Topic 9:
Basic Graph Alg.
- Representations
- Basic traversal algorithms
- Topological sort
What Is A Graph

- Graph \( G = (V, E) \)
  - \( V \): set of nodes
  - \( E \): set of edges

- Example:
  - \( V = \{ a, b, c, d, e, f \} \)
  - \( E = \{(a, b), (a, d), (a, e), (b, c), (b, d), (b, e), (c, e), (e,f)\} \)
- Un-directed graph
  - $E \leq \binom{V}{2}$
- Directed graph
  - $E \leq V^2$
Un-directed graphs
**Vertex Degree**

- \( \text{deg}(v) = \text{degree of } v = \# \text{ edges incident on } v \)

- Lemma: \( \sum_{v_i \in V(G)} \text{deg}(v_i) = 2|E| \)
Some Special Graphs

- Complete graph
- Path
- Cycle
- Planar graph
- Tree
Representations of Graphs

- **Adjacency lists**
  - Each vertex $u$ has a list, recording its neighbors
    - i.e., all $v$’s such that $(u, v) \in E$
  - An array of $V$ lists
    - $V[i].degree = \text{size of adj list for node } v_i$
    - $V[i].AdjList = \text{adjacency list for node } v_i$
Adjacency Lists

- For vertex $v \in V$, its adjacency list
  - has size: $\text{deg}(v)$
  - decide whether $(v, u) \in E$ or not in time $\Theta(\text{deg}(v))$

- Size of data structure (space complexity):
  - $\Theta(|V| + |E|) = \Theta(V + E)$
Adjacency Matrix

- $V \times V$ matrix $A$
  - $A[i, j] = 1$ if $(v_i, v_j)$ is an edge
  - Otherwise, $A[i, j] = 0$

![Graph](image)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency Matrix

- Size of data structure:
  - $\Theta(V \times V)$

- Time to determine if $(v, u) \in E$:
  - $\Theta(1)$

- Though larger, it is simpler compared to adjacency list.
Sample Graph Algorithm

- Input: Graph $G$ represented by adjacency lists

```
Func(G)
1  k ← 0;
2  foreach vertex $v_i \in V(G)$ do
3    foreach edge $(v_i, v_j)$ incident on $v_i$ do
4      k ← k + 1;
5    end
6  end
7  return (k);
```

Running time: $\Theta(V + E)$
Connectivity

- A *path* in a graph is a sequence of vertices \((u_1, u_2, \ldots, u_k)\) such that there is an edge \((u_i, u_{i+1})\) between any two adjacent vertices in the sequence.

- Two vertices \(u, w \in V(G)\) are *connected* if there is a path in \(G\) from \(u\) to \(w\).
  - We also say that \(w\) is *reachable* from \(u\).

- A graph \(G\) is *connected* if every pair of nodes \(u, w \in V(G)\) are connected.
Connectivity Checking

- How to check if the graph is connected?
- One approach: Graph traversal
  - BFS: breadth-first search
  - DFS: depth-first search
BFS: Breadth-first search

- **Input:**
  - Given graph $G = (V, E)$, and a source node $s \in V$

- **Output:**
  - Will visit all nodes in $V$ reachable from $s$
  - For each $v \in V$, output a value $v.d$
    - $v.d = \text{distance (smallest # of edges) from } s \text{ to } v.$
    - $v.d = \infty$ if $v$ is not reachable from $s$. 
Intuition

- Starting from source node $s$,
  - Spread a wavefront to visit other nodes
  - First visit all nodes one edge away from $s$
  - Then all nodes two edges away from $s$
  - ...

Need a data-structure to store nodes to be explored.
A node can be:
- un-discovered
- discovered, but not explored
- explored (finished)

\( v.d \):
- is set when node \( v \) is first discovered.

Need a data structure to store discovered but un-explored nodes
- FIFO ! Queue
Pseudo-code

Use adjacency list representation

Time complexity: $\Theta(V+E)$
Correctness of Algorithm

- A node not reachable from \( s \) will not be visited
- A node reachable from \( s \) will be visited
- \( v, d \) computed is correct:
  - Intuitively, if all nodes \( k \) distance away from \( s \) are in level \( k \), and no other nodes are in level \( k \),
  - Then all nodes \( (k+1) \)-distance away from \( s \) must be in level \( (k+1) \).
- Rigorous proof by induction
BFS tree

- A node \( v \) is the parent of \( u \) if
  - \( u \) was first discovered when exploring \( v \)

- A BFS tree \( T \)
  - Root: source node \( s \)
  - Nodes in level \( k \) of \( T \) are distance \( k \) away from \( s \)
Connectivity-Checking

procedure QueryConnected(G)
1. foreach vertex $v_i$ of $G$ do
2.     $v_i$.mark $\leftarrow$ NotVisited;
3. end
4. BFS($V$, $E$, $v_1$);
5. foreach vertex $v_i$ of $G$ do
6.     if ($v_i$.mark = NotVisited) then
7.         return false;
8.     end
9. end
10. return true;

Time complexity: $\Theta(V+E)$
Summary for BFS

- Starting from source node $s$, visits remaining nodes of graph from small distance to large distance

- This is one way to traverse an input graph
  - With some special property where nodes are visited in non-decreasing distance to the source node $s$.
  - Return distance between $s$ to any reachable node in time $\Theta(|V| + |E|)$
DFS: Depth-First Search

- Another graph traversal algorithm
- BFS:
  - Go as broad as possible in the algorithm
- DFS:
  - Go as deep as possible in the algorithm
Example

- Perform DFS starting from $v_1$
- What if we add edge $(v_1, v_8)$
DFS

procedure DFS(G, i)

/* Depth first search from vertex v_i */

1. $G.V[i].mark \leftarrow$ Visited;

2. foreach edge $(i, j)$ incident on vertex $i$ do
   3. if $(G.V[j].mark = \text{NotVisited})$ then
   4.     DFS $(G, j)$;
   5. end
5. end

- Time complexity
  - $\Theta(V+E)$
Depth First Search Tree

- If \( v \) is discovered when exploring \( u \)
  - Set \( v.parent = u \)
- The collection of edges
  - \( \{(v.parent, v)\} \) form a tree,
  - called Depth-first search tree.
procedure DFStree(G, i)
 /* Depth first search from vertex \( v_i \) */
 1 \( G.V[i].\text{mark} \leftarrow \text{Visited}; \)
 2 \( \text{foreach edge} \ (i,j) \ \text{incident on vertex} \ i \ \text{do} \)
      \( \text{if} \ (G.V[j].\text{mark} = \text{NotVisited}) \ \text{then} \)
      \( G.V[j].\text{parent} \leftarrow i; \)
      \( \text{DFStree} \ (G,j); \)
 3 \( \text{end} \)
 7 \( \text{end} \)
Example

- Perform DFS starting from $v_1$
procedure QueryConnected(G)

1  foreach vertex \( v_i \) of \( G \) do
2      \( v_i\).mark ← NotVisited;
3  end

4  DFS(\( G,1 \));

5  foreach vertex \( v_i \) of \( G \) do
6      if (\( v_i\).mark = NotVisited) then return false;
7  end

8  return true;
procedure DFS-All(G)

    /* Use Depth first search to traverse all nodes \(v_i\) in \(G\) */

1. foreach vertex \(v_i\) of \(G\) do
2.     \(G.V[i].mark \leftarrow \text{NotVisited};\)
3. end

4. for \(i \leftarrow 1\) to \(V\) do
5.     if \((G.V[i].mark = \text{NotVisited()})\) then
6.         DFS \((G,i)\);
7.     end
8. end
Remarks

- DFS($G, k$)
  - Another way to compute all nodes reachable to the node $v_k$.
- Same time complexity as BFS.
- There are nice properties of DFS and DFS tree that we are not reviewing in this class.
Directed Graphs
**Un-directed graph**
- $E \leq \binom{V}{2}$

**Directed graph**
- Each edge $(u, v)$ is directed from $u$ to $v$
- $E \leq V^2$
Vertex Degree

- \( \text{indeg}(v) = \# \text{ edges of the form } (u, v) \)
- \( \text{outdeg}(v) = \# \text{ edges of the form } (v, u) \)

Lemma: \( \sum_{v_i \in V(G)} \text{indeg}(v_i) = |E| \)

Lemma: \( \sum_{v_i \in V(G)} \text{outdeg}(v_i) = |E| \)
Representations of Graphs

- **Adjacency lists**
  - Each vertex $u$ has a list, recording its neighbors
    - i.e., all $v$’s such that $(u, v) \in E$
  - An array of $V$ lists
    - $V[i].degree$ = size of adj list for node $v_i$
    - $V[i].AdjList$ = adjacency list for node $v_i$
Adjacency Lists

- For vertex $v \in V$, its adjacency list
  - has size: $\text{outdeg}(v)$
  - decide whether $(v, u) \in E$ or not in time $O(\text{outdeg}(v))$

- Size of data structure (space complexity):
  - $\Theta(V+E)$
Adjacency Matrix

- $V \times V$ matrix $A$
  - $A[i, j] = 1$ if $(v_i, v_j)$ is an edge
  - Otherwise, $A[i, j] = 0$
Adjacency Matrix

- Size of data structure:
  - $\Theta (V \times V)$

- Time to determine if $(v, u) \in E$:
  - $O(1)$

- Though larger, it is simpler compared to adjacency list.
Sample Graph Algorithm

- Input: Directed graph $G$ represented by adjacency list

```
Func(G)
1  k ← 0;
2  foreach vertex $v_i \in V(G)$ do
3      foreach edge $(v_i, v_j)$ incident on $v_i$ do
4          k ← k + 1;
5      end
6  end
7  return (k);
```

Running time: $O(V + E)$
Connectivity

- A *path* in a graph is a sequence of vertices 
  \((u_1, u_2, \ldots, u_k)\) such that there is an edge 
  \((u_i, u_{i+1})\) between any two adjacent vertices in the sequence.

- Given two vertices \(u, w \in V(G)\), we say that \(w\) *is reachable* from \(u\) if there is a path in \(G\) from \(u\) to \(w\).

  - Note: \(w\) is reachable from \(u\) *DOES NOT* necessarily mean that \(u\) is reachable from \(w\).
Reachability Test

- How many (or which) vertices are reachable from a source node, say $v_1$?
BFS and DFS

- The algorithms for BFS and DFS remain the same
  - Each edge is now understood as a directed edge

- BFS(V,E, s):
  - visits all nodes reachable from s in non-decreasing order
BFS

- Starting from source node $s$,
  - Spread a wavefront to visit other nodes
  - First visit all nodes one edge away from $s$
  - Then all nodes two edges away from $s$
  - ...

![Graph demonstrating BFS]

0 1 2 3

\( s \) 1 2 3

\( a \) 1 2 3

\( b \) 3

\( c \) 1

\( e \) 2

\( f \) 3

\( g \) 2

\( h \) 3

\( i \) 3
Pseudo-code

Use adjacency list representation

Time complexity: $\Theta(V+E)$
Number of Reachable Nodes

procedure NumReachable(G, v_k)
\begin{align*}
&1 \text{ foreach vertex } v_i \text{ of } G \text{ do} \\
&2 \quad v_i.\text{mark} \leftarrow \text{NotVisited}; \\
&3 \text{ end} \\
&4 \text{ BFS}(V,E, v_k); \\
&5 \text{ count} \leftarrow 0; \\
&6 \text{ foreach vertex } v_i \text{ of } G \text{ do} \\
&7 \quad \text{if } (v_i.\text{mark} = \text{Visited}) \text{ then} \\
&8 \quad \quad \text{count} \leftarrow \text{count} + 1; \\
&9 \quad \text{end} \\
&10 \text{ end} \\
&11 \text{ return count;}
\end{align*}

Compute # nodes reachable from $v_k$

Time complexity: $\Theta(V+E)$
DFS: Depth-First Search

- Similarly, DFS remains the same
  - Each edge is now a directed edge
- If we start with all nodes unvisited,
  - Then DFS(G, \( k \)) visits all nodes reachable to node \( v_k \)
- BFS from previous NumReachable() procedure can be replaced with DFS.
More Example

- Is $v_1$ reachable from $v_{12}$?
- Is $v_{12}$ reachable from $v_1$?
procedure IsReachable(G, k, q)
1 foreach vertex v_i of G do
2 \[ v_i \text{.mark} \gets \text{NotVisited}; \]
3 end
4 DFS(G, k);
5 if (v_q\text{.mark} = \text{Visited}) then return true;
6 else return false;

- DFS above can be replaced with BFS
Topological Sort
Directed Acyclic Graph

- A directed cycle is a sequence \((u_1, u_2, \ldots, u_k, u_1)\) such that there is a directed edge between any two consecutive nodes.

- DAG: directed acyclic graph
  - Is a directed graph with no directed cycles.
A topological sort of a DAG $G = (V, E)$

- A linear ordering $A$ of all vertices from $V$
- If edge $(u,v) \in E \Rightarrow A[u] < A[v]$

Diagram:
- undershorts
  - pants
  - belt
    - shirt
      - tie
    - jacket
  - shoes
  - socks
  - watch
Another Example

Why requires DAG?

Is the sorting order unique?
A topological sorted order of graph $G$ exists if and only if $G$ is a directed acyclic graph (DAG).
Question

- How to topologically sort a given DAG?

Intuition:

- Which node can be the first node in the topological sort order?
- A node with in-degree 0!
- After we remove this, the process can be repeated.
Example

- undershorts
- pants
- belt
- shirt
- tie
- jacket
- socks
- shoes
- watch
procedure TopologicalSort(G)
    /* Report vertices of G in topologically sorted order */
    1 foreach vertex vᵢ of G do
        /* S is a stack of vertices. */
        2 if (indeg(vᵢ) = 0) then S.Push(vᵢ);
    3 end
    4 while (S is not empty) do
        5 vᵢ ← S.Pop();
        6 Report vᵢ;
        7 foreach edge (vᵢ, vⱼ) incident on vᵢ do
            8 Delete (vᵢ, vⱼ) from G;
            9 if (indeg(vⱼ) = 0) then S.Push(vⱼ);
        10 end
    11 end
procedure TopologicalSort(G)
   /* Report vertices of G in topologically sorted order */
   1 foreach vertex \( v_i \) of G do
      /* \( S \) is a stack of vertices. */
      2 inCount[\( v_i \)] \( \leftarrow \) indeg(\( v_i \));
      3 if (inCount[\( v_i \)] = 0) then \( S.Push(v_i) \);
   4 end
   5 while (\( S \) is not empty) do
      6 \( v_i \) \( \leftarrow \) \( S.Pop() \);
      7 Report \( v_i \);
      8 foreach edge \( (v_i, v_j) \) incident on \( v_i \) do
         9 inCount[\( v_j \)] \( \leftarrow \) inCount[\( v_j \)] - 1;
         10 if (inCount[\( v_j \)] = 0) then \( S.Push(v_j) \);
      11 end
   12 end
- Time complexity
  - $\Theta(V+E)$

- Correctness:
  - What if the algorithm terminates before we finish visiting all nodes?
  - Procedure `TopologicalSort(G)` outputs a sorted list of all nodes if and only if the input graph $G$ is a DAG
    - If $G$ is not DAG, the algorithm outputs only a partial list of vertices.
Remarks

- Other topological sort algorithm by using properties of DFS
Analyzing graph algorithms

- Adjacency list representation or
- Adjacency matrix representation
- Edge-weighted un-directed graph $G = (V, E)$ and edge weight function $w: E \rightarrow R$
  - E.g, road network, where each node is a city, and each edge is a road, and weight is the length of this road.
Example 1

Assume G is represented by adjacency list

Q is priority-queue implemented by min-heap

```
Func3(G)
    /* G = edge weighted graph */
    foreach vertex v_i of G do
        k ← i;
        foreach edge (v_i, v_j) incident on v_i do
            if (k = i) or (weight(v_i, v_k) > weight(v_i, v_j)) then
                k ← j;
            end
        end
    end
    Q.Insert((v_i, v_k), weight(v_i, v_k));
    end
while Q is not empty do
    (v_i, v_j) ← Q.ExtractMin();
    Report edge (v_i, v_j);
end
```
Example 2

Assume G is represented by adjacency matrix

Func(G)
/* G = edge weighted graph */

1. foreach vertex v_i of G do
2.     k ← i;
3.     foreach vertex v_j of G do
4.         if A[i, j] == 1 then
5.             if (k = i) or (weight(v_i, v_k) > weight(v_i, v_j)) then
6.                 k ← j;
7.             end
8.         end
9.     end
10.    Q.Insert((v_i, v_k), weight(v_i, v_k));
11. end
12. while Q is not empty do
13.     (v_i, v_j) ← Q.ExtractMin();
14.     Report edge (v_i, v_j);