Graph Algorithms
A graph $G = (V, E)$ is an ordered pair consisting of
- a set $V$ of vertices (singular: vertex),
- a set $E \subseteq V \times V$ of edges.

$E$ can be a set of ordered pair or unordered pairs
- $G = (V, E)$ is directed graph if $E$ consists of ordered pairs of vertices.
- $G = (V, E)$ is an undirected graph if $E$ consists of unordered pairs of vertices.
- Number of vertices: $|V|
- Number of edges: $|E|$
Graph Example

- Here is a graph $G = (V, E)$
  - Each edge is a pair $(v_1, v_2)$, where $v_1, v_2$ are vertices in $V$

\begin{align*}
V &= \{A, B, C, D, E, F\} \\
E &= \{(A,B),(A,D),(B,C),(C,D),(C,E),(D,E)\}
\end{align*}

\begin{align*}
V &= \{A, B, C\} \\
E &= \{(A,B),(A,C),(B,C)\}
\end{align*}
Undirected Graph: Terminology

- u and v are **adjacent** in an undirected graph G if (u,v) is an edge in G
  - edge $e = (u,v)$ is **incident** with vertex u and vertex v
- **degree** of a node: $\text{deg}(v)$: the number of edges incident to v
Vertex $u$ is adjacent to vertex $v$ in a directed graph $G$ if $(u,v)$ is an edge in $G$
  ○ vertex $u$ is the initial vertex of $(u,v)$
Vertex $v$ is adjacent from vertex $u$
  ○ vertex $v$ is the terminal (or end) vertex of $(u,v)$

- the indegree of a node $v$, $\text{indeg}(v)$, is the number of edges entering $v$
- the outdegree of a node $v$, $\text{outdeg}(v)$, is the number of edges leaving $v$
- $\text{deg}(u) = \text{indeg}(u) + \text{outdeg}(u)$
- **Source**: a node $u$ is a source if $\text{indeg}(u) = 0$
- **Sink**: a node $u$ is a sink if $\text{outdeg}(u) = 0$
Terminologies

- **A Path of length k:**
  - In the graph $G=(V,E)$, a path from vertex $u$ to vertex $v$ is a sequence $(v_0, v_1, \cdots, v_k)$ of vertices such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path.

- The vertex $v$ is **reachable** from $u$ if there is a path from $u$ to $v$

- **Simple path:** A path in which all the vertices are distinct

- **Cycle:** A special type of path where starting and ending vertices are the same. $(v_0, v_1, \cdots, v_k)$ forms a cycle if $v_0 = v_k$, and the path contains at least one edge.

- A cycle is a special type of path starting and ending at the same vertex.

- A graph with no cycles is **acyclic**.

- **Subgraph:** graph $H = (V', E')$ is subgraph of $G = (V, E)$ if $V'$ is subset of $V$, and $E'$ is a subset of $E$. 
Graph properties

In both undirected and directed graphs: $|E| = O(|V|^2)$

- **Proof:**
  - every edge connects two distinct vertices (G has no loops)
  - No two edges connect the same pair of vertices (G has no multi-edges)
  - G has at most $\binom{n}{2}$ edges in an undirected graph and $2 \times \binom{n}{2}$ in a directed graph

$O(V + E) = O(V^2)$

- Which one is a better runtime? $O(v^2)$ or $O(V+E)$?
  - $O(V + E)$
Graph properties

- A graph is called **dense** if $E = \Theta(V^2)$
  - Most pairs of vertices are connected by an edge
- A graph is called **sparse** if it is not dense: $E \ll V^2$
  - There are very few edges in the graph
- **Connected graph**: an undirected graph in which every vertex is reachable from all other vertices
- In an undirected graph, if the graph is connected:
  - There must be at least $|V| - 1$ edges $\rightarrow |E| \geq |V| - 1$
  - Proof by induction on the number of vertices
- $E = \Theta(V^2)$ and $|E| \geq |V| - 1 \rightarrow \log |E| = \Theta(\log V)$
- **Strongly connected graph**: a directed graph in which every two vertices are reachable from each other
Graph

- **Tree**:  
  - A connected (undirected) graph without any cycle  
  - Tree is a graph with exactly one path between any pair of vertices  
  - Yet another definition: a tree is a connected graph with $|V| - 1$ edges, i.e., $|E| = |V| - 1$  
    - Proof by induction
Storing Graphs

There are two ways to store a graph:

- Adjacency matrix
- Adjacency list
Storing Graphs: Adjacency-matrix

The adjacency matrix of a graph $G = (V, E)$

- $V = \{1, 2, \ldots, n\}$ is the set of vertices of the graphs
- is an $n \times n$ matrix $A$

$$A[i, j] = \begin{cases} 
0 & \text{if } (i, j) \notin E \\
1 & \text{if } (i, j) \in E 
\end{cases}$$

- Space: $O(n^2)$
Storing Graphs: Adjacency-matrix: Example

$$\Theta(V^2) \text{ storage}$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Storing Graphs: Adjacency list

- An adjacency list of a vertex $v \in V$ is the list $\text{Adj}[v]$ of vertices adjacent to $v$
- For undirected graphs, $|\text{Adj}[v]| = \text{degree}(v)$
- For directed graphs, $|\text{Adj}[v]| = \text{out-degree}(v)$
- Adjacency lists use $\Theta(V + E)$ storage
- **Handshaking Lemma**: for undirected graphs $\sum_{v \in V} \text{deg}(v) = 2|E|$

$$
\text{Adj}[1] = \{2, 3\} \\
\text{Adj}[2] = \{3\} \\
\text{Adj}[3] = \{} \\
\text{Adj}[4] = \{3\}
$$
## Storing Graphs

- **Adjacency Matrix vs Adjacency List**

<table>
<thead>
<tr>
<th></th>
<th>Adjacency matrix</th>
<th>Adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>((u, v) \in E)</td>
<td>(\Theta(1))</td>
<td>(O(\text{deg}(u)))</td>
</tr>
<tr>
<td>Time to list u’s neighbor</td>
<td>(\Theta(n))</td>
<td>(\Theta(\text{deg}(u)))</td>
</tr>
<tr>
<td>Time to list all edges</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n+m))</td>
</tr>
<tr>
<td>Space complexity</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n+m))</td>
</tr>
</tbody>
</table>
Storing Graphs

Adjacency Matrix

- **Advantage:**
  - O(1) test for presence or absence of edges
- **Disadvantage:**
  - Inefficient for sparse graphs
  - Storage not efficient
  - Accessing edges not efficient

Adjacency List

- **Advantage:**
  - Good for sparse graphs
  - Accessing edges are easy
- **Disadvantage:**
  - More complex data structure
  - Not possible to access an edge in O(1)
BFS
Breadth-First search

- A graph traversal algorithm
- **Input:** A graph $G=(V, E)$ and a source vertex $s$
- **Output:**
  - Visits the vertices in order of their distance from $s$
  - Find the shortest distance from $s$ to each reachable vertex
Breadth-First search: pseudocode: Basic version

- During execution of the algorithm, the vertices are in one of the three following states:
  - Undiscovered (White)
  - Discovered (Gray)
  - Fully-explored (Black)

```
Initialization: mark all vertices undiscovered (white)
BFS(G, s)
    mark s "discovered"
    Q = [s]
    while Q not empty
        u = dequeue(Q)
        for each neighbor v of u:
            if (v is undiscovered):
                mark v discovered
                enqueue(Q, v)
    mark u fully-explored
```
BFS Example
BFS Example
BFS Example

Queue

w  r

1  1
BFS Example

Queue

r  t  x

1  2  2
BFS Example

Queue

<table>
<thead>
<tr>
<th>t</th>
<th>x</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
BFS Example

Queue: x v u

2 2 3
BFS Example

Queue

\[
\begin{array}{c}
v & u & y \\
2 & 3 & 3 \\
\end{array}
\]
BFS Example

Queue:

```
<table>
<thead>
<tr>
<th>u</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```
BFS Example
BFS Example
BFS Example

Distance 1: s, w
Distance 2: r, v, t, x
Distance 3: u, y
BFS: Basic version: Runtime Analysis: **Naive Analysis**

Initialization: mark all vertices undiscovered

\[
\text{BFS}(G, s) \\
\quad \text{mark } s \text{ "discovered"} \\
\quad Q = [s] \\
\quad \text{while } Q \text{ not empty} \\
\quad \quad u = \text{dequeue}(Q) \\
\quad \quad \text{for each neighbor } v \text{ of } u: \\
\quad \quad \quad \text{if (} v \text{ is undiscovered):} \\
\quad \quad \quad \quad \text{mark } v \text{ discovered} \\
\quad \quad \quad \quad \text{enqueue}(Q, v) \\
\quad \quad \text{mark } u \text{ fully-explored}
\]

Runtime: \(O(V^2)\)
BFS: Basic version: Runtime Analysis: **Aggregate Analysis**

**Initialization:** mark all vertices undiscovered

BFS(G, s)

mark s "discovered"

Q = [s]

while Q not empty

u = dequeue(Q)

for each neighbor v of u:

if (v is undiscovered):

mark v discovered

enqueue(Q, v)

mark u **fully-explored**

- Each vertex is enqueued at most once (when it is discovered)
- When a vertex u is dequeued, the for loop is executed for \(\text{deg}(u)\) iterations
- So the total time complexity is \(O(n + \sum_{u \in V} \text{deg}(u)) = O(n+m)\)
BFS: Finding the path from \( s \) to \( v \)

- **How to trace back a path from \( s \) to \( v \)**
  - We can add an array `parent[v]`
  - When a vertex \( v \) is discovered within the for loop of vertex \( u \), then we set `parent[v] = u`.
  - Now to trace out a path \( v \) to \( s \), we just need to write a for loop that starts from \( v \) and keep going to its parent until we reach vertex \( s \)
  - For all vertices \( v \) reachable from \( s \), the edges \((v, parent[v])\) form a tree, called the **BFS tree**

- **Also useful to store level (distance)**
BFS: Detailed Version

\[
\text{BFS}(G, s)
\]
\[
Q = \emptyset
\]
for each node \(u\) in \(G\):
\[
dist[u] = \infty
\]
\[
color[u] = \text{WHITE}
\]
\[
pred[u] = \text{NULL}
\]
\[
dist[s] = 0
\]
\[
color[s] = \text{GRAY} \quad \#\text{mark } s \text{ "discovered"}
\]
enqueue\((Q, s)\)
while \(Q\) not empty
\[
u = \text{dequeue}(Q)
\]
for each neighbor \(v\) of \(u\):
\[
\text{if } \text{color}[v] == \text{WHITE}: \quad \#(u\text{ is undiscovered})
\]
\[
color[v] = \text{GRAY} \quad \#\text{mark } u \text{ discovered}
\]
\[
distance[v] = dist[u] + 1
\]
\[
pred[v] = u
\]
\[
\text{enqueue}(Q, v)
\]
\[
color[u] = \text{BLACK} \quad \#\text{mark } u \text{ fully-explored}
\]
BFS: Shortest paths

- $\delta(s, v) = \text{shortest path distance from } s \text{ to } v$
  - Minimum number of edges in any path from vertex $s$ to vertex $v$
- $\delta(s, v) = \infty$
  - If there is no path from $s$ to $v$
- A shortest path path from $s$ to $v$ has length $\delta(s, v)$
- **BFS correctly computes the shortest path distances**
BFS: Proof of correctness

- To prove correctness of BFS,
  - Upon termination of BFS, \( \text{dist}[v] = \delta(s, v) \) for all \( v \) in \( V \)
  - Therefore, all the vertices that are reachable from \( s \) must be discovered otherwise \( \text{dist}[v] = \infty > \delta(s, v) \).
BFS Properties

- **Lemma 1.** Let $G = (V, E)$ be a directed or undirected graph, and $s$ be an arbitrary vertex. Then, for any edge $(u, v)$ in $E$:
  - $\delta(s, v) \leq \delta(s, u) + 1$

- **Proof:**
  - If $u$ is reachable from $s$, then so is $v$ (since they are two ends of an edge)
    - the shortest path from $s$ to $v$ cannot be longer than the shortest path from $s$ to $u$ followed by the edge $(u, v)$ → inequality holds
  - If $u$ is not reachable from $s$, then $\delta(s, u) = \infty$ and the inequality holds
BFS Properties

- **Lemma 2.** The value dist[v] computed by BFS satisfies the following inequality:
  - \( \text{dist}[v] \geq \delta(s, v) \)
- **Proof.** by induction on the number of enqueue operations
- **Induction hypothesis:** \( \text{dist}[v] \geq \delta(s, v) \) for all \( v \) in \( V \)
- **Base case:** In the first enqueue operation
  - \( \text{dist}[s] = 0 = \delta(s, s) \)
  - \( \text{dist}[v] = \infty \geq \delta(s, v) \) for all \( v \) in \( V\{-s\} \)
- **Induction step:**
  - assume lemma is true for the first \((n-1)\) enqueue operations. Now, in the \(n\)-th enqueue operation: vertex \( v \) is discovered and \( v \) is white. The vertex \( v \) is discovered after dequeuing \( u \). The algorithm sets:
    - \( \text{dist}[v] = \text{dist}[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v) \)
      - I.H.
      - Lemma 1
Lemma 3. At any time, the distance values of the vertices on the queue have the following property
- $\text{dist}[i] \leq \text{dist}[i+1]
- \text{dist}[\text{end}] \leq \text{dist}[\text{front}] + 1$

In other words at any time the queue looks like this

Proof. By induction
- See CLRS

Implications:

- **Corollary 1.** If vertex $u$ is enqueued before $v$ then $\text{dist}[u] \leq \text{dist}[v]$
- **Corollary 2.** If vertex $u$ is dequeued before $v$ then $\text{dist}[u] \leq \text{dist}[v]
BFS Properties: Proof of Lemma 3

- **Proof**: by induction on the number of queue operation. Let’s prove for dequeue operation.

  - **Base-case**: When the queue contains only s and we dequeue s, the queue becomes empty and the property holds
  - **Induction hypothesis**: The property is true for the first n-1 dequeue operations
    - \( \text{dist}[i] \leq \text{dist}[i+1] \)
    - \( \text{dist}[\text{end}] \leq \text{dist}[\text{front}] + 1 \)
  - **Induction step**: After the n-th dequeue, we need to show:
    - \( \text{dist}[i] \leq \text{dist}[i+1] \) (true by I.H.)
    - \( \text{dist}[\text{end}] \leq \text{dist}[\text{newfront}] + 1 \)
    - \( \text{dist}[\text{end}] \leq \text{dist}[\text{front}] + 1 \) (I.H.)
    - \( \leq \text{dist}[\text{front}+1] + 1 = \text{dist}[\text{newfront}] + 1 \)

- Similarly you can prove by induction on enqueue operation
Theorem 1. BFS(G, s) discovers every vertex v in V that is reachable from s, and upon termination dist[v] = δ(s, v) for all v in V.

Proof. By contradiction. Assume ∃ v s.t. dist[v] ≠ δ(s, v). Also, assume v is the vertex with minimum δ(s, v) that receives an incorrect distance. We prove these assumptions leads to a contradiction.

1. dist[v] ≥ δ(s, v) (according to lemma 2). Therefore if dist[v] ≠ δ(s, v) then dist[v] > δ(s, v)
2. Vertex v must be reachable from s, if not δ(s, v) = ∞ ≥ dist[v]
3. Let u be the vertex before v on the shortest path from s to v,
   - δ(s, v) = δ(s, u) + 1
   - δ(s, u) < δ(s, v)
   - dist[u] = δ(s, u) because of how we chose v
     - dist[v] > δ(s, v) = δ(s, u) + 1 = dist[u] + 1 → dist[v] > dist[u]+1
4. When BFS chooses to dequeue u from Q, vertex v is in one of the following states:
   - WHITE: dist[v] = dist[u] + 1 → contradiction
   - BLACK: It was already removed from the queue which means dist[v] ≤ dist[u] → contradiction
     - Recall: if v is dequeued before u then dist[v] ≤ dist[u]
   - GRAY
     - It was painted gray upon dequeuing some vertex w
     - w was removed from Q before u and dist[v] = dist[w] + 1
     - dist[w] ≤ dist[u] (was removed from queue before u)
     - dist[v] = dist[w] + 1 ≤ dist[u] + 1 → contradiction
BFS properties

- **Theorem 2.** For any vertex $v$ reachable from $s$, one of the shortest path from $s$ to $v$ is a shortest path from $s$ to $\text{pred}[v]$ followed by the edge $(\text{pred}[v], v)$

- **Proof of the second part.**
  - According to Theorem 1, $\text{dist}[v] = \delta(u, v)$ for all $v$ in $V$.
  - All the vertices reachable from $s$ must be discovered otherwise $\text{dist}[v] = \infty > \delta(s, v)$.
  - If $\text{pred}[v] = u$ → $\text{dist}[v] = \text{dist}[u] + 1$ (That is what the BFS algorithm does)
  - The shortest path from $s$ to $v = $ shortest path from $s$ to $\text{pred}[v]$ and the edge $(\text{pred}[v], v)$
BFS Application

● BFS can be used to check whether a graph is connected or not
  ○ Checking whether all vertices are marked discovered

● The connected component containing s
  ○ by returning all the vertices which are discovered

● Whether there is a path from s to v
  ○ Checking whether v is discovered

● Exercises:
  ○ Enhance BFS to find all connected components in time $O(n + m)$
    ■ If a graph is not connected
  ○ Use BFS to find if a connected graph has a cycle.
  ○ Prove that if $(u, v)$ in $E$ then $\text{level}(u)$, $\text{level}(v)$ differ by 0 or 1.
Breadth-First Tree

- BSF builds a breadth-first tree:
  - A tree where the path from s to every node is the shortest path
Breadth-First Tree

● Subgraph $G'=(V', E')$ is generated after running the algorithm BFS
  ○ $V' = \{v \in V: \text{pred}[v] \neq \text{NULL}\} \cup \{s\}$
    ■ $V'$ consists of vertices in $V$ which are reachable from $s$, since the algorithm sets $\text{pred}[v] = u$ if and only if $(u, v)$ is an edge in $E$ and $v$ is reachable from $s \rightarrow G'$ is a connected graph
  ○ $E' = \{(\text{pred}[v], v): v \in V' - \{s\}\}$

● $G'$ is a Tree since it is connected and $|E'| = |V'| - 1$
● Since $G'$ is a tree there is a unique simple path from $s$ to every vertex in $V'$
● According to Theorem 2, this path is the shortest path.
  ○ How? You can prove that using induction and using Theorem 2