Graphs

Graphs are generalizations of trees

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

Determining if there is a way to get from one web page to another, following hyperlinks

Finding the fastest route from one city to another

Determining an ordering of courses so that prerequisites are satisfied

Analysis of communication networks (e.g., cost of building a network with various connectivity requirements)

Analysis of social networks ("six degrees of separation")
\[ G = (V, E) : \]
- \( V \) = set of nodes
- \( E \) = set of edges

In this class, \( G \) is usually assumed to be directed, so that an edge is a pair of nodes \((v, w)\) (graphically, \( v \rightarrow w \))

If \((v, w) \in E\), let’s call \( w \) a **successor** of \( v \), and \( v \) a **predecessor** of \( w \)

**Successor**\((v) := \) set of all successors of \( v \)

An **undirected** graph can be thought of as a directed graph, where \((v, w) \in E \Rightarrow (w, v) \in E\)
Graph representations

- **Sparse**: an array of *adjacency lists*
  - an array $A$ indexed by $V$, where $A[v]$ is a linked list containing all successors of $v$
  - size: $O(|V| + |E|)$
  - this will be the “default”

- **Dense**: an *adjacency matrix*
  - a 2D boolean array $A$ indexed by $V \times V$, where $A[v, w] = true$ iff $(v, w) \in E$
  - size: $O(|V|^2)$
  - May be preferable when $|E| \approx |V|^2$
Example:

\[ V = \{1, 2, 3, 4, 5, 6\} \]
\[ E = \{(1, 2), (1, 4), (2, 3), (2, 4), (2, 5), (3, 4), (5, 3), (6, 3), (6, 5)\} \]

Adjacency Lists

Adjacency Matrix

\[
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{pmatrix}
\]
Weighted graphs

Edges can have weights, representing costs or distances.

**Weighted Digraph**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>0.2</td>
<td>0.4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>0.6</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0.5</td>
<td>∞</td>
<td>∞</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>0.1</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Adjacency matrix**

- Adjacency Lists
- Adjacency matrix
Directed graphs: cycles

A **cycle** in a graph is a path (with at least one edge) from a node back to itself:

$$\nu_0 \rightarrow \nu_1 \rightarrow \nu_2 \cdots \rightarrow \nu_{k-1} \rightarrow \nu_0$$

Example:

A **DAG (Directed Acyclic Graph)** is a directed graph with no cycles
Topological Sort

Suppose \( G = (V, E) \) is a DAG

A topological sort of \( G \) is an ordering of the vertices \((v_1, v_2, \ldots, v_n)\) such that \((v_i, v_j) \in E \Rightarrow i < j\)

“all arrows go from left to right”

Example:
Topological Sort

Applications:

- “Generic application”:
  - Nodes represent “tasks”
  - Edges represent “scheduling constraints”
    - \( v \rightarrow w \) means “task \( v \) must be performed before task \( w \)”
  - A topological sort is a schedule for performing all tasks that respects all of the constraints

- Concrete applications:
  - Software build systems (e.g., Unix `make` command)
  - Software packaging systems (e.g., `apt-get` from Linux)
  - Spreadsheets (ordering of formula cell evaluation)
Why only DAGs?

There was only one catch and that was Catch-22, which specified that a concern for one's safety in the face of dangers that were real and immediate was the process of a rational mind. Orr was crazy and could be grounded. All he had to do was ask; and as soon as he did, he would no longer be crazy and would have to fly more missions. Orr would be crazy to fly more missions and sane if he didn’t, but if he was sane he had to fly them. If he flew them he was crazy and didn’t have to; but if he didn’t want to he was sane and had to. Yossarian was moved very deeply by the absolute simplicity of this clause of Catch-22 and let out a respectful whistle.

“That’s some catch, that Catch-22,” he observed.

“It’s the best there is,” Doc Daneeka agreed.

– Joseph Heller, Catch-22
Topological Sort

**Theorem:** A directed graph $G$ is a DAG $\iff$ it has a topological sort

**Proof:** ($\iff$) — *easy direction*

Want to show: if $G$ has a top. sort, then $G$ is a DAG

*Proof by contradiction:* assume $G$ has a top. sort, but $G$ is not a DAG . . . and derive a contradiction

$G$ has a cycle, say: $\nu_0 \rightarrow \nu_1 \rightarrow \nu_2 \cdots \rightarrow \nu_{k-1} \rightarrow \nu_0$

Fix a particular top. sort

There is an edge from $\nu_0$ to $\nu_1$, so $\nu_0$ must come before $\nu_1$ in the top. sort

There is also a path from $\nu_1$ to $\nu_0$, so $\nu_1$ must come before $\nu_0$ in the top. sort

$\Rightarrow \Leftarrow$
Topological Sort

**Theorem:** A directed graph $G$ is a DAG $\iff$ it has a topological sort

**Proof:** ($\implies$) — *harder direction*

Want to show: if $G$ is a DAG, then $G$ has a top. sort

Proof by induction on $n = \# \text{ nodes}$

Base case: $n = 1 \checkmark$

Let $n > 1$, and assume every DAG with $n - 1$ nodes has a top. sort (induction hypothesis)

We show that every DAG with $n$ nodes has a top. sort
Topological Sort

(Proof cont’d)

Let $G$ be a DAG with $n$ nodes

Want to show: $G$ has a top. sort

There must be a node $v_1$ that has in-degree 0, i.e., with no incoming edges (why?)

Delete $v_1$ (along with all of its outgoing edges) from $G$, giving us a DAG $G'$ with $n - 1$ nodes

By induction hypothesis, $G'$ has a top. sort $(v_2, \ldots, v_n)$

$(v_1, v_2, \ldots, v_n)$ is a top. sort for $G$ (why?)

QED
Topological Sort

The proof suggests an algorithm:

\[
L \leftarrow () \hspace{1cm} // \text{Initialize empty list}
\]

while \( G \) contains a node \( \nu \) of in-degree 0 do
    delete \( \nu \) (and its outgoing edges) from \( G \)
    append \( \nu \) to \( L \)

if \( G \) is empty
    then output \( L \)
else output “graph contains a cycle”
Example:

There's only one vertex with in-degree 0 (vertex 1), so we start with that one (and think of it as removed from the graph):
Topological Sort

Example:

As we “remove” 1, now 2 and 3 have in-degree 0, so the topological sort could continue with either one of these — we’ll choose 2.
Topological Sort

Example:

1, 2
Topological Sort

Example:

1, 2, 4
Topological Sort

Example:

1, 2, 4, 8
Topological Sort

Example:

1, 2, 4, 8, 5
Topological Sort

Example: ...

1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13
Topological Sort

Kahn’s Algorithm: a linear time — $O(|V| + |E|)$ — implementation

We assume:

• $V = \{0, \ldots, n - 1\}$
• Adjacency list representation

We need:

• An array $deg[0..n)$ (initially all zero)
  // Keeps track of in-degrees
• A set $S$ of nodes (initially empty)
  // tracks all nodes of in-degree 0
• A list $L$ (initially empty)
  // The topological sort
Topological Sort

// Initialize in-degrees
for \( v \) in \([0..n]\) do
    for each \( w \in \text{Successor}(v) \) do
        increment \( \text{deg}[w] \)

// Initialize \( S \) to all nodes of in-degree 0
for \( v \) in \([0..n]\) do
    if \( \text{deg}[v] = 0 \) then add \( v \) to \( S \)

// Main loop
while \( S \) is not empty do
    remove some \( v \) from \( S \)
    append \( v \) to \( L \)
    for each \( w \in \text{Successor}(v) \) do
        decrement \( \text{deg}[w] \)
        if \( \text{deg}[w] = 0 \) then add \( w \) to \( S \)

output \( L \)  // Graph is a DAG \( \iff \) \( L \) has length \( n \)
Running time analysis:

- “Initialize in-degrees” and “Main loop”: each vertex \( v \) and each edge \( v \rightarrow w \) is visited once.
- \( \therefore \) total running time is \( O(|V| + |E|) \).
Example: gathering coins

Given a DAG $G = (V, E)$

On each node $v$ there are $N[v]$ coins

Goal: find the max number of coins that can be gathered on any one path through $G$
Solution:
Assume $V = [0..n)$ and let $TopSort[0..n)$ be an array that lists vertices in a topological order.
We will compute $P[v] = \text{maximum number of coins you can gather on any path starting at } v$.

Idea:
Evaluate $P$ values right to left:

$$P[u] = N[u] + \max \{ P[v], P[v'] \}$$

Algorithm:
for $i$ in reverse $[0..n)$
    $u \leftarrow TopSort[i]$
    $max \leftarrow 0$
    for each $v \in Successor(u)$ do
        if $P[v] > max$ then $max \leftarrow P[v]$
    $P[u] \leftarrow N[u] + max$