of trees that are reachable from the tree roots
The DFS algorithm

• Input: \( G = (V, E) \)
  - directed or undirected.
  - No source vertex is assigned

• Output: 2 timestamps for each vertex
  - \( v.d \) = discovery time
  - \( v.f \) = finishing time

• Explore every edge
  - start from different vertices as needed
  - as soon as a vertex is discovered, explore from it
  - keep track of a predecessor tree by keeping track of a predecessor function \( v.\pi \) for each node

• As algorithm progresses vertex colors represent
  - white: undiscovered node
  - gray: discovered node but not finished exploring
  - black: finished exploring from this node
DFS pseudocode

DFS(G)
for each vertex \( u \in G.V \)
    \( u.\text{color} = \text{white}; \ u.\pi = \text{nil}; \)
\( \text{time} = 0 \)
for each vertex \( u \in G.V \)
    if \( u.\text{color} == \text{white} \)
        DFS-visit(G, u)

DFS-visit(G, u)
\( \text{time} = \text{time} + 1; \ u.d = \text{time}; \ u.\text{color} = \text{gray} \)
for each \( v \in G.\text{Adj}[u] \)
    if \( v.\text{color} == \text{white} \)
        \( v.\pi = u \);
        DFS-visit(G, v)
\( u.\text{color} = \text{black}; \ \text{time} = \text{time} + 1; \ u.f = \text{time} \)
Example DFS