1 Graph Problems

- The next couple of weeks we will discuss graph algorithms.
- You should already know about graphs
  - Today we will quickly review basic definitions and a few fundamental graph algorithms.

1.1 Definitions

- A graph $G = (V, E)$ consists of a finite set of vertices $V$ and a finite set of edges $E$.
  - Directed graphs: $E$ is a set of ordered pairs of vertices $(u, v)$ where $u, v \in V$
    
    ![Directed Graph](image)
    
    $V = \{1, 2, 3, 4, 5, 6\}$
    $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$
  - Undirected graph: $E$ is a set of unordered pairs of vertices $\{u, v\}$ where $u, v \in V$
    
    ![Undirected Graph](image)
    
    $V = \{1, 2, 3, 4, 5, 6\}$
    $E = \{\{1,2\}, \{1,5\}, \{2,5\}, \{3,6\}\}$

- Edge $(u, v)$ is incident to $u$ and $v$
- Degree of vertex in undirected graph is the number of edges incident to it.
- In (out) degree of a vertex in directed graph is the number of edges entering (leaving) it.
- A path from $u_1$ to $u_2$ is a sequence of vertices $< u_1=v_0, v_1, v_2, \ldots, v_k=u_2 >$ such that $(v_i,v_{i+1}) \in E$ (or $\{v_i,v_{i+1}\} \in E$)
  - We say that $u_2$ is reachable from $u_1$
  - The length of the path is $k$
  - It is a cycle if $v_0 = v_k$
• An undirected graph is \textit{connected} if every pair of vertices are connected by a path
  
  – The \textit{connected components} are the equivalence classes of the vertices under the “reachability” relation. (All connected pair of vertices are in the same connected component).

• A directed graph is \textit{strongly connected} if every pair of vertices are reachable from each other
  
  – The \textit{strongly connected components} are the equivalence classes of the vertices under the “mutual reachability” relation.

• Graphs appear all over the place in all kinds of applications, e.g:
  
  – Trees ($|E| = |V| - 1$)
  – Connectivity/dependencies (house building plans, WWW-page connections, …)

• Often the edges $(u, v)$ in a graph have weights $w(u, v)$, e.g.
  
  – Road networks (distances)
  – Cable networks (capacity)

1.2 \textbf{Representation}

• \textit{Adjacency-list} representation:
  
  – Array of $|V|$ list of edges incident to each vertex.

Examples:

\begin{itemize}
  \item Note: For undirected graphs, every edge is stored twice.
  \item If graph is weighted, a weight is stored with each edge.
\end{itemize}
• **Adjacency-matrix** representation:

  – $|V| \times |V|$ matrix $A$ where

  $$a_{ij} = \begin{cases} 
  1 & \text{if } (i,j) \in E \\
  0 & \text{otherwise}
  \end{cases}$$

  Examples:

  ![Graph and Adjacency Matrices](image)

  – Note: For undirected graphs, the adjacency matrix is symmetric along the main diagonal ($A^T = A$).
  – If graph is weighted, weights are stored instead of one's.

• **Comparison of matrix and list representation:**

<table>
<thead>
<tr>
<th>Adjacency list</th>
<th>Adjacency matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Good if graph <em>sparse</em> ($</td>
<td>E</td>
</tr>
<tr>
<td>No quick access to $(u,v)$</td>
<td>$O(1)$ access to $(u,v)$</td>
</tr>
</tbody>
</table>

• We will use adjacency list representation unless stated otherwise ($O(|V| + |E|)$ space).

2 **Graph traversal**

• There are two standard (and simple) ways of traversing all vertices/edges in a graph in a systematic way

  – Breadth-first
  – Depth-first

• We can use them in many fundamental algorithms, e.g finding cycles, connected components, ...

3
2.1 Breadth-first search (BFS)

- Main idea:
  - Start at some source vertex \( s \) and visit,
  - All vertices at distance 1,
  - Followed by all vertices at distance 2,
  - Followed by all vertices at distance 3,
    
- BFS corresponds to computing shortest path distance (number of edges) from \( s \) to all other vertices.
- To control progress of our BFS algorithm, we think about coloring each vertex
  - White before we start,
  - Gray after we visit the vertex but before we have visited all its adjacent vertices,
  - Black after we have visited the vertex and all its adjacent vertices (all adjacent vertices are gray).
- We use a queue \( Q \) to hold all gray vertices—vertices we have seen but are still not done with.
- We remember from which vertex a given vertex \( v \) is colored gray (\( \text{visit}[v] \)).
- Algorithm:

\[
\text{BFS}(s) \\
\quad \text{color}[s] = \text{gray} \\
\quad d[s] = 0 \\
\text{ENQUEUE}(Q, s) \\
\text{WHILE } Q \text{ not empty DO} \\
\quad \text{DEQUEUE}(Q, u) \\
\quad \text{FOR } (u, v) \in E \text{ DO} \\
\quad \quad \text{IF color}[v] = \text{white THEN} \\
\quad \quad \quad \text{color}[v] = \text{gray} \\
\quad \quad \quad d[v] = d[u] + 1 \\
\quad \quad \quad \text{visit}[v] = u \\
\quad \quad \quad \text{ENQUEUE}(Q, v) \\
\quad \quad \text{FI} \\
\quad \text{color}[u] = \text{black} \\
\text{OD}
\]

- Algorithm runs in \( O(|V| + |E|) \) time
**Example (for directed graph):**

- ![Diagram A](image1)
- ![Diagram B](image2)
- ![Diagram C](image3)
- ![Diagram D](image4)
- ![Diagram E](image5)
- ![Diagram F](image6)
- ![Diagram G](image7)
- ![Diagram H](image8)
- ![Diagram I](image9)

**Note:**
- visit[$v$] forms a tree; **BFS-tree**.
- $d[v]$ contains length of shortest path from $s$ to $v$.
- We can use visit[$v$] to find the shortest path from $s$ to a given vertex.

- If graph is not connected we have to try to start the traversal at all nodes.

```plaintext
FOR each vertex $u \in V$ DO
    IF color[$u$] = white THEN BFS($u$)
OD
```

- Note: We can use algorithm to compute connected components in $O(|V| + |E|)$ time.
2.2 Depth-first search (DFS)

- If we use stack instead of queue $Q$ we get another traversal order; depth-first
  - We go “as deep as possible”,
  - Go back until we find unexplored adjacent vertex,
  - Go as deep as possible,
  
- Often we are interested in “start time” and “finish time” of vertex $u$
  - $Start\ time\ (d[u])$: indicates at what “time” vertex is first visited.
  - $Finish\ time\ (f[u])$: indicates at what “time” all adjacent vertices have been visited.

- Instead of using a stack in a DFS algorithms, we can write a recursive procedure
  - We will color a vertex gray when we first meet it and black when we finish processing all adjacent vertices.

- Algorithm:

  \[
  \text{DFS}(u) \\
  \text{color}[u] = \text{gray} \\
  d[u] = \text{time} \\
  \text{time} = \text{time} + 1 \\
  \text{FOR } (u, v) \in E \text{ DO} \\
  \quad \text{IF color}[v] = \text{white} \text{ THEN} \\
  \quad \quad \text{visit}[v] = u \\
  \quad \quad \text{DFS}(v) \\
  \quad \text{FI} \\
  \text{OD} \\
  \text{color}[u] = \text{black} \\
  f[u] = \text{time} \\
  \text{time} = \text{time} + 1
  \]

- Algorithm runs in $O(|V| + |E|)$ time
  - As before we can extend algorithm to unconnected graphs and we can use it to detect cycles in $O(|V| + |E|)$ time.
• Example:

a)

b)

c)

d)

e)

f)

g)

h)

i)

j)

k)

l)
• As previously visit[\(v\)] forms a tree; DFS-tree
  
  – Note: If \(u\) is descendent of \(v\) in DFS-tree then \(d[v] < d[u] < f[u] < f[v]\)

3 Topological sorting

• Definition: Topological sorting of directed acyclic graph \(G = (V, E)\) is a linear ordering of
  vertices \(V\) such that \((u, v) \in E \Rightarrow u\) appear before \(v\) in ordering.

• Topological ordering can be used in scheduling:
  
  – Example: Dressing (arrow implies “must come before”)

We want to compute order in which to get dressed. One possibility:

The given order is one possible topological order.

• Algorithm: Topological order just reverse DFS finish time (\(\Rightarrow O(|V| + |E|)\) running time).
• Correctness: \((u, v) \in E \iff f(v) < f(u)\)
  
  – Proof: When \((u, v)\) is explored by DFS algorithm, \(v\) must be white or black (gray \(\Rightarrow\) cycle).
    * \(v\) white: \(v\) visited and finished before \(u\) is finished \(\Rightarrow f(v) < f(u)\)
    * \(v\) black: \(v\) already finished \(\Rightarrow f(v) < f(u)\)

• Alternative algorithm: Count in-degree of each vertex and repeatedly number and remove in-degree 0 vertex and its outgoing edges:

```
FOR all vertices \(v\) DO
  degree\([v]\) = 0
OD
FOR all edges \((u, v) \in E\) DO
  degree\([v]\) = degree\([v]\) + 1
  IF degree\([v]\) = 0 THEN ENQUEUE\((Q, v)\)
OD
i = 0
WHILE \(Q \neq \emptyset\) DO
  DEQUEUE\((Q, u)\)
  Topsort\((u) = i\)
  i = i + 1
  FOR all edges \((u, v) \in E\) DO
    degree\([v]\) = degree\([v]\) - 1
    IF degree\([v]\) = 0 THEN ENQUEUE\((Q, v)\)
  OD
OD
```