Motivation

- We will now see how several problems in Graph Theory can be modeled and solved using Graph algorithms
- Many real life problems can be modeled with graphs
- We will give algorithms that solve some common graph problems
- We will see how choice of data structures is important in increasing the performance of algorithms
Definitions

A graph $G = (V,E)$ consists of a set of vertices, $V$, and a set of edges, $E$.

Each edge is a pair $(u,w)$, where $u,w \in E$.

Edges are also called as arcs.

If the pair is ordered, then the graph is directed (digraph), otherwise it is undirected graph.

Vertex $w$ is adjacent to $v$ if and only if $(v,w) \in E$.

In an undirected graph with edge $(v,w)$, and hence $(w,v)$, $w$ is adjacent to $v$, and $v$ is adjacent to $w$.

An edge may have a third component which is called weight or cost.

Example

Directed Graph $G$

![Directed Graph G Diagram]

$G = (V,E)$

$V = \{1,2,3,4,5,6,7\}$

$E = \{(1,2),(1,4),(1,3),(2,4),(2,5),(3,6),(4,3),(4,6),(4,7),(5,4),(5,7),(7,6)\}$

3 is adjacent to 1, but 1 is not adjacent to 3.
Definitions

A path in a graph is a sequence of vertices \( w_1, w_2, \ldots, w_N \) such that \((w_i, w_{i+1}) \in E\) for \(1 \leq i < N\).

The length of such a path is the number of edges on the path, which is equal to \(N-1\).

There can be a path from a vertex to itself. If this path contains no edges, then the path length is 0.

If graph contains a path from a vertex \(v\) to itself, then we say that the graph contains a loop.

A simple path is a path that all vertices are distinct, except that the first and last vertex could be the same.

A cycle in a directed graph is a path at least 1 such that \(w_1 = w_N\). For undirected graphs, we requires that the edges are distinct.

A directed graph is acyclic if it has no cycles. Such a graphs is also referred as DAG.

An undirected graph is connected if there is a path from every vertex to every other vertex.

If such a graph is directed, then it is said that it is strongly connected.

If a directed graph is not strongly connected, but the underlying graph (without directions) is connected, then the graph is said to be weakly connected.

A complete graph is a graph in which there is an edge between every pair of vertices.
Representation of Graphs

- We will consider representation of directed graphs. Undirected graphs are similarly represented.
- Support we number the vertices, starting from one.
- There are two methods
  1. *Adjacency matrix* representation
  2. *Adjacency list* representation

Some Common Graph Problems and Algorithms

- Topological Sort
- Shortest-Path Algorithms
- Network Flow Problems
- Minimum Spanning Tree
- Depth First Search and Applications
We will represent the graph above as an example.

**Adjacency Matrix Representation**

- Use a two-dimensional array A.
- For each edge \((u,v)\), set \(A[u][v]\) to true, otherwise to false.
- If the edge has a weight (cost) associated with it, then we set the \(A[u][v]\) equal to the weight.
  - Use a very large or very small weight as a *sentinel* to indicate nonexistent edges.
- Space requirement \(O(|V|^2)\)
- Good if the graph is *dense*. 
### Adjacency Matrix Representation

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

### Adjacency List Representation

- For each vertex, keep a list of all adjacent vertices.
- Space requirement is $O(|E| + |V|)$
  - Linear in the size of the graph.
- Standard way to represent graphs
- Undirected graphs can be similarly represented; each edge $(u,v)$ appears in two lists.
  - Space usage doubles.
**Adjacency List Representation**

![Adjacency List Diagram]

**Some Common Graph Problems and Algorithms**

- Topological Sort
- Shortest-Path Algorithms
- Network Flow Problems
- Minimum Spanning Tree
- Depth First Search and Applications
A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from \( v_i \) to \( v_j \), then \( v_j \) appears after \( v_i \) in ordering.

- A topological ordering is not possible if the graph contains cycles, since for two vertices \( v \) and \( w \) on a cycle, \( v \) precedes \( w \) and \( w \) precedes \( v \).

Ordering is not necessarily unique.

- \( v_1, v_2, v_5, v_4, v_3, v_7, v_6 \) (one ordering)
- \( v_1, v_2, v_5, v_4, v_7, v_3, v_6 \) (another ordering)
Topological Sort Algorithm Sketch

1. Find a vertex, \( v \), with no incoming edges.
2. Print this vertex \( v \); and remove it, along with all its edges, from the graph.
3. Repeat steps 1 and 2 until graph is empty.

Topological Sort Algorithm - Formally

- **Definition:**
  - *Indegree* of a vertex \( v \) is the number of edges in the form \((u,v)\).

- **Algorithm:**
  - Compute the indegrees of all vertices in the graph.
  - Read the graph into an adjacency list.
  - Apply the algorithm in the previous slide.
### Topological Sort Algorithm - Formally

```cpp
void Graph::topsort()
{
    Vertex v;     // vertex, v, that has indegree equal to 0
    Vertex w;     // vertex adjacent to v
    int counter;  // keeps the topological order number: 0,1,2,...

    for (counter = 0; counter < NUM_VERTICES; counter++)
    {
        v = findNewVertexOfDegreeZero();     // O(N)
        if (v == NOT_A_VERTEX)
            throw CycleFound();
        v.topNum = counter; // index in topological order
        for each w adhacent to v
            w.indegree--;     // O(N)
    }
}
```

- Running time = $O(|V|^2)$

### More efficient algorithm

- Keep the vertices which have indegree equal to zero in a separate box (stack or queue).
  - 1. Start with an empty box.
  - 2. Scan all the vertices in the graph
  - 3. Put vertices that have indegree equal to zero into the box.
  - 4. While the box (queue) is not empty
    - 4.1. Remove head of queue: vertex $v$.
    - 4.2. Print $v$
    - 4.3. Decrease the indegrees of all the vertices adjacent to $v$ by one.
    - 4.4. Go to step 4.
After Enqueue

After Dequeue

Print

Indegrees

\[
\begin{array}{c|c}
\text{v}_1 & 0 \\
\text{v}_2 & 1 \\
\text{v}_3 & 2 \\
\text{v}_4 & 3 \\
\text{v}_5 & 1 \\
\text{v}_6 & 3 \\
\text{v}_7 & 2 \\
\end{array}
\]

Updated Indegrees

\[
\begin{array}{c|c}
\text{v}_1 & 0 \\
\text{v}_2 & 0 \\
\text{v}_3 & 1 \\
\text{v}_4 & 2 \\
\text{v}_5 & 1 \\
\text{v}_6 & 3 \\
\text{v}_7 & 2 \\
\end{array}
\]
Updated Indegrees

| \(v_1\) | 0  |
|\(v_2\) | 0  |
|\(v_3\) | 1  |
|\(v_4\) | 1  |
|\(v_5\) | 0  |
|\(v_6\) | 3  |
|\(v_7\) | 2  |

After Enqueue After Dequeue Print

\(v_5\)

\(v_4\)

\(v_4\)

\(v_5\)

\(v_4\)

\(v_4\)

\(v_5\)

\(v_4\)

\(v_4\)
Updated Indegrees

\[
\begin{array}{c|c}
\nu_1 & 0 \\
\nu_2 & 0 \\
\nu_3 & 0 \\
\nu_4 & 0 \\
\nu_5 & 0 \\
\nu_6 & 2 \\
\nu_7 & 0 \\
\end{array}
\]

After Enqueue

\[
\begin{array}{c|c}
\nu_3 & \\
\nu_7 & \\
\end{array}
\]

After Dequeue

\[
\begin{array}{c|c}
\nu_7 & \\
\nu_3 & \\
\end{array}
\]

Print \( v_3 \)

---

Updated Indegrees

\[
\begin{array}{c|c}
\nu_1 & 0 \\
\nu_2 & 0 \\
\nu_3 & 0 \\
\nu_4 & 0 \\
\nu_5 & 0 \\
\nu_6 & 2 \\
\nu_7 & 0 \\
\end{array}
\]

After Enqueue

\[
\begin{array}{c|c}
\nu_7 & \\
\nu_3 & \\
\end{array}
\]

After Dequeue

\[
\begin{array}{c|c}
\nu_7 & \\
\nu_3 & \\
\end{array}
\]

Print \( v_3 \)
Pseudocode of Efficient Topological Sort Algorithm

```c
void Graph::topsort()
{
    Queue q(NUM_VERTICES);
    int counter = 0; //topological order of a vertex:1,2,3,...,NUM_VERTICES
    Vertex v, w;
    q.makeEmpty();
    for each vertex v
        if (v.indegree == 0)
            q.enqueue(v);
    while (!q.isEmpty())
    {
        v = q.dequeue();
        v.topNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
    if (counter != NUM_VERTICES)
        throw CycleFound();
}
```

Updated Indegrees

<table>
<thead>
<tr>
<th>vertex</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>0</td>
</tr>
<tr>
<td>v2</td>
<td>0</td>
</tr>
<tr>
<td>v3</td>
<td>0</td>
</tr>
<tr>
<td>v4</td>
<td>0</td>
</tr>
<tr>
<td>v5</td>
<td>0</td>
</tr>
<tr>
<td>v6</td>
<td>0</td>
</tr>
<tr>
<td>v7</td>
<td>0</td>
</tr>
</tbody>
</table>

After Enqueue | After Dequeue | Print
---|---|---
| v6 | .... | v6

Finished! Result: 1,2,5,4,3,7,6