Lab Session 6

Elementary Graph Algorithms

The Art of Computer Programming 2 — Week 7

Objectives

- Represent graphs as adjacency lists and compute the transpose
- Implement BFS and use it for shortest paths and connected components
- Implement DFS with timestamps and edge classification
- Detect cycles and compute topological orderings (DFS + Kahn)
- Find strongly connected components with Kosaraju's algorithm

All programming is done in a single Python file. We represent graphs as **adjacency lists** using a dictionary of lists:

```
# Directed graph: {vertex: [list of neighbors]}
graph = {
    'u': ['v', 'x'],
    'v': ['y'],
    'x': ['v'],
    'y': ['x'],
    'w': ['y', 'z'],
    'z': ['z'],
}
```

An undirected graph is represented the same way, with each edge appearing in both adjacency lists.

1 Graph Representation (10 min)

1.1 Build an adjacency list from edges

Write a function `edges_to_graph(edges, directed=True)` that takes a list of pairs `(u, v)` and returns the adjacency-list dictionary. If `directed=False`, each edge (u, v) should appear in both directions. Make sure that isolated vertices (those that appear as a source but have no listed neighbor, and vice versa) are included as keys with an empty list.

```
def edges_to_graph(edges, directed=True):
    """Build an adjacency-list graph from a list of (u, v) pairs.
    If directed=False, each edge is added in both directions."""
    # YOUR CODE HERE
    pass
```

```
# Test
edges = [('A','B'), ('A','D'), ('B','C'), ('B','D'), ('C','F'),
         ('D','E'), ('E','F')]
g = edges_to_graph(edges, directed=False)
assert set(g['A']) == {'B', 'D'}
assert set(g['F']) == {'C', 'E'}
print("edges_to_graph test passed!")
```

1.2 Transpose a directed graph

Write a function `transpose(graph)` that returns a new directed graph G^T where every edge (u, v) in the input becomes (v, u).

```python
def transpose(graph):
    """Return G^T: every edge (u,v) becomes (v,u)."""
    # YOUR CODE HERE
    pass
```

```python
# Test
g = {'a': ['b'], 'b': ['c'], 'c': ['a']}
gt = transpose(g)
assert gt == {'a': ['c'], 'b': ['a'], 'c': ['b']}
print("transpose test passed!")
```

2 Breadth-First Search (15 min)

Recall

BFS explores a graph level by level from a source s using a FIFO queue. It computes the **shortest-path distance** $\delta(s, v)$ (number of edges) for every reachable vertex v, and builds a **BFS tree** via parent pointers. Runs in $\mathcal{O}(V + E)$.

2.1 Implement BFS

```python
from collections import deque

def bfs(graph, s):
    """BFS from source s. Return (dist, parent) dictionaries.
    dist[v] = shortest distance from s (inf if unreachable).
    parent[v] = predecessor on shortest path (None for s)."""
    # YOUR CODE HERE
    pass
```

```python
# Test on the undirected graph from Section 1
dist, parent = bfs(g, 'A')
assert dist == {'A': 0, 'B': 1, 'C': 2, 'D': 1, 'E': 2, 'F': 3}
print("BFS distances test passed!")
```

2.2 Shortest path reconstruction

Using the `parent` dictionary returned by BFS, write a function `shortest_path(graph, s, t)` that returns the list of vertices on a shortest s-t path (inclusive), or `None` if t is not reachable from s.

```python
def shortest_path(graph, s, t):
    """Return the vertex list of a shortest s-t path, or None."""
    # YOUR CODE HERE
    pass
```

```python
# Test
path = shortest_path(g, 'A', 'F')
assert path is not None and len(path) == 4
```

```
assert path[0] == 'A' and path[-1] == 'F'
# ['A', 'B', 'C', 'F'] or ['A', 'D', 'E', 'F'] --- both valid
print(f"Shortest path A->F: {' -> '.join(path)}")
```

2.3 Connected components

Write a function `connected_components(graph)` that returns the list of connected components of an undirected graph. Each component is a set of vertices. Use BFS internally.

```
def connected_components(graph):
    """Return a list of sets, one per connected component."""
    # YOUR CODE HERE
    pass
```

```
# Test
g2 = edges_to_graph([('A','B'), ('C','D')], directed=False)
cc = connected_components(g2)
assert len(cc) == 2
assert {'A','B'} in cc and {'C','D'} in cc
print("Connected components test passed!")
```

3 Depth-First Search (20 min)

Recall

DFS explores a graph by going as deep as possible before backtracking. It assigns **discovery time** $d[v]$ and **finish time** $f[v]$ to each vertex. Edges are classified as **tree**, **back**, **forward**, or **cross** based on colors/timestamps. Runs in $\mathcal{O}(V + E)$.

3.1 DFS with timestamps

Implement DFS with full timestamps. The function should iterate over vertices in the order given by the dictionary (as in the lecture code) and return three dictionaries: `d` (discovery times), `f` (finish times), and `parent`.

```
def dfs(graph):
    """DFS with timestamps. Return (d, f, parent) dictionaries."""
    # YOUR CODE HERE
    pass
```

```
# Test on the graph from the introduction
d, f, parent = dfs(graph)
assert d['u'] == 1 and f['u'] == 8
assert d['w'] == 9 and f['w'] == 12
print("DFS timestamps test passed!")
```

3.2 Edge classification

Write a function `classify_edges(graph)` that runs DFS and returns a dictionary mapping each edge (u, v) to one of `'tree'`, `'back'`, `'forward'`, or `'cross'`.

Recall

Edge type	Condition when exploring (u, v)
Tree	v is white (undiscovered)
Back	v is gray (ancestor on the current path)
Forward / Cross	v is black (already finished)
Forward	$d[u] < d[v]$ (descendant)
Cross	$d[u] > d[v]$ (neither ancestor nor descendant)

```python
def classify_edges(graph):
    """Run DFS and classify every edge as tree/back/forward/cross."""
    # YOUR CODE HERE
    pass
```

```python
# Test
types = classify_edges(graph)
assert types[('u','v')] == 'tree'
assert types[('x','v')] == 'back'
assert types[('u','x')] == 'forward'
assert types[('w','y')] == 'cross'
print("Edge classification test passed!")
```

3.3 Cycle detection

Write a function `has_cycle(graph)` that returns `True` if the directed graph contains a cycle, `False` otherwise. Your algorithm must run in $\mathcal{O}(V + E)$ time.

Hint

A directed graph has a cycle if and only if DFS discovers a **back edge**.

```python
def has_cycle(graph):
    """Return True if the directed graph contains a cycle (O(V+E))."""
    # YOUR CODE HERE
    pass
```

```python
# Test
assert has_cycle(graph) == True
dag = {'a': ['b', 'c'], 'b': ['c'], 'c': []}
assert has_cycle(dag) == False
print("Cycle detection test passed!")
```

4 Topological Sort (15 min)

Recall

A **topological sort** of a DAG is a linear ordering of vertices such that for every edge (u, v), vertex u appears before v. Two approaches: (1) DFS-based (vertices ordered by decreasing finish time), (2) Kahn's algorithm (BFS-based, using in-degree counting).

4.1 DFS-based topological sort

Implement a topological sort using DFS. The function should raise a `ValueError` if the graph has a cycle.

```python
def topological_sort(graph):
    """DFS-based topological sort. Return a list of vertices.
    Raises ValueError if the graph has a cycle."""
    # YOUR CODE HERE
    pass
```

```python
# Test
courses = {
    'MATH101': ['CS201'],
    'CS101':   ['CS201', 'CS202'],
    'CS201':   ['CS301'],
    'CS202':   ['CS301'],
    'CS301':   [],
}
order = topological_sort(courses)
assert order.index('MATH101') < order.index('CS201')
assert order.index('CS101') < order.index('CS202')
assert order.index('CS201') < order.index('CS301')
print(f"Topological order: {order}")
```

4.2 Kahn's algorithm (BFS-based)

Implement Kahn's algorithm for topological sort. Compare the output with the DFS-based version.

```python
def topological_sort_kahn(graph):
    """BFS-based topological sort (Kahn's algorithm).
    Raises ValueError if the graph has a cycle."""
    # YOUR CODE HERE
    # 1. Compute in-degree of every vertex
    # 2. Initialize a queue with all vertices of in-degree 0
    # 3. Repeatedly dequeue, append to result, decrement neighbors'
    #    in-degree; enqueue those reaching 0
    # 4. If |result| < |V|, the graph has a cycle
    pass
```

```python
# Test
order_kahn = topological_sort_kahn(courses)
assert order_kahn.index('MATH101') < order_kahn.index('CS201')
assert order_kahn.index('CS201') < order_kahn.index('CS301')
print(f"Kahn's order: {order_kahn}")
```

Question: Can the two algorithms produce different valid orderings? Why?

5 Strongly Connected Components (15 min)

Recall

Kosaraju's algorithm finds all SCCs in $\mathcal{O}(V + E)$:

1. Run DFS on G and record the **finish order**.
2. Compute the transpose G^T.
3. Run DFS on G^T, processing vertices in **reverse finish order**. Each DFS tree in step 3 is

an SCC.

5.1 Kosaraju's algorithm

Implement Kosaraju's algorithm. Use your `transpose` function from Section 1.

```python
def kosaraju(graph):
    """Kosaraju's algorithm. Return a list of SCCs (each a list)."""
    # YOUR CODE HERE
    pass
```

```python
# Test
g_scc = {
    1: [2], 2: [3, 4], 3: [1],
    4: [5], 5: [6], 6: [4],
}
sccs = kosaraju(g_scc)
scc_sets = [set(c) for c in sccs]
assert {1,2,3} in scc_sets
assert {4,5,6} in scc_sets
print(f"SCCs: {sccs}")
```

5.2 Condensation DAG

Write a function `scc_dag(graph, sccs)` that takes a graph and the list of SCCs returned by Kosaraju's algorithm, and returns the **DAG of SCCs** (the condensation graph): each SCC is contracted into a single vertex, and there is an edge from SCC C_i to SCC C_j ($i \neq j$) if there exists an edge from some vertex in C_i to some vertex in C_j in the original graph.

```python
def scc_dag(graph, sccs):
    """Build the condensation DAG from a graph and its SCCs."""
    # YOUR CODE HERE
    pass
```

```python
# Test
dag = scc_dag(g_scc, sccs)
# Two SCCs, one edge from {1,2,3} -> {4,5,6}
assert len(dag) == 2
print(f"Condensation DAG: {dag}")
```

Bonus Exercise

Bonus — Autograd Computation Graph

The loss function $L = (w_1 x_1 + w_2 x_2 - y)^2$ can be decomposed into elementary operations:

$$a = w_1 x_1, \qquad b = w_2 x_2, \qquad c = a + b, \qquad d = c - y, \qquad L = d^2.$$

1. Model this computation as a DAG: each intermediate variable is a vertex, and there is an edge from each operand to the operation that uses it.
2. Use `topological_sort` to obtain the **forward-pass** order.
3. Reverse the topological order to obtain the **backward-pass** order (needed for backpropagation).

```python
comp_graph = {
    'w1': ['a'], 'x1': ['a'],
    'w2': ['b'], 'x2': ['b'],
    'a':  ['c'], 'b':  ['c'],
    'y':  ['d'], 'c':  ['d'],
    'd':  ['L'], 'L':  [],
}
fwd = topological_sort(comp_graph)
assert fwd.index('a') < fwd.index('c')
assert fwd.index('d') < fwd.index('L')
bwd = fwd[::-1]
print("Forward:", fwd)
print("Backward:", bwd)
```

Question: Why is the reverse topological order the correct order for backpropagation?

Key Complexities

BFS / DFS	$\mathcal{O}(V + E)$
Connected components (BFS)	$\mathcal{O}(V + E)$
Edge classification (DFS)	$\mathcal{O}(V + E)$
Topological sort (DFS / Kahn)	$\mathcal{O}(V + E)$
Kosaraju's SCC	$\mathcal{O}(V + E)$
Condensation DAG	$\mathcal{O}(V + E)$

This is the last lab of the semester — good luck on the final exam!