CMSC 132: Object-Oriented Programming II

DIRECTED GRAPHS

Graphs slides are modified from COS 126 slides of Dr. Robert Sedgewick.
Directed graphs

- Digraph
  - Set of vertices connected pairwise by directed edges.
Road network

Vertex = intersection; edge = one-way street.

Baltimore inner harbor
WordNet graph

Vertex = synset; edge = hypernym relationship.
## Digraph applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
Some digraph problems

- **Path:**
  - Is there a directed path from s to t?
- **Shortest path:**
  - What is the shortest directed path from s to t?
- **Topological sort:**
  - Can you draw a digraph so that all edges point upwards?
- **Strong connectivity:**
  - Is there a directed path between all pairs of vertices?
- **Transitive closure:**
  - For which vertices v and w is there a path from v to w?
- **PageRank:**
  - What is the importance of a web page?
Digraph Implementation

public class Digraph

    Digraph(int V)
    
    Digraph(In in)
    
    void addEdge(int v, int w)
    
    Iterable<Integer> adj(int v)
    
    int V()
    
    int E()
    
    Digraph reverse()
    
    String toString()
Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.
Adjacency-lists digraph implementation

```java
public class Graph {
    private final int V;
    private final Bag<Integer>[] adj;
    public Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }
    public void addEdge(int v, int w) {
        adj[v].add(w);
    }
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
**Digraph representation**

Comparisons of three different representations:

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from v to w</th>
<th>edge from v to w?</th>
<th>iterate over vertices pointing from v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$E$</td>
<td>1</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>1$^*$</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>1</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
</tr>
</tbody>
</table>

$^*$ disallows parallel edges
Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph (with edges in both directions).
  - DFS is a digraph algorithm.

DFS (to visit a vertex v)
Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.
Depth-first search demo

To visit a vertex $v$:

Mark vertex $v$ as visited.

Recursively visit all unmarked vertices pointing from $v$. 
Depth-first search demo

![Graph diagram]

- Vertices: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
- Edges: Various connections between vertices

**Table:**

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>

Reachable from vertex 0
Depth-first search Implementation

Code for directed graphs identical to undirected one.

```java
public class DirectedDFS {
    private boolean[] marked;
    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }
    public boolean visited(int v) {
        return marked[v];
    }
}
```
Reachability application: program control-flow analysis

- Every program is a digraph.
  - Vertex = basic block of instructions (straight-line program).
  - Edge = jump.
- Dead-code elimination.
  - Find (and remove) unreachable code.
Reachability application: mark-sweep garbage collector

- Every data structure is a digraph.
  - Vertex = object.
  - Edge = reference.

- Roots:
  - Objects known to be directly accessible by program (e.g., stack).

- Reachable objects:
  - Objects indirectly accessible by program (starting at a root and following a chain of pointers).
Breadth-first search in digraphs

Same method as for undirected graphs. Every undirected graph is a digraph (with edges in both directions). BFS is a digraph algorithm.

**BFS (from source vertex s)**
Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex pointing from v:
  add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$. 
Directed breadth-first search demo

Repeat until queue is empty:
  Remove vertex \( v \) from queue.
  Add to queue all unmarked vertices pointing from \( v \) and mark them.
Directed breadth-first search demo

Repeat until queue is empty:

Remove vertex \( v \) from queue.

Add to queue all unmarked vertices pointing from \( v \) and mark them.
Multiple-source shortest paths

- Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.
- Use BFS, but initialize by enqueuing all source vertices.

Example:
\[ S = \{1, 7, 10\} \].
Shortest path to 4 is \(7 \rightarrow 6 \rightarrow 4\). 1
Shortest path to 5 is \(7 \rightarrow 6 \rightarrow 0 \rightarrow 5\)
Shortest path to 12 is \(10 \rightarrow 12\).
Topological Sort
Precedence scheduling

Goal:

- Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

Digraph model:

- vertex = task;
- edge = precedence constraint.

0.CMSC216
1.CMSC330
2.CMSC351
3.CMSC131
4.CMSC420
5.CMSC250
6.CMSC132
Topological sort

- DAG:
  - Directed acyclic graph.
- Topological sort:
  - Redraw DAG so all edges point upwards.
Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder

**Diagram:**

```
0 -> 2 -> 3 -> 6 -> 1 -> 5
```

**Postorder:**
4, 1, 2, 5, 0, 6, 3

**Topological order:**
3, 6, 0, 5, 2, 1, 4
public class DepthFirstOrder {
    private boolean[] marked;
    private Stack<Integer> reversePost;
    public DepthFirstOrder(Digraph G) {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePost.push(v);
    }
    public Iterable<Integer> reversePost() {
        return reversePost;
    }
}

Depth-first search order
Topological sort

- Kahn's algorithm
  - First described by Kahn (1962),

1. find a vertex which has no incoming edges
2. insert it into a set $S$; at least one such vertex must exist in a non-empty acyclic graph.
2. Remove outgoing edges from that vertex, and repeat 1