1 Graph Representation

1.1 Build an adjacency list from edges

Solution

```python
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."""
    graph = {}
    for u, v in edges:
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        if not directed:
            graph[v].append(u)
    return graph
```

Key points:

- Both endpoints must be added as keys, even if they only appear as targets.
- For undirected graphs, each edge produces two directed entries.

Common Mistakes

- Forgetting to add target vertices as keys (so some vertices are missing from the dictionary).
- For undirected graphs, adding each edge only once (missing the reverse direction).

1.2 Transpose a directed graph

Solution

```python
def transpose(graph):
    """Return G^T: every edge (u,v) becomes (v,u)."""
    gt = {u: [] for u in graph}
    for u in graph:
        for v in graph[u]:
            gt[v].append(u)
    return gt
```

Initialize all keys first (even vertices with no incoming edges need to appear), then reverse each edge. Complexity: $\mathcal{O}(V + E)$.

2 Breadth-First Search

2.1 Implement BFS

Solution

```python
from collections import deque

def bfs(graph, s):
    """BFS from source s.  Return (dist, parent) dictionaries."""
    dist = {v: float('inf') for v in graph}
    parent = {v: None for v in graph}
    dist[s] = 0
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v in graph[u]:
            if dist[v] == float('inf'):
                dist[v] = dist[u] + 1
                parent[v] = u
                queue.append(v)
    return dist, parent
```

Output:

```
dist = {'A': 0, 'B': 1, 'C': 2, 'D': 1, 'E': 2, 'F': 3}
BFS distances test passed!
```

Explanation

BFS uses `dist[v] == float('inf')` as the "white" check (unvisited). This is equivalent to maintaining a separate `color` dictionary but simpler. Each vertex is enqueued and dequeued exactly once, giving $\mathcal{O}(V + E)$ total.

Common Mistakes

- Using a list and `pop(0)` instead of `deque.popleft()`: correct but $\mathcal{O}(V^2)$ instead of $\mathcal{O}(V + E)$.
- Forgetting to initialize distances for all vertices (crash if a vertex is unreachable).

2.2 Shortest path reconstruction

Solution

```python
def shortest_path(graph, s, t):
    """Return the vertex list of a shortest s-t path, or None."""
    dist, parent = bfs(graph, s)
    if dist[t] == float('inf'):
        return None
    path = []
    v = t
    while v is not None:
        path.append(v)
        v = parent[v]
    return path[::-1]
```

Output:

```
Shortest path A->F: A -> B -> C -> F
```

The path is reconstructed by following parent pointers from t back to s, then reversing. Alternative valid path: `A -> D -> E -> F`.

2.3 Connected components

Solution

```python
def connected_components(graph):
    """Return a list of sets, one per connected component."""
    visited = set()
    components = []
    for s in graph:
        if s not in visited:
            comp = set()
            queue = deque([s])
            visited.add(s)
            while queue:
                u = queue.popleft()
                comp.add(u)
                for v in graph[u]:
                    if v not in visited:
                        visited.add(v)
                        queue.append(v)
            components.append(comp)
    return components
```

Simply run BFS from each unvisited vertex, collecting all reachable vertices into a component. Total complexity: $\mathcal{O}(V + E)$ since each vertex and edge is processed at most once.

Grading Notes

Accept any correct BFS-based approach (or DFS-based, if the student uses DFS). The key requirement is $\mathcal{O}(V + E)$ total time across all components.

3 Depth-First Search

3.1 DFS with timestamps

Solution

```python
def dfs(graph):
    """DFS with timestamps.  Return (d, f, parent) dictionaries."""
    color = {v: 'W' for v in graph}
    d, f = {}, {}
    parent = {v: None for v in graph}
    time = [0]

    def dfs_visit(u):
        time[0] += 1
        d[u] = time[0]
        color[u] = 'G'
        for v in graph[u]:
            if color[v] == 'W':
                parent[v] = u
                dfs_visit(v)
        time[0] += 1
        f[u] = time[0]
        color[u] = 'B'

    for u in graph:
        if color[u] == 'W':
            dfs_visit(u)
    return d, f, parent
```

Output on the reference graph:

```
d = {'u': 1, 'v': 2, 'y': 3, 'x': 4, 'w': 9, 'z': 10}
f = {'u': 8, 'v': 7, 'y': 6, 'x': 5, 'w': 12, 'z': 11}
DFS timestamps test passed!
```

Note: `time` is wrapped in a list so that the nested function `dfs_visit` can modify it (Python closure semantics).

Common Mistakes

- Using a plain integer for `time` instead of a mutable container — the nested function cannot rebind a variable in the enclosing scope (Python 3 requires `nonlocal` or a list/object).
- Forgetting to iterate over all vertices in the outer loop (only calling `dfs_visit` on the first vertex), which misses disconnected components.
- Incrementing the timer only on discovery, not on finish.

3.2 Edge classification

Solution

```python
def classify_edges(graph):
    """Run DFS and classify every edge as tree/back/forward/cross."""
    color = {v: 'W' for v in graph}
    d, f = {}, {}
    time = [0]
    edge_type = {}

    def dfs_visit(u):
        time[0] += 1
        d[u] = time[0]
        color[u] = 'G'
        for v in graph[u]:
            if color[v] == 'W':
                edge_type[(u, v)] = 'tree'
                dfs_visit(v)
            elif color[v] == 'G':
                edge_type[(u, v)] = 'back'
            else:  # color[v] == 'B'
                if d[u] < d[v]:
                    edge_type[(u, v)] = 'forward'
                else:
                    edge_type[(u, v)] = 'cross'
        time[0] += 1
        f[u] = time[0]
        color[u] = 'B'

    for u in graph:
        if color[u] == 'W':
            dfs_visit(u)
    return edge_type
```

Output:

```
(u,v) = tree,  (v,y) = tree,  (y,x) = tree
(x,v) = back,  (u,x) = forward
(w,y) = cross, (w,z) = tree,  (z,z) = back
Edge classification test passed!
```

Explanation

The classification logic:

- **White** → tree edge (extends the DFS tree).
- **Gray** → back edge (points to an ancestor on the current recursion stack).
- **Black** with $d[u] < d[v]$ → forward edge (points to a descendant already finished).
- **Black** with $d[u] > d[v]$ → cross edge (points to a vertex in a different subtree or component).

Note: in an *undirected* graph, there are only tree and back edges (no forward or cross).

3.3 Cycle detection

Solution

```python
def has_cycle(graph):
    """Return True if the directed graph contains a cycle (O(V+E))."""
    color = {v: 'W' for v in graph}

    def dfs_visit(u):
        color[u] = 'G'
        for v in graph[u]:
            if color[v] == 'G':
                return True          # back edge -> cycle
            if color[v] == 'W' and dfs_visit(v):
                return True
        color[u] = 'B'
        return False

    for u in graph:
        if color[u] == 'W':
            if dfs_visit(u):
                return True
    return False
```

A directed graph is acyclic if and only if DFS finds **no back edges**. A back edge (u, v) means v is an ancestor of u in the DFS tree, so the path from v to u in the tree plus the edge (u, v) forms a cycle.

Common Mistakes

- Checking only for *visited* vertices instead of *gray* vertices: a cross edge to a black vertex does *not* indicate a cycle.
- Using the full `classify_edges` function: correct but wasteful since we can stop early at the first back edge.

4 Topological Sort

4.1 DFS-based topological sort

Solution

```python
def topological_sort(graph):
    """DFS-based topological sort. Raises ValueError on cycle."""
    color = {v: 'W' for v in graph}
    order = []

    def dfs_visit(u):
        color[u] = 'G'
        for v in graph[u]:
            if color[v] == 'G':
                raise ValueError("Graph has a cycle!")
            if color[v] == 'W':
                dfs_visit(v)
        color[u] = 'B'
        order.append(u)

    for u in graph:
        if color[u] == 'W':
            dfs_visit(u)
    return order[::-1]
```

Output:

```
Topological order: ['CS101', 'MATH101', 'CS202', 'CS201', 'CS301']
```

Vertices are appended when they *finish* (turn black), so the list is in reverse topological order. We reverse it at the end. This is equivalent to sorting by decreasing finish time.

Explanation

Correctness: For every edge (u, v) in a DAG, u finishes *after* v during DFS (because u cannot be finished while v is still being explored). Therefore, in the reversed finish-order list, u appears before v.

4.2 Kahn's algorithm (BFS-based)

Solution

```python
def topological_sort_kahn(graph):
    """BFS-based topological sort (Kahn's algorithm)."""
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    queue = deque(u for u in graph if in_degree[u] == 0)
    order = []

    while queue:
        u = queue.popleft()
        order.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    if len(order) != len(graph):
        raise ValueError("Graph has a cycle!")
    return order
```

Output:

```
Kahn's order: ['MATH101', 'CS101', 'CS201', 'CS202', 'CS301']
```

Answer to the question: Yes, the two algorithms can produce different valid orderings. Any permutation respecting all precedence constraints is a valid topological sort. The DFS-based version depends on the iteration order of the dictionary and the recursion structure; Kahn's depends on which zero-in-degree vertices are dequeued first. Both are correct.

Grading Notes

Accept any valid topological ordering. To verify: for every edge (u, v), check that u appears before v in the output.

5 Strongly Connected Components

5.1 Kosaraju's algorithm

Solution

```python
def kosaraju(graph):
    """Kosaraju's algorithm. Return list of SCCs (each a list)."""
    # Step 1: DFS on G, record finish order
    visited = set()
    finish_order = []

    def dfs1(u):
        visited.add(u)
        for v in graph[u]:
            if v not in visited:
                dfs1(v)
        finish_order.append(u)

    for u in graph:
        if u not in visited:
            dfs1(u)

    # Step 2: transpose
    gt = transpose(graph)

    # Step 3: DFS on G^T in reverse finish order
    visited.clear()
    sccs = []

    def dfs2(u, component):
        visited.add(u)
        component.append(u)
        for v in gt[u]:
            if v not in visited:
                dfs2(v, component)

    for u in reversed(finish_order):
        if u not in visited:
            comp = []
            dfs2(u, comp)
            sccs.append(comp)
    return sccs
```

Output:

```
SCCs: [[1, 3, 2], [4, 6, 5]]
  -> sets: {1,2,3} and {4,5,6}
```

Why it works:

1. The first DFS computes finish times. Vertices in "root" SCCs (no incoming inter-SCC edges) finish last.
2. Processing in reverse finish order on G^T ensures we start from an SCC with no incoming edges in the component graph. DFS on G^T from such a vertex explores exactly that SCC.

Common Mistakes

- Running the second DFS on G instead of G^T.
- Processing vertices in finish order (not *reverse* finish order) in step 3.
- Forgetting to clear the `visited` set between steps 1 and 3.
- Using an iterative DFS but forgetting to append to `finish_order` at the correct time (must be

after all neighbors are explored).

Explanation

Complexity: $\mathcal{O}(V + E)$ — two DFS traversals plus one transpose, each $\mathcal{O}(V + E)$.
Alternative: Tarjan's algorithm finds SCCs with a single DFS pass using a stack and "low-link" values. It is conceptually harder but avoids computing the transpose.

5.2 Condensation DAG

Solution

```python
def scc_dag(graph, sccs):
    """Build the condensation DAG from a graph and its SCCs."""
    vertex_to_scc = {}
    for i, comp in enumerate(sccs):
        for v in comp:
            vertex_to_scc[v] = i

    n = len(sccs)
    dag = {i: [] for i in range(n)}
    seen_edges = set()
    for u in graph:
        for v in graph[u]:
            su, sv = vertex_to_scc[u], vertex_to_scc[v]
            if su != sv and (su, sv) not in seen_edges:
                dag[su].append(sv)
                seen_edges.add((su, sv))
    return dag
```

Output:

```
Condensation DAG: {0: [1], 1: []}
  (SCC 0 = {1,2,3} -> SCC 1 = {4,5,6})
```

The `seen_edges` set prevents duplicate edges in the condensation. Without it, multiple edges between the same pair of SCCs would be added.

Grading Notes

The condensation DAG should have exactly k vertices (one per SCC) and no self-loops. Duplicate edges should be removed. Accept any vertex labeling scheme (integer indices, frozensets, etc.).

Bonus — Autograd Computation Graph

Solution

```python
def autograd_demo():
    """Model a computation graph and derive forward/backward order."""
    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)
    bwd = fwd[::-1]
    print("Forward pass order: ", fwd)
    print("Backward pass order:", bwd)
    assert fwd.index('a') < fwd.index('c')
    assert fwd.index('b') < fwd.index('c')
    assert fwd.index('c') < fwd.index('d')
    assert fwd.index('d') < fwd.index('L')
    return fwd, bwd
```

Output:

```
Forward pass order:  ['w2', 'x2', 'w1', 'x1', 'y', 'b', 'a', 'c', 'd', 'L']
Backward pass order: ['L', 'd', 'c', 'a', 'b', 'y', 'x1', 'w1', 'x2', 'w2']
```

Why reverse topological order is correct for backpropagation:
In backpropagation, we need to compute $\frac{\partial L}{\partial v}$ for every node v. By the chain rule:

$$\frac{\partial L}{\partial v} = \sum_{w:(v,w)\in E} \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial v}$$

This means we need $\frac{\partial L}{\partial w}$ (for all successors w of v) *before* we can compute $\frac{\partial L}{\partial v}$. In the reversed topological order, L comes first (gradient $= 1$), then each node is processed only after all its successors, ensuring all required gradients are available.