CHAPTER 13
GRAPH ALGORITHMS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO AND JORY DENNY
DIRECTED GRAPHS
A digraph is a graph whose edges are all directed
- Short for “directed graph”

Applications
- one-way streets
- flights
- task scheduling
A graph \( G = (V, E) \) such that
- Each edge goes in one direction:
- Edge \((a, b)\) goes from \(a\) to \(b\), but not \(b\) to \(a\)
- If \( G \) is simple, \( m < n(n - 1) \)
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size
Scheduling: edge \((a, b)\) means task \(a\) must be completed before \(b\) can be started.
DIRECTED DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.
- In the directed DFS algorithm, we have four types of edges:
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex $s$ determines the vertices reachable from $s$. 
REACHABILITY

- DFS tree rooted at $v$: vertices reachable from $v$ via directed paths
STRONG CONNECTIVITY

- Each vertex can reach all other vertices
STRONG CONNECTIVITY ALGORITHM

- Pick a vertex \( v \) in \( G \)
- Perform a DFS from \( v \) in \( G \)
  - If there’s a \( w \) not visited, print “no”
- Let \( G' \) be \( G \) with edges reversed
- Perform a DFS from \( v \) in \( G' \)
  - If there’s a \( w \) not visited, print “no”
  - Else, print “yes”
- Running time: \( O(n + m) \)
STRONGLY CONNECTED COMPONENTS

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n + m)$ time using DFS, but is more complicated (similar to biconnectivity).
Given a digraph $G$, the transitive closure of $G$ is the digraph $G^*$ such that:

- $G^*$ has the same vertices as $G$
- if $G$ has a directed path from $u$ to $v$ ($u \rightarrow v$), $G^*$ has a directed edge from $u$ to $v$

The transitive closure provides reachability information about a digraph.
We can perform DFS starting at each vertex
- $O(n(n + m))$

If there's a way to get from $A$ to $B$ and from $B$ to $C$, then there's a way to get from $A$ to $C$.

Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm
FLOYD-WARSHALL TRANSITIVE CLOSURE

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, ..., k, as intermediate vertices:

Uses only vertices numbered \(i, \ldots, k\)
(\text{add this edge if it’s not already in})

Uses only vertices numbered \(i, \ldots, k-1\)

Uses only vertices numbered \(k, \ldots, j\)
Number vertices $v_1, \ldots, v_n$

Compute digraphs $G_0, \ldots, G_n$

- $G_0 \leftarrow G$
- $G_k$ has directed edge $(v_i, v_j)$ if $G$ has a directed path from $v_i$ to $v_j$

We have that $G_n = G^*$

In phase $k$, digraph $G_k$ is computed from $G_{k-1}$

Running time: $O(n^3)$, assuming $G$.areAdjacent$(v_i, v_j)$ is $O(1)$ (e.g., adjacency matrix)

**Algorithm** FloydWarshall($G$)

**Input:** Digraph $G$

**Output:** Transitive Closure $G^*$ of $G$

1. Name each vertex $v \in G$.vertices() with $i = 1 \ldots n$
2. $G_0 \leftarrow G$
3. for $k \leftarrow 1 \ldots n$ do
4.   $G_k \leftarrow G_{k-1}$
5.   for $i \leftarrow 1 \ldots n \mid i \neq k$ do
6.     for $j \leftarrow 1 \ldots n \mid j \neq i, k$ do
7.       if $G_{k-1}$.areAdjacent$(v_i, v_k) \land G_{k-1}$.areAdjacent$(v_k, v_j) \land \neg G_k$.areAdjacent$(v_i, v_j)$ then
8.         $G_k$.insertDirectedEdge$(v_i, v_j)$
9.     return $G_n$
FLOYD-WARSHALL, ITERATION 1
FLOYD-WARSHALL, ITERATION 3
FLOYD-WARSHALL, ITERATION 4
FLOYD-WARSHALL, ITERATION 5
FLOYD-WARSHALL, ITERATION 6
FLOYD-WARSHALL, CONCLUSION
A directed acyclic graph (DAG) is a digraph that has no directed cycles.

A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices such that for every edge $(v_i, v_j)$, we have $i < j$.

Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints.

Theorem - A digraph admits a topological ordering if and only if it is a DAG.
Number vertices, so that \((u, v)\) in \(E\) implies \(u < v\)
Number vertices, so that $(u, v)$ in $E$ implies $u < v$
**ALGORITHM FOR TOPOLOGICAL SORTING**

- Note: This algorithm is different than the one in the book

```
Algorithm TopologicalSort(G)
1. \( H \leftarrow G \)
2. \( n \leftarrow G.\text{numVertices()} \)
3. while \( \neg H.\text{empty()} \) do
4. Let \( v \) be a vertex with no outgoing edges
5. Label \( v \leftarrow n \)
6. \( n \leftarrow n - 1 \)
7. \( H.\text{eraseVertex}(v) \)
```
IMPLEMENTATION WITH DFS

- Simulate the algorithm by using depth-first search
- $O(n + m)$ time.

**Algorithm** topologicalDFS($G$)
**Input:** DAG $G$
**Output:** Topological ordering of $G$

1. $n \leftarrow G$.numVertices()
2. Initialize all vertices as $UNEXPLORERED$
3. for each vertex $v \in G$.vertices() do
4.   if $v$.getLabel() = $UNEXPLORERED$ then
5.     topologicalDFS($G, v$)
6.   else
7.     // $e$ is a forward, cross, or back edge
8.     Label $v$ with topological number $n$
9.     $n \leftarrow n - 1$

**Algorithm** topologicalDFS($G, v$)
**Input:** DAG $G$, start vertex $v$
**Output:** Labeling of the vertices of $G$

in the connected component of $v$

1. $v$.setLabel(VISITED)
2. for each $e \in v$.outEdges() do
3.   $w \leftarrow e$.dest()
4.   if $w$.getLabel() = $UNEXPLORERED$ then
5.     // $e$ is a discovery edge
6.     topologicalDFS($G, w$)
7.   else
8.     // $e$ is a forward, cross, or back edge
9.     Label $v$ with topological number $n$
10. $n \leftarrow n - 1$
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE

Diagram of a directed graph with nodes labeled 1 to 9, showing the topological order.