Graphs
An Introduction

Outline
• What are Graphs?
• Applications
  Terminology and Problems
  Representation (Adj. Mat and Linked Lists)
• Searching
  – Depth First Search (DFS)
  – Breadth First Search (BFS)

Graphs
• A graph \( G = (V,E) \) is composed of:
  – \( V \): set of vertices
  – \( E \subset V \times V \): set of edges connecting the vertices
An edge \( e = (u,v) \) is a ___ pair of vertices
  – Directed graphs (ordered pairs)
  – Undirected graphs (unordered pairs)
Undirected Graph

Applications

- Air Flights, Road Maps, Transportation.
- Graphics / Compilers
- Electrical Circuits
- Networks
- Modeling any kind of relationships (between people/web pages/cities/...)

Some More Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Media</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>cascade</td>
<td>gate</td>
<td>wings</td>
</tr>
</tbody>
</table>
**World Wide Web**

- Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.

![Web graph diagram](image)

**9-11 Terrorist Network**

- Social network graph.
- Node: people.
- Edge: relationship between two people.

![9-11 terrorist network diagram](image)

**Ecological Food Web**

- Food web graph.
- Node: species.
- Edge: from prey to predator.

![Ecological food web diagram](image)
TERMINOLOGY

• a is adjacent to b iff \((a, b) \in E\).
• degree(a) = number of adjacent vertices (Self loop counted twice)
• Self Loop: \((a, a)\)
• Parallel edges: \(E = \{ ...(a, b), (a, b)... \}\)

TERMINOLOGY

• 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.

MORE...

simple graph  multigraph  pseudograph
**QUESTION**

- Max Degree node? Min Degree Node?
- Isolated Nodes? Total sum of degrees over all vertices? Number of edges?

**QUESTION**

- Max Degree = 4. Isolated vertices = 1.
- $|V| = 8$, $|E| = 8$
- Sum of degrees = 16 = ?
  - (Formula in terms of $|V|$, $|E|$ ?)

**QUESTION**

- Max Degree = 4. Isolated vertices = 1.
- $|V| = 8$, $|E| = 8$
- Sum of degrees = 2$|E| = \sum_{v \in V} \text{degree}(v)$
  - Handshaking Theorem. Why?
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 $\text{degree}(v)$ is odd for all $v \in V_2$, $|V_2|$ must be even.
**TERMINOLOGY**

A graph \( H(V_h, E_h) \) is a subgraph of \( G(V_c, E_c) \) if and only if \( V_h \subseteq V_c \) and \( E_h \subseteq E_c \).

**Path and Cycle**

- An alternating sequence of vertices and edges beginning and ending with vertices
  - each edge is incident with the vertices preceding and following it.
  - No edge appears more than once.
  - A path is **simple** if all nodes are distinct.
- **Cycle**
  - A path is a cycle if and only if \( v_0 = v_k \)
    - The beginning and end are the same vertex.

**PATH EXAMPLE**
**Connected Graph**

- Undirected Graphs: If there is at least one path between every pair of vertices. (otherwise disconnected)
- Directed Graphs:
  - Strongly connected
  - Weakly connected

**Hamiltonian Cycle**

- A cycle that transverses every vertex exactly once.

In general, the problem of finding a Hamiltonian circuit is NP-Complete.

**Complete Graph**

- Every pair of graph vertices is connected by an edge.

\[ n(n-1)/2 \] edges
**DIRECTED A CY CLIC GRAPHS**

A DAG is a directed graph with no cycles

Often used to indicate precedences among events, i.e., event $a$ must happen before $b$

---

**TREE**

A connected graph with $n$ nodes and $n-1$ edges

A Forest is a collection of trees.

---

**TREES**

- An undirected graph is a **tree** if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
**ROOTED TREES**

- Rooted tree. Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

```
    r
   / \
  a   b
 /     /
v     v
```

- Importance. Models hierarchical structure.

**PHYLOGENY TREES**

- Phylogeny trees. Describe evolutionary history of species.

```
  - Plants
  - Animals
  - Fish
  - Bacteria
  - Viruses
```

**GUI CONTAINMENT HIERARCHY**

- GUI containment hierarchy. Describe organization of GUI widgets.
**SPANNING TREE**

Connected subset of a graph G with \( n-1 \) edges which contains all of V.

**INDEPENDENT SET**

- An independent set of G is a subset of the vertices such that no two vertices in the subset are adjacent.

**CLIQUE**

- a.k.a. complete subgraphs.
TOUGH PROBLEM

• Find the maximum cardinality independent set of a graph G.
  – NP-Complete
  – Unknown if a poly time algorithm exists unless P = NP.

TSP

• Given a weighted graph G, the nodes of which represent cities and weights on the edges, distances; find the shortest tour that takes you from your home city to all cities in the graph and back.
  – Can be solved in O(n!) by enumerating all cycles of length n.
  – Dynamic programming can be used to reduce it in O(n²ⁿ).

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>4</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

**Adjacency List**

1 → 2 → 3 → 4
2 → 1 → 4
3 → 1 → 4
4 → 1 → 2

**Adjacency Matrix**

\[
\begin{pmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{pmatrix}
\]
Another Example

1. Adjacency Matrix

2. Adjacency List

AL Vs AM

• AL: Takes $O(|V| + |E|)$ space

• AM: Takes $O(|V|^2)$ 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 $O(|V|^2)$ 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.
AL VS AM

- AL: Total space = $8|V| + 16|E|$ bytes (For undirected graphs its $8|V| + 32|E|$ bytes)
- AM: $|V| \times |V| / 8$

Question: What is better for very sparse graphs? (Few number of edges)

GRAPH TRAVERSAL

- 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.

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.

Source: Lecture notes by Sheung-Hung POON

BFS

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 `β[s][v] := false;`
3. `Q := empty queue;`
4. `β[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 `β[s][w] = false`
10. then `β[s][w] := true;`
11. `enqueue(Q, w)

EXAMPLE

```
Q = \{ \}
Initialize Q to be empty
```

```
Adjacency List
```
```
Visited Table (T/F)
```

```
Initialize visited table (all empty F)
```
Example

Adjacency List
0: [0, 1, 2, 3]
1: [2, 3, 4, 5]
2: [1, 3, 4]
3: [2, 4, 5]
4: [2, 3, 6]
5: [4, 6]
6: [4, 5]
7: [8, 9]
8: [7, 9]
9: [7, 8]

Visited Table (T/F)
F  T  T  F  T  T  T  F  T  

Q = {2}
Place source 2 on the queue.

Flag that 2 has been visited.

Q = {2} → {8, 1, 4}
Dequeue 2.
Place all unvisited neighbors of 2 on the queue.

Mark neighbors as visited.

Q = {8, 1, 4} → {1, 4, 0, 9}
Mark new visited neighbors.

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 \} → \{ 4, 0, 9, 3, 7 \}

Mark new visited neighbors.

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

Q = \{ 4, 0, 9, 3, 7 \} → \{ 0, 9, 3, 7 \}

Dequeue 4.
→ 4 has no unvisited neighbors!

Q = \{ 0, 9, 3, 7 \} → \{ 9, 3, 7 \}

Dequeue 0.
→ 0 has no unvisited neighbors!
\( Q = \{9, 3, 7\} \rightarrow \{3, 7\} \)

Dequeue 9.
→ 9 has no unvisited neighbors!

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

Dequeue 3.
→ place neighbor 5 on the queue.

\( Q = \{7, 5\} \rightarrow \{5, 6\} \)

Dequeue 7.
→ place neighbor 6 on the queue.
**EXAMPLE**

Adjacency List

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

**Visited Table (T/F)**

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

**Q = \{5, 6\} \rightarrow \{6\}**

Dequeue 5.

-- no unvisited neighbors of 5.

**EXAMPLE**

Adjacency List

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

**Visited Table (T/F)**

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

**Q = \{6\} \rightarrow \{\}**

Dequeue 6.

-- no unvisited neighbors of 6.

**EXAMPLE**

Adjacency List

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

**Visited Table (T/F)**

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

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

1. for each vertex \( v \)
2. \( \text{do } fa[s] \equiv \text{false;} \)
3. \( Q \equiv \text{empty queue;} \)
4. \( fa[s] \equiv \text{true;} \)
5. \( \text{enque}(Q, s); \)
6. while \( Q \) is not empty
7. \( \text{deq}(Q, v); \)
8. for each \( w \) adjacent to \( v \)
9. \( \text{do if } fa[w] \equiv \text{false } \)
10. \( \text{then } fa[w] \equiv \text{true;} \)
11. \( \text{enque}(Q, w); \)

\[ O(n + m) \]

No more than \( n \) vertices are ever put on the queue.
How many adjacent nodes will we ever visit? This is related to the number of edges. How many edges are there?
\[ \sum_{v} deg(v) = 2m \]

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

1. for each vertex \( v \)
2. \( \text{do } fa[s] \equiv \text{false;} \)
3. \( Q \equiv \text{empty queue;} \)
4. \( fa[s] \equiv \text{true;} \)
5. \( \text{enque}(Q, s); \)
6. while \( Q \) is not empty
7. \( \text{deq}(Q, v); \)
8. for each \( w \) adjacent to \( v \)
9. \( \text{do if } fa[w] \equiv \text{false } \)
10. \( \text{then } fa[w] \equiv \text{true;} \)
11. \( \text{enque}(Q, w); \)

\[ O(n^2) \]

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

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 \( v \)
**Algorithm BFS**

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

---

**Example**

**Adjacency List**

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

**Visited Table**

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

**Example**

**Adjacency List**

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

**Visited Table**

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

**Q = \{2\}**

Flag that 2 has been visited.

Place source 2 on the queue.
**Example**

### Adjacency List

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

### Visited Table (T/F)

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

### Marking Process

1. **Dequeue 2.** Place all unvisited neighbors of 2 on the queue: 8, 4, 1.
4. Notice that 2 is not placed on the queue again; it has been visited.
6. Record in Pred who was visited by 8: 2.
7. Dequeue 1. Place all unvisited neighbors of 1 on the queue: 3, 7.
8. Only nodes 3 and 7 haven't been visited yet.
Q = \{4, 0, 9, 3, 7\} \rightarrow \{0, 9, 3, 7\}

Dequeue 4.

4 has no unvisited neighbors!

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

Dequeue 0.

0 has no unvisited neighbors!

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

Dequeue 9.

9 has no unvisited neighbors!
Adjacency List

source
0
1
2
3
4
5
6
7
8
9

Visited Table (T/F)

T
T
T
T
T
T
F
T
T
T

Q = \{3, 7\} → \{7, 5\}

Dequeue 3.
Query 3. Place neighbor 5 on the queue.
Mark new visited Vertex 5.
Record in Pred who was visited by 3.

Q = \{7, 5\} → \{5, 6\}

Dequeue 7.
Query 7. Place neighbor 6 on the queue.
Mark new visited Vertex 6.
Record in Pred who was visited by 7.

Q = \{5, 6\} → \{6\}

Dequeue 5.
Query 5. No unvisited neighbors of 5.
Example

Adjacency List

Visited Table (T/F)

Q = \{6\} \rightarrow \{

Dequeue 6.

no unvisited neighbors of 6.

Q = \{

STOP!!! Q is empty!!!

Pred now stores the path!

Pred array represents paths

Algorithm Path(u)
1. if \text{pred(u)} \neq -1 then
2. \text{Path}([\text{pred}(u)])
3. output u

Try some examples.
Path(0) =>
Path(6) =>
Path(1) =>
**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?

**CONNECTED COMPONENT**
- Connected component. Find all nodes reachable from $s$.

**FLOOD FILL**
- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.
FLOOD FILL

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.

CONNECTED COMPONENT

- Connected component. Find all nodes reachable from s.

R will consist of nodes to which s has a path
Initially R={s}
While there is an edge (u,v) where u ∈ R and v ∉ R
Add v to R
Endwhile

MORE ON PATHS AND TREES IN GRAPHS
BFS

• Another way to think of the BFS tree is the physical analogy of the BFS Tree.
  Sphere-String Analogy: Think of the nodes as spheres and edges as unit length strings. Lift the sphere for vertex s.

**SPHERE-STRING ANALOGY**

BFS: PROPERTIES

- At some point in the running of BFS, Q only contains vertices/nodes at layer d.
- If u is removed before v in BFS then \( \text{dist}(u) \leq \text{dist}(v) \)
- At the end of BFS, for each vertex v reachable from s, the \( \text{dist}(v) \) equals the shortest path length from s to v.
BFS

Processes nodes layer by layer

BFS: ADVANCING WAVEFRONT

OLD WINE IN NEW BOTTLE

forall v ∈ V:
dist(v) = ∞; prev(v) = null;
dist(s) = 0
Queue q; q.push(s);
while (!Q.empty())
v = Q.dequeue();
for all e=(v,w) in E
if dist(w) = ∞:
  − dist(w) = dist(v)+1
  − Q.enqueue(w)
  − prev(w)= v
DIJKSTRA’S SSSP ALG

BFS WITH POSITIVE INT WEIGHTS

• For every edge $e = (a, b) \in E$, let $w_e$ be the weight associated with it. Insert $w_e-1$ dummy nodes between $a$ and $b$. Call this new graph $G'$.
• Run BFS on $G'$. $\text{dist}(u)$ is the shortest path length from $s$ to node $u$.
• Why is this algorithm bad?

HOW DO WE SPEED IT UP?

• If we could run BFS without actually creating $G'$, by somehow simulating BFS of $G'$ on $G$ directly.
• Solution: Put a system of alarms on all the nodes. When the BFS on $G'$ reaches a node of $G$, an alarm is sounded. Nothing interesting can happen before an alarm goes off.

AN EXAMPLE
Another Example

**Alarm Clock Alg**

```
alarm(s) = 0
until no more alarms
  - wait for an alarm to sound. Let next alarm that
    goes off is at node v at time t.
  • dist(s, v) = t
  • for each neighbor w of v in G:
    - if there is no alarm for w, alarm(w) = t + weight(v, w)
    - if w's alarm is set further in time than t + weight(v, w), reset
      it to t + weight(v, w).
```

**Recall BFS**

```
for all v \in V:
  dist(v) = \infty; prev(v) = null;
dist(s) = 0
Queue q; q.push(s);
while (!Q.empty())
  v = Q.dequeue();
  for all e=(v, w) in E
    if dist(w) = \infty:
      - dist(w) = dist(v) + 1
      - Q.enqueue(w)
      - prev(w) = v
```

**DIJKSTRA’S SSSP**

forall $v \in V$:
- $\text{dist}(v) = \infty$; $\text{prev}(v) = \text{null}$;
- $\text{dist}(s) = 0$

Magic_DS $Q$; $Q$.insert($s$,0);

while (!$Q$.empty())
- $v = Q$.delete_min();
  for all $(v,w) \in E$
    if $\text{dist}(w) > \text{dist}(v) + \text{weight}(v,w)$ :
      - $\text{dist}(w) = \text{dist}(v) + \text{weight}(v,w)$
      - $Q$.insert($w$, $\text{dist}(w)$)
      - $\text{prev}(w) = v$

**THE MAGIC DS: PQ**

- What functions do we need?
  - insert() : Insert an element and its key. If the element is already there, change its key (only if the key decreases).
  - delete_min() : Return the element with the smallest key and remove it from the set.

**EXAMPLE**
1. Start from s
2. Grow a region R around s such that the SPT from s is known inside the region.
3. Add v to R such that v is the closest node to s outside R.
4. Keep building this region till R = V.
HOW DO WE FIND $V$?

Pick $v \notin R$ s.t.

$$\min_{x \in R} \text{dist}(s, x) + \text{weight}(x, v)$$

Let $(x^*, v^*)$ be the opt.

EXAMPLE

![Graph with nodes and edges]

Is this the shortest path to $V^*$?

Why?
OLD WINE IN NEW BOTTLE

forall v ∈ V:
dist(v) = ∞; prev(v) = null;
dist(s) = 0
R = {};
while R ≠ V
Pick v not in R with smallest distance to s
for all edges (v,z) ∈ E
if(dist(z) > dist(v) + weight(v,z))
dist(z) = dist(v) + weight(v,z)
prev(z) = v;
Add v to R

UPDATES

Update rule:
(Best way to reach z?)

RUNNING TIME?

delete-min = ?
insert = ?
RUNNING TIME?

\[ \text{delete-min} = |V| \]
\[ \text{insert} = |E| \]

If we used a linked list as our magic data structure:

\[ \text{delete-min}() \rightarrow O(|V|) \]
\[ \text{insert}() \rightarrow O(1) \cdot O(|V|) \]
\[ \text{Total} = |V| \text{ delete-min}() + |E| \text{ insert()} = O(|V|^2) \]

BINARY HEAP?

\[ \text{delete-min()} \rightarrow O(\log |V|) \]
\[ \text{insert()} \rightarrow O(\log |V|) \]
\[ \text{Total} \rightarrow O(|E| \log |V|) \]
**D-ARY HEAP**

- \( \text{delete-min}() \rightarrow O(d \log dv) \)
- \( \text{insert}() \rightarrow O(d \log dv) \)
- \( \text{Total} \rightarrow O(\frac{dv}{d} + \frac{E}{d} \log dv) \)

**FIBONACCI HEAP**

- \( \text{delete-min}() \rightarrow O(1) \) \text{ Amortized} \n- \( \text{insert}() \rightarrow O(\log dv) \)
- \( \text{Total} \rightarrow O(dv \log dv + E) \)

**A SPANNING TREE**

- Recall?
- Is it unique?
- Is shortest path tree a spanning tree?
- Is there an easy way to build a spanning tree for a given graph \( G \)?
- Is it defined for disconnected graphs?
Spanning Tree

Connected subset of a graph $G$ with $n-1$ edges which contains all of $V$.

A connected, undirected graph

Some spanning trees of the graph

Easy Algorithm

To build a spanning tree:

Step 1: $T =$ one node in $V$, as root.

Step 2: At each step, add to tree one edge from a node in tree to a node that is not yet in the tree.
Spanning Tree Property

Adding an edge \( e=(a,b) \) not in the tree creates a cycle containing only edge \( e \) and edges in spanning tree.

Why?

Spanning Tree Property

- Let \( c \) be the first node common to the path from \( a \) and \( b \) to the root of the spanning tree.
- The concatenation of \( (a,b) \ (b,c) \ (c,a) \) gives us the desired cycle.

Lemma 1

- In any tree, \( T=(V,E) \), \( |E|=|V|-1 \)
- Why?
LEMA 1

- In any tree, $T = (V,E)$,
  $|E| = |V| - 1$

  Why?
  - Tree $T$ with 1 node has zero edges.
  - For all $n > 0$, $P(n)$ holds, where
    - $P(n)$: A Tree with $n$ nodes has $n - 1$ edges.
  - Apply MI. How do we prove that given $P(m)$ true for all $1..m$, $P(m+1)$ is true?

UNDIRECTED GRAPHS N TREES

- An undirected graph $G = (V,E)$ is a tree iff
  1. it is connected
  2. $|E| = |V| - 1$

LEMA 2

Let $C$ be the cycle created in a spanning tree $T$
by adding the edge $e = (a,b)$ not in the tree.
Then removing any edge from $C$ yields
another spanning tree.

Why? How many edges and vertices does the new graph have? Can $(x,y)$ in $G$ get disconnected in this new tree?
LEMMA 2

- Let $T'$ be the new graph
- $T'$ has $n$ nodes and $n-1$ edges, so it must be a tree if it is connected.
- Let $(x,y)$ be not connected in $T'$. The only problem in the connection can be the removed edge $(a,b)$. But if $(a,b)$ was contained in the path from $x$ to $y$, we can use the cycle $C$ to reach $y$ (even if $(a,b)$ was deleted from the graph).

WEIGHTED SPANNING TREES

Let $w_e$ be the weight of an edge $e$ in $G=(V,E)$.

Weight of spanning tree = Sum of edge weights.

Question: How do we find the spanning tree with minimum weight. This spanning tree is also called the Minimum Spanning Tree.

Is the MST unique?

MINIMUM SPANNING TREES

- Applications
  - networks
  - cluster analysis
    - used in graphics/pattern recognition
  - approximation algorithms (TSP)
  - bioinformatics/CFD
CUT PROPERTY

Let X be a subset of V. Among edges crossing between X and V \ X, let e be the edge of minimum weight. Then e belongs to the MST.

Proof?

CYCLE PROPERTY

For any cycle C in a graph, the heaviest edge in C does not appear in the MST.

Proof?

QUESTION

Is the SSSP Tree and the Minimum spanning tree the same? Is one the subset of the other always?
**QUESTION**

- Is the SSSP Tree and the Minimum spanning tree the same?
- Is one the subset of the other always?

**OLD WINE IN NEW BOTTLE**

forall $v \in V$:
- $dist(v) = \infty$; $prev(v) = \text{null}$;
- $dist(s) = 0$
- Heap $Q$; $Q.insert(s,0)$;
- while (!$Q.empty()$)
  - $v = Q.delete\_min()$;
  - for all $e=(v,w) \in E$
    - if $dist(w) > dist(v)+weight(v,w)$ :
      - $dist(w) = dist(v)+weight(v,w)$
      - $Q.insert(w, dist(w))$
      - $prev(w)= v$

**A SLIGHT MODIFICATION**

JARNIK’S OR PRIM’S ALG.

forall $v \in V$:
- $dist(v) = \infty$; $prev(v) = \text{null}$;
- $dist(s) = 0$
- Heap $Q$; $Q.insert(s,0)$;
- while (!$Q.empty()$)
  - $v = Q.delete\_min()$;
  - for all $e=(v,w) \in E$
    - if $dist(w) > dist(v)+weight(v,w)$ :
      - $dist(w) = dist(v)+weight(v,w)$
      - $Q.insert(w, dist(w))$
      - $prev(w)= v$
OUR FIRST MST ALG.

forall \( v \in V \):
\[
\text{dist}(v) = \infty; \text{prev}(v) = \text{null};
\]
\[
\text{dist}(s) = 0
\]

Magic_DS Q; Q.insert(s,0);

while (!Q.empty())
\[
\text{v} = Q.\text{delete\_min}();
\]

for all \( e=(v,w) \) in \( E \)
\[
\text{if dist}(w) > \text{weight}(v,w) :
\]
\[
- \text{dist}(w) = \text{weight}(v,w)
- Q.\text{insert}(w, \text{dist}(w))
- \text{prev}(w) = v
\]

HOW DOES THE RUNNING TIME DEPEND ON THE MAGIC_DS?

- heap?
- insert()?
- delete_min()?
- Total time?
- What if we change the Magic_DS to fibonacci heap?

PRIM’S/JARNIK’S ALGORITHM

- best running time using fibonacci heaps

- \( O(E + V \log V) \)

Why does it compute the MST?
**ANOTHER ALG: KRUSHKAL’S**

- sort the edges of G in increasing order of weights
- Let $S = \emptyset$
- for each edge $e$ in G in sorted order
  - if the endpoints of $e$ are disconnected in $S$
  - Add $e$ to $S$

**HAVE YOU SEEN THIS BEFORE?**

- Sort edges of G in increasing order of weight
- $T = \emptyset$  // Collection of trees
- For all $e$ in $E$
  - If $T \cup \{e\}$ has no cycles in $T$
    - then $T = T \cup \{e\}$

return $T$

Naïve running time $O((|V| + |E|)|V|) = O(|E| |V|)$

**HOW TO SPEED IT UP?**

- To $O(E + V \log V)$
  - Using union find data structures.
- Surprisingly the idea is very simple.
3.4 TESTING BIPARTITENESS

BIPARTITE GRAPHS

An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

- **Applications.**
  - Stable marriage: men = red, women = blue.
  - Scheduling: machines = red, jobs = blue.
**Testing Bipartiteness**

Given a graph $G$, is it bipartite?

- Many graph problems become:
  - Easier if the underlying graph is bipartite (matching)
  - Tractable if the underlying graph is bipartite (independent set)

- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![A bipartite graph $G$](image1)

![Another drawing of $G$](image2)

---

**An Obstruction to Bipartiteness**

- **Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle.

- **Pf.** Not possible to 2-color the odd cycle, let alone $G$.

![Bipartite (2-colorable)](image3)

![Not bipartite (not 2-colorable)](image4)

---

**Bipartite Graphs**

- **Lemma.** Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds:
  1. No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
  2. An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

![Case (i)](image5)

![Case (ii)](image6)
**BIPARTITE GRAPHS**

- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds:
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

**Pf.**

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

**OBSTRUCTION TO BIPARTITENESS**

- Corollary. A graph G is bipartite iff it contain no odd length cycle.
3.5 CONNECTIVITY IN DIRECTED GRAPHS

DIRECTED GRAPHS
- Directed graph. \( G = (V, E) \)
  - Edge \((u, v)\) goes from node \(u\) to node \(v\).

- Ex. Web graph - hyperlink points from one web page to another.
  - Directedness of graph is crucial.
  - Modern web search engines exploit hyperlink structure to rank web pages by importance.

GRAPH SEARCH
- Directed reachability. Given a node \(s\), find all nodes reachable from \(s\).

- Directed \(s\)-\(t\) shortest path problem. Given two node \(s\) and \(t\), what is the length of the shortest path between \(s\) and \(t\)?

- Graph search. BFS extends naturally to directed graphs.

- Web crawler. Start from web page \(s\). Find all web pages linked from \(s\), either directly or indirectly.
Strong Connectivity

- **Def.** Node \( u \) and \( v \) are mutually reachable if there is a path from \( u \) to \( v \) and also a path from \( v \) to \( u \).
- **Def.** A graph is strongly connected if every pair of nodes is mutually reachable.
- **Lemma.** Let \( s \) be any node. \( G \) is strongly connected iff every node is reachable from \( s \), and \( s \) is reachable from every node.
- **Pf.** \( \Rightarrow \) Follows from definition.
- **Pf.** \( \Leftarrow \) Path from \( u \) to \( v \): concatenate \( u \)-s path with s-v path.
  Path from \( v \) to \( u \): concatenate \( v \)-s path with s-u path.

Strong Connectivity: Algorithm

- **Theorem.** Can determine if \( G \) is strongly connected in \( O(m + n) \) time.
- **Pf.**
  - Pick any node \( s \).
  - Run BFS from \( s \) in \( G \).
  - Run BFS from \( s \) in \( G^{\text{rev}} \).
  - Return true iff all nodes reached in both BFS executions.
  - Correctness follows immediately from previous lemma.

3.6 DAGs and Topological Ordering
**Directed Acyclic Graphs**

- Def. An **DAG** is a directed graph that contains no directed cycles.
- Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).
- Def. A topological order of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).

![Diagram of a DAG and a topological ordering]

---

**Precedence Constraints**

- Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).
- Applications.
  - Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
  - Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).

---

**Directed Acyclic Graphs**

- Lemma. If \(G\) has a topological order, then \(G\) is a DAG.
- Pf. (by contradiction)
  - Suppose that \(G\) has a topological order \(v_1, \ldots, v_n\) and that \(G\) also has a directed cycle \(C\).
  - Let’s see what happens.
  - Let \(v_i\) be the lowest-indexed node in \(C\), and let \(v_j\) be the node just before \(v_i\); thus \((v_j, v_i)\) is an edge.
  - By our choice of \(i\), we have \(i < j\).
  - On the other hand, since \((v_j, v_i)\) is an edge and \(v_1, \ldots, v_n\) is a topological order, we must have \(j < i\), a contradiction. ▪

![Diagram of a supposed cycle \(C\) and the supposed topological order \(v_1, \ldots, v_n\)]
Directed Acyclic Graphs

- **Lemma.** If $G$ has a topological order, then $G$ is a DAG.

- **Q.** Does every DAG have a topological ordering?

- **Q.** If so, how do we compute one?

Directed Acyclic Graphs

- **Lemma.** If $G$ is a DAG, then $G$ has a node with no incoming edges.

  - **Pf.** (by contradiction)
    - Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
    - Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$, we can walk backward to $u$.
    - Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
    - Repeat until we visit a node, say $w$, twice.
    - Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle. □

Directed Acyclic Graphs

- **Lemma.** If $G$ is a DAG, then $G$ has a topological ordering.

  - **Pf.** (by induction on $n$)
    - Base case: true if $n = 1$.
    - Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
    - $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
    - By inductive hypothesis, $G - \{v\}$ has a topological ordering.
    - Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges. □

To compute a topological ordering of $G$:

- Find a node $v$ with no incoming edges and order it first.
- Delete $v$ from $G$.
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. □
Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Proof.
- Maintain the following information:
  - $\text{count}[w] = \text{remaining number of incoming edges}$
  - $S = \text{set of remaining nodes with no incoming edges}$

  Initialisation: $O(m + n)$ via single scan through graph.

  Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge