Data Structures
Lecture 15

Fang Yu
Department of Management Information Systems
National Chengchi University
Graphs II

Digraphs, Strongly Connective Component, Topological Sorting, and Minimum Spanning Tree
Digraphs

- A digraph is a graph whose edges are all directed
  - Short for “directed graph”

- Applications
  - one-way streets
  - flights
  - task scheduling
Digraph Properties

- 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, \( m \leq n \cdot (n - 1) \)

- 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).

\[\{ a, c, g \}\]
\[\{ f, d, e, b \}\]
Transitive Closure

- 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 $v$ ($u \neq v$), $G^*$ has a directed edge from $u$ to $v$

- The transitive closure provides reachability information about a digraph
Computing the Transitive Closure

- 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 1, ..., k-1
  - Uses only vertices numbered 1, ..., k-1
  - Uses only vertices numbered 1, ..., k-1
  - (add this edge if it’s not already in)
Floyd-Warshall’s Algorithm

- Number vertices \( v_1, \ldots, v_n \)

- Compute digraphs \( G_0, \ldots, G_n \)
  - \( G_0 = G \)
  - \( G_k \) has directed edge \((v_i, v_j)\) if \( G \) has a directed path from \( v_i \) to \( v_j \) with intermediate vertices in \( \{v_1, \ldots, v_k\} \)

- We have that \( G_n = G^* \)

- In phase \( k \), digraph \( G_k \) is computed from \( G_{k-1} \)

- Running time: \( O(n^3) \), assuming \( \text{areAdjacent} \) is \( O(1) \) (e.g., adjacency matrix)

---

**Algorithm** \( \text{FloydWarshall}(G) \)

**Input** digraph \( G \)

**Output** transitive closure \( G^* \) of \( G \)

\[
i \leftarrow 1
\]

for all \( v \in G.\text{vertices()} \)

- denote \( v \) as \( v_i \)

\[
i \leftarrow i + 1
\]

\[
G_0 \leftarrow G
\]

**for** \( k \leftarrow 1 \) to \( n \) do

\[
G_k \leftarrow G_{k-1}
\]

**for** \( i \leftarrow 1 \) to \( n \) (\( i \neq k \)) do

**for** \( j \leftarrow 1 \) to \( n \) (\( j \neq i, k \)) do

  - if \( G_{k-1}.\text{areAdjacent}(v_i, v_k) \land G_{k-1}.\text{areAdjacent}(v_k, v_j) \)

  - if \( \neg G_k.\text{areAdjacent}(v_i, v_j) \)

  - \( G_k.\text{insertDirectedEdge}(v_i, v_j, k) \)

**return** \( G_n \)
Floyd-Warshall Example
Floyd-Warshall, Iteration 1
Floyd-Warshall, Iteration 2
Floyd-Warshall, Iteration 3
Floyd-Warshall, Iteration 4
Floyd-Warshall, Iteration 5
Floyd-Warshall, Iteration 6
Floyd-Warshall, Conclusion
DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles.
DAGs and Topological Ordering

- A topological ordering of a digraph is a numbering $v_1, ..., v_n$ 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.
Topological Sorting

- Number vertices, so that \((u,v)\) in \(E\) implies \(u < v\)

A typical student day:
- Wake up
- Study computer sci.
- Eat
- Nap
- More c.s.
- Play
- Write c.s. program
- Bake cookies
- Sleep
- Dream about graphs
- Work out
Algorithm for Topological Sorting

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

**Algorithm** TopologicalSort($G$)

- \( H \leftarrow G \)  // Temporary copy of $G$
- \( n \leftarrow G.numVertices() \)
- **while** $H$ is not empty **do**
  - Let $v$ be a vertex with no outgoing edges
  - Label $v \leftarrow n$
  - \( n \leftarrow n - 1 \)
  - Remove $v$ from $H$

- Running time: $O(n + m)$
Implementation with DFS

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

**Algorithm** \( \text{topologicalDFS}(G, v) \)

**Input** graph \( G \) and a start vertex \( v \) of \( G \)

**Output** labeling of the vertices of \( G \) in the connected component of \( v \)

\( \text{setLabel}(v, \text{VISITED}) \)

for all \( e \in G.\text{outEdges}(v) \)

\( w \leftarrow \text{opposite}(v,e) \)

if \( \text{getLabel}(w) = \text{UNEXPLORED} \)

\( \text{topologicalDFS}(G, w) \)

else

\( \text{topologicalDFS}(G, v) \)

Label \( v \) with topological number \( n \)

\( n \leftarrow n - 1 \)
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example

Graph:

- Nodes: 3, 4, 5, 6, 7, 8, 9
- Edges:
  - 3 → 4
  - 4 → 5
  - 4 → 6
  - 4 → 7
  - 5 → 7
  - 6 → 8
  - 7 → 8
  - 7 → 9
  - 8 → 9

Sorting Example:

1. Start with node 3 (no incoming edges)
2. Add node 3 to the sorted list
3. Remove node 3 from the graph
4. Next, consider nodes 4, 6, and 7 (only one incoming edge each)
5. Choose any of these nodes (e.g., 4)
6. Add node 4 to the sorted list
7. Remove node 4 from the graph
8. Choose the next node based on incoming edges
9. Add node 5 to the sorted list (no incoming edges)
10. Add node 6 to the sorted list (one incoming edge)
11. Add node 7 to the sorted list (two incoming edges)
12. Add node 8 to the sorted list (one incoming edge)
13. Add node 9 to the sorted list (one incoming edge)

Sorted order: 3, 4, 5, 6, 7, 8, 9
Topological Sorting Example
Topological Sorting Example

This is a graph showing a topological sort example. Nodes are numbered 1 to 9, and the arrows indicate the order in which they should be processed.
A Quiz

- Fang loves CS courses and wants to plan his course schedule. The course prerequisites are:
  - CS15: (none)
  - CS16: CS15
  - CS22: (none)
  - CS31: CS15
  - CS32: CS16, CS31
  - CS126: CS22, CS32, CS16
  - CS127: CS16
  - CS141: CS22, CS16
  - CS169: CS32

Please help Fang to find the sequence of courses that allows him to satisfy all the prerequisites.
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph $G$ containing all the vertices of $G$

Spanning tree
- Spanning subgraph that is itself a tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

Applications
- Communications networks
- Transportation networks
Cycle Property

Cycle Property:

- Let $T$ be a minimum spanning tree of a weighted graph $G$.
- Let $e$ be an edge of $G$ that is not in $T$ and $C$ let be the cycle formed by $e$ with $T$.
- For every edge $f$ of $C$, $\text{weight}(f) \leq \text{weight}(e)$.

Proof:

- By contradiction.
- If $\text{weight}(f) > \text{weight}(e)$ we can get a spanning tree of smaller weight by replacing $e$ with $f$.
Partition Property

- Partition Property:
  - Consider a partition of the vertices of $G$ into subsets $U$ and $V$
  - Let $e$ be an edge of minimum weight across the partition
  - There is a minimum spanning tree of $G$ containing edge $e$

Replacing $f$ with $e$ yields another MST
Kruskal’ s Algorithm

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge “closest” clusters and their MSTs

- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge

- At the end of the algorithm
  - One cluster and one MST

Minimum Spanning Trees

**Algorithm KruskalMST(G)**

```plaintext
for each vertex v in G do
    Create a cluster consisting of v
let Q be a priority queue.
Insert all edges into Q
T ← Ø
{T is the union of the MSTs of the clusters}
while T has fewer than n - 1 edges do
    e ← Q.removeMin().getValue()
    [u, v] ← G.endVertices(e)
    A ← getCluster(u)
    B ← getCluster(v)
    if A ≠ B then
        Add edge e to T
        mergeClusters(A, B)
return T
```
Example
Example (contd.)

four steps

two steps

four steps
Prim-Jarnik’s Algorithm

- We pick an arbitrary vertex $s$ and we grow the MST as a cloud of vertices, starting from $s$.

- We store with each vertex $v$ label $d(v)$ representing the smallest weight of an edge connecting $v$ to a vertex in the cloud.

- At each step:
  - We add to the cloud the vertex $u$ outside the cloud with the smallest distance label.
  - We update the labels of the vertices adjacent to $u$. 
Prim-Jarnik’s Algorithm (cont.)

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
  - Key: distance
  - Value: vertex
  - Recall that method `replaceKey(l,k)` changes the key of entry `l`

- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Entry in priority queue

```
Algorithm `PrimJarnikMST(G)`
1. `Q ← new heap-based priority queue`
2. `s ← a vertex of G`
3. for all `v ∈ G.vertices()`
   1. if `v = s`
      1. `setDistance(v, 0)`
   2. else
      1. `setDistance(v, ∞)`
      2. `setParent(v, ∅)`
   4. `l ← Q.insert(getDistance(v), v)`
   5. `setLocator(v, l)`
4. while ¬ `Q.isEmpty()`
   1. `l ← Q.removeMin()`
   2. `u ← l.getValue()`
   3. for all `e ∈ G.incidentEdges(u)`
       1. `z ← G.opposite(u,e)`
       2. `r ← weight(e)`
       3. if `r < getDistance(z)`
           1. `setDistance(z, r)`
           2. `setParent(z,e)`
           3. `Q.replaceKey(getEntry(z), r)`
```
Example
Example (contd.)
Baruvka’s Algorithm

- Like Kruskal’s Algorithm, Baruvka’s algorithm grows many clusters at once and maintains a forest $T$

- Each iteration of the while loop halves the number of connected components in forest $T$

- The running time is $O(m \log n)$

**Algorithm** $BaruvkaMST(G)$

$T \leftarrow V$ \{just the vertices of $G$\}

while $T$ has fewer than $n - 1$ edges do

for each connected component $C$ in $T$ do

- Let edge $e$ be the smallest-weight edge from $C$ to another component in $T$

if $e$ is not already in $T$ then

- Add edge $e$ to $T$

return $T$
Example of Baruvka’s Algorithm (animated)
Schedule on Jan. 10

<table>
<thead>
<tr>
<th>Time</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00~10:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:00~11:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00~12:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Schedule on Jan. 10

<table>
<thead>
<tr>
<th>Time</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00-3:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:00-5:00</td>
<td>MAKEUP Exam</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Makeup exam on Jan. 17

- Friday 3:10-5:00. College of Commerce 313
- Maximal 80 points
- Dynamic programing on LCS
- Binary Search Tree (AVL)
- Hash Table
- Cycle Detection
- Minimum Spanning Tree