CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

35–36: Directed Graphs

Alexandra Papoutsaki
Lectures

Mark Kampe
Labs
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Directed Graph Terminology

- **Directed Graph (or digraph)**: set of vertices $V$ connected pairwise by a set of directed edges $E$.
  - E.g., $V = \{0,1,2,3,4,5,6,7,8,9,10,11,12\}$,
    $E = \{\{0,1\}, \{0,5\}, \{2,0\}, \{2,3\}, \{3,2\}, \{3,5\}, \{4,2\}, \{4,3\}, \{5,4\}, \{6,0\}, \{6,4\}, \{6,9\}, \{7,6\}, \{7,8\}, \{8,7\}, \{8,9\}, \{9,10\}, \{9,11\}, \{10,12\}, \{11,4\}, \{11,12\}, \{12,9\}\}.$

- **Directed path**: a sequence of vertices in which there is a directed edge pointing from each vertex in the sequence to its successor in the sequence, with no repeated edges.
  - A simple directed path is a directed path with no repeated vertices.

- **Directed cycle**: Directed path with at least one edge whose first and last vertices are the same.
  - A simple directed cycle is a directed cycle with no repeated vertices (other than the first and last).

- The length of a cycle or a path is its number of edges.
Directed Graph Terminology

- **Self-loop**: an edge that connects a vertex to itself.
- Two edges are **parallel** if they connect the same pair of vertices.
- The **outdegree** of a vertex is the number of edges pointing from it.
- The **indegree** of a vertex is the number of edges pointing to it.
- A vertex \( w \) is **reachable** from a vertex \( v \) if there is a directed path from \( v \) to \( w \).
- Two vertices \( v \) and \( w \) are **strongly connected** if they are mutually reachable.
Directed Graph Terminology

- A digraph is **strongly connected** if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
- A **directed acyclic graph (DAG)** is a digraph is a graph with no directed cycles.
Anatomy of a digraph
## Digraph Applications

<table>
<thead>
<tr>
<th>Digraph</th>
<th>Vertex</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>Web page</td>
<td>Link</td>
</tr>
<tr>
<td>Cell phone</td>
<td>Person</td>
<td>Placed call</td>
</tr>
<tr>
<td>Financial</td>
<td>Bank</td>
<td>Transaction</td>
</tr>
<tr>
<td>Transportation</td>
<td>Intersection</td>
<td>One-way street</td>
</tr>
<tr>
<td>Game</td>
<td>Board</td>
<td>Legal move</td>
</tr>
<tr>
<td>Citation</td>
<td>Article</td>
<td>Citation</td>
</tr>
<tr>
<td>Infectious Diseases</td>
<td>Person</td>
<td>Infection</td>
</tr>
<tr>
<td>Food web</td>
<td>Species</td>
<td>Predator-prey relationship</td>
</tr>
</tbody>
</table>
## Popular digraph problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-&gt;t path</td>
<td>Is there a path from s to t?</td>
</tr>
<tr>
<td>Shortest s-&gt;t path</td>
<td>What is the shortest path from s to t?</td>
</tr>
<tr>
<td>Directed cycle</td>
<td>Is there a directed cycle in the digraph?</td>
</tr>
<tr>
<td>Topological sort</td>
<td>Can vertices be sorted so all edges point from earlier to later vertices?</td>
</tr>
<tr>
<td>Strong connectivity</td>
<td>Is there a directed path between every pair of vertices?</td>
</tr>
</tbody>
</table>
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components
Basic Graph API

- **public class** `Digraph`
- `Digraph(int V)`: create an empty digraph with V vertices.
- `void addEdge(int v, int w)`: add an edge v->w.
- `Iterable<Integer> adj(int v)`: return vertices adjacent from v.
- `int V()`: number of vertices.
- `int E()`: number of edges.
- `Digraph reverse()`: reverse edges of digraph.
DIRECTED GRAPHS

Digraph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent from $v$.
- Space efficient ($|E| + |V|$).
- Constant time for adding a directed edge.
- Lookup of a directed edge or iterating over vertices adjacent from $v$ is $\text{outdegree}(v)$. 
Adjacency-list digraph representation in Java

```java
public class Digraph {

    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    //Initializes an empty digraph with V vertices and 0 edges.
    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    //Adds the directed edge v->w to this digraph.
    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
    }

    //Returns the vertices adjacent from vertex v.
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components
Reachability

- Find all vertices reachable from $s$ along a directed path.
Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
  - Maximum number of edges in a simple digraph is \( n(n - 1) \).
- **DFS** (to visit a vertex \( v \))
  - Mark vertex \( v \).
  - Recursively visit all unmarked vertices \( w \) adjacent from \( v \).
- Typical applications:
  - Find a directed path from source vertex \( s \) to a given target vertex \( v \).
  - Topological sort.
  - Directed cycle detection.
4.2 Directed DFS Demo
Directed depth-first search in Java

```java
public class DirectedDFS {
    private boolean[] marked; // marked[v] = is there an s->v path?

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // directed depth first search from v
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
}
```
Alternative iterative implementation with a stack

```java
public class DirectedDFS {
    private boolean[] marked; // marked[v] = is there an s->v path?

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // iterative dfs that uses a stack
    private void dfs(Digraph G, int v) {
        Stack stack = new Stack();
        s.push(v);
        while (!stack.isEmpty()) {
            int vertex = stack.pop();
            if (!marked[vertex]) {
                marked[vertex] = true;
                while (int w : G.adj(vertex)) {
                    if (!marked[w]){
                        stack.push(w);
                    }
                }
            }
        }
    }
}
```
Depth-first search Analysis

- DFS marks all vertices reachable from $s$ in time proportional to $|V| + |E|$ in the worst case.
- Initializing arrays marked takes time proportional to $|V|$.
- Each adjacency-list entry is examined exactly once and there are $E$ such edges.
- Once we run DFS, we can check if vertex $v$ is reachable from $s$ in constant time. We can also find the $s \rightarrow v$ path (if it exists) in time proportional to its length.
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components
Breadth-first search

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

- BFS (from source vertex \( S \))
  - Put \( S \) on queue and mark \( S \) as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex \( v \).
    - Enqueue all unmarked vertices adjacent from \( v \), and mark them.

- Typical applications:
  - Find the shortest (in terms of number of edges) directed path between two vertices in time proportional to \(|E| + |V|\).
4.2 Directed BFS Demo
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components
Depth-first orders

- If we save the vertex given as argument to recursive dfs in a data structure, we have three possible orders of seeing the vertices:
  - **Preorder**: Put the vertex on a queue before the recursive calls.
  - **Postorder**: Put the vertex on a queue after the recursive calls.
  - **Reverse postorder**: Put the vertex on a stack after the recursive calls.
public class DepthFirstOrder {
    private boolean[] marked;       // marked[v] = has v been marked in dfs?
    private Queue<Integer> preorder;  // vertices in preorder
    private Queue<Integer> postorder; // vertices in postorder
    private Stack<Integer> reversePostOrder; // vertices in reverse postorder

    /**
     * Determines a depth-first order for the digraph G.
     * @param G the digraph
     */
    public DepthFirstOrder(Digraph G) {
        postorder = new Queue<Integer>();
        preorder = new Queue<Integer>();
        reversePostOrder = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    // run DFS in digraph G from vertex v and compute preorder/postorder
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        preorder.enqueue(v);
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        reversePostOrder.push(v);
    }
}
Depth-first orders

TOPOLOGICAL SORT

Depth-first orders

```
preorder is order of dfs() calls

pre          post          reversePost
0             0             3
5             4             1
0             4             5
5             5             4
0             5             1
4             4             1
done          done          done
dfs(5)        done          done
4             done          done
dfs(4)        done          done
5             done          done
dfs(1)        done          done
1             done          done
dfs(6)        done          done
9             done          done
dfs(9)        done          done
dfs(11)       done          done
12            done          done
dfs(12)       done          done
11            done          done
dfs(10)       done          done
done          done          done
check 11      done          done
dfs(11)       done          done
check 12      done          done
dfs(12)       done          done
12            done          done
done          done          done
check 4       done          done
dfs(0)        done          done
0             done          done
done          done          done
check 1       done          done
dfs(2)        done          done
check 0       done          done
dfs(3)        done          done
check 5       done          done
dfs(7)        done          done
check 6       done          done
dfs(6)        done          done
7             done          done
dfs(7)        done          done
check 7       done          done
dfs(8)        done          done
check 8       done          done
done          done          done
check 10      done          done
done          done          done
check 11      done          done
check 12      done          done
```

queue
4
4
5
queue
4
5
1
queue
1
5
4
stack
Topological sort

- **Goal**: Order the vertices of a DAG so that all edges point from an earlier vertex to a later vertex.
  - Think of modeling major requirements as a DAG.
  - Reverse postorder in DAG is a topological sort.
  - With DFS, we can topologically sort a DAG in $|E| + |V|$ time.
4.2 Topological Sort Demo
Summary

- Single-source reachability in a digraph: DFS/BFS.
- Shortest path in a digraph: BFS.
- Topological sort in a DAG: DFS.
Lecture 35-36: Directed Graphs

- Introduction to Directed Graphs
- Digraph API
- Depth-First Search
- Breadth-First Search
- Topological Sort
- Strongly Connected Components
Is a digraph strongly connected?

- Pick a random starting vertex $S$.
- Run DFS/BFS starting at $S$.
  - If have not reached all vertices, return false.
- Reverse edges.
- Run DFS/BFS again on reversed graph.
  - If have not reached all vertices, return false.
  - Else return true.
ASSIGNED READINGS AND PRACTICE PROBLEMS

Readings:

- Textbook: Chapter 4.2 (Pages 566-594)
- Website:
  - [https://algs4.cs.princeton.edu/42digraph/](https://algs4.cs.princeton.edu/42digraph/)

Practice Problems:

- 4.2.1-4.27