CS 313
- Kangmei Yang
Graph

\[ G = (V, E) \]
- Consists of a set of vertices and a set of edges
- Edge is a pair of vertices. Also called arc.

- Represents many real-life data, such as airline flights, relationships, traffic flow, or course prerequisites.

- Tree is a special form of graph

**Undirected**

![Undirected Graph](image)

**Directed - Digraph**

![Directed Graph](image)

**Unordered edge**

**Ordered edge weight/cost**
Undirected Graph

Unordered edge
-Edge (v1, v2), Edge (v2, v1), v2 is adjacent to v1 and v1 is adjacent to v2
-Edge (v2, v3), Edge (v3, v2),
-Edge (v2, v4), Edge (v4, v2),
-Edge (v3, v4), Edge (v4, v3),
-Edge (v1, v2), Edge (v2, v1)

Path – sequences of vertices connected by edges

Length – number of edges on a path

Vertex a is reachable to vertex b if there is a path from a to b
Directed Graph

Ordered edge
- Edge \((v_1, v_2)\), 
  \(v_2\) is adjacent to \(v_1\) but 
  \(v_1\) is NOT adjacent to \(v_2\)
- Edge \((v_3, v_2)\), Edge \((v_4, v_2)\),
- Edge \((v_3, v_4)\), Edge \((v_4, v_3)\),
- Edge \((v_1, v_4)\), Edge \((v_4, v_1)\),

- An edge is incident to vertex \(a\), if vertex \(a\) is its endpoint. (Unordered edge incidents to both vertices it connects)
- Degree of a vertex is the number of edges incident to it
- Cycle – path of at least 1 edge that starts and ends at the same vertex. (Not considering a vertex loop to itself)
- Directed acyclic graph (DAG) contains no cycles
- Weighted graph has a weight/cost with each edge
Graph

**Undirected**

- Each vertex has a path to every other vertex

**Directed - Digraph**

- Connected
- Strongly Connected
- Weakly connected
  - underlying graph is connected

**Complete Graph**

- there’s an edge between every pair of vertices
Graph Representation

Two way to represent a graph
- Adjacency Matrix
  2D array, where $A[v1][v2]$ represents Edge $(v1, v2)$
  - Boolean array represents if there’s an edge
  - Int array stores the weight of the edge, $\infty/-1/-\infty$ represents no edge
- Adjacency list
  For each vertex, it contains a list of all adjacent vertices, if it’s weighted graph, this information is stored in the list as well.

Adjacency Matrix (Dense)

\[
\begin{array}{cccc}
  v1 & v2 & v3 & v4 \\
  v1 & -1 & 3 & -1 & 2 \\
  v2 & -1 & -1 & -1 & -1 \\
  v3 & -1 & 2 & -1 & 1 \\
  v4 & 3 & 2 & 3 & -1 \\
\end{array}
\]

Adjacency List (Sparse)

\[
\begin{align*}
  v1 & \rightarrow \{v2, 3, v4, 2\} \\
  v2 & \rightarrow \{v3, 2\} \\
  v3 & \rightarrow \{v2, 2\} \\
  v4 & \rightarrow \{v1, 3\}
\end{align*}
\]
Graph Representation

Graph has V vertices and E edges

Space Comparison
- Adjacency Matrix: \( O(V^2) \)
- Adjacency List: \( O(V+E) \) Linear time

Note: A complete graph has \( V^2 \) edges

It’s more efficient to use adjacency matrix if the graph is dense, use adjacency list otherwise.

It’s harder to look if an edge is in the graph represented in adjacency list (need iterate through the list). Modify-Map/Store the list for each vertex

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Adjacency Matrix (Dense)

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>V4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td>2</td>
<td>V4</td>
<td>1</td>
</tr>
<tr>
<td>V1</td>
<td>3</td>
<td>V2</td>
<td>2</td>
</tr>
<tr>
<td>V3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph Traversal

- Systematic way to visit each vertex in the graph.
- Check whether there’s a path between two vertices and whether the graph contains a cycle.

Two common traversal:
Depth-first search
- explore each path as deeply as possible before exploring next path
Pre-order traversal is one example of depth-first search

Breadth-first search
- explore vertices that closer to the starting vertex before move on to farther vertices
Level-order traversal is one example of breadth-first search
Depth-First Search

- An array to store whether the vertex has been visited
- An array to store the vertex that current vertex came from (adjacent to)

DFS(v)

\[
\text{visited}[v] = \text{true} \\
\text{for } (w: \text{adj_list}[v]) \text{ if } (! \text{visited}[w]) \\
\text{parent}[w] = v \\
\text{DFS}(w)
\]

DFS(v1): visit[v1] = T

\[
\text{adj}[v1]: v2, \text{parent}[v2] \\
- \text{DFS}(v2): \text{visit}[v2] = T \\
\text{adj}[v2]: \text{EMPTY} \\
\text{adj}[v1]: v4, \text{parent}[v2] \\
- \text{DFS}(v4): \text{visit}[v4] = T \\
\text{adj}[v4]: v1-visited \\
\text{adj}[v4]: v2-visited \\
\text{adj}[v4]: v3, \text{parent}[v3] \\
- \text{DFS}(v3): \text{visit}[v3] = T \\
\text{adj}[v3]: v2-visited
\]
Breadth-First Search

- Array: if the vertex has been visited
- Array: store vertex came from
- A queue: store waiting vertices

BFS(v)

visited[v] = true
start queue, enqueue v
while queue not empty: dequeue v
for (w: adj[v]) if !visited[w]
parent[w] = v
visited[w] = true
queue enqueue w

BFS(v1):
visit[v1] = T, enqueue v3
dequeue, adj[v1]: v2
prnt[v2], vst[v2] = T, enqueue adj[v1]: v4
prnt[v4], vst[v4] = T, enqueue adj[v2]: v3
dequeue, adj[v4]: v3
prnt[v3], vst[v3] = T, enqueue, adj[v3]
Graph Traversal

Depth First Search
v1
v2
v3
v6
v4
v5

Breadth First Search
v1
v2,v4
v1,v3,v4,v5
v1,v2,v3
v6
v2,v3,v4,v6
v1,v4,v5

Visited
F
F
F
F
F

Parent
v1
v2
v3
v4
v5
v6