21.1 Graphs

A graph has two parts: **vertices** (one vertex) also called **nodes**. An **undirected graph** has undirected **edges**. Two vertices joined by edge are **neighbors**. A **directed graph** has directed **edges/arcs**; each arc goes from **in-neighbor** to **out-neighbor**. Examples include:

- city map
- circuit diagram
- chemical molecule
- family tree

A **path** is sequence of vertices with successive vertices joined by edge/arc. A **cycle** is a sequence of vertices ending up where started such that successive vertices are joined by edge/arc. A graph is **connected** (a directed graph is **strongly connected**) if there is a path from every vertex to every other vertex.

![Connected and Not Strongly Connected Graphs]

21.2 Graph Representation

There are two standard approaches to storing a graph:

**Adjacency Matrix**

1) container of numbered vertices, and
2) array where each entry has info about the corresponding edge.

**Adjacency List**

1) container of vertices, and
2) for each vertex an unsorted bag of out-neighbors.
An example directed graph (with labeled vertices and arcs):

Adjacency array:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>—</td>
<td>orange</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>—</td>
<td>—</td>
<td>black</td>
<td>green</td>
<td>blue</td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
<td>—</td>
<td>yellow</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E</td>
<td>white</td>
<td>red</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Adjacency list:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>orange, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>black, C</td>
<td>green, D</td>
<td>blue, E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>D</td>
<td>—</td>
<td>—</td>
<td>yellow</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>E</td>
<td>red, B</td>
<td>white, A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The advantage of the adjacency matrix is that determining $\text{isAdjacent}(u,v)$ is $O(1)$. The disadvantage of adjacency matrix is that it can be space-inefficient, and enumerating $\text{outNeighbors}$ etc. can be slow.

### 21.3 Aside

**Practice.** Draw each of the following without lifting your pen or going over the same line twice.

### 21.4 Topological Sort

A *DAG*, directed acyclic graph, is a directed graph without directed cycles. The classic application is scheduling constraints between tasks of a project.
A topological ordering is an ordering of the vertices such that every arc goes from lower number to higher number vertex.

**Example.** In the following DAG, one topological ordering is: E A F B D C.

A source is a vertex with no in-arcs and a sink is one with no out-arcs.

**Theorem:**

a) If a directed graph has a cycle, then there is no topological ordering.
b) A DAG has at least one source and one sink.
c) A DAG has a topological ordering.

Consider the proof of (a). If there is a cycle, then we have an insoluble constraint: if, say the cycle is $A \rightarrow B \rightarrow C \rightarrow A$, then that means $A$ must occur before $B$, $B$ before $C$, and $C$ before $A$, which cannot be done.

Consider the proof of (b). We prove the contrapositive. Consider a directed graph without a sink. Then consider walking around the graph. Every time we visit a vertex we can still leave, because it is not a sink. Because the graph is finite, we must eventually revisit a vertex we’ve been to before. This means that the graph has a cycle. The proof for the existence of a source is similar.

The proof of (c) is given by the algorithm below.

### 21.5 Algorithm for Topological Ordering

Here is an algorithm for finding a topological ordering:

**Algorithm:** TopologicalOrdering()

Repeatedly

Find source, output and remove

For efficiency, use the Adjacency List representation of the graph. Also:

1. maintain a counter in-degree at each vertex $v$; this counts the arcs into the vertex from “nondeleted” vertices, and decrement every time the current source has an arc to $v$ (no actual deletions).

2. every time a decrement creates a source, add it to a container of sources.

There is even an efficient way to initially calculate the in-degrees at all vertices simultaneously. (How?)
Sample Code

Here is an abstract base class \texttt{DAG}, an implementation of topological sort for that class, and an adjacency-list implementation of the class

\begin{verbatim}
Dag.h
GraphAlgorithms.cpp
AListDAG.h
AListDAG.cpp
\end{verbatim}