Directed Eulerian Cycle

Learning Objectives

By the end of this lab, you will be able to:

- Implement Hierholzer's algorithm for finding Eulerian cycles
- Handle self-loops and parallel edges in directed graphs
- Manage edge traversal state efficiently

Problem Statement

A *directed Eulerian cycle* is a directed cycle that uses every edge of a digraph exactly once. Write a digraph client that finds a directed Eulerian cycle or reports that no such cycle exists.

When does a directed Eulerian cycle exist?

A directed graph has an Eulerian cycle if and only if:

1. The graph is *strongly connected* (every vertex is reachable from every other vertex), and
2. $\text{indegree}(v) = \text{outdegree}(v)$ for every vertex v.

Note: For this lab, we assume the input digraph is already strongly connected. Your implementation should verify the degree condition.

Reminder: The input graph may contain

- self-loops (edges of the form $v \to v$), and
- parallel edges (multiple edges between the same ordered pair of vertices).

Your implementation must handle these correctly. Self-loops and parallel edges may appear multiple times in the adjacency list and are traversed like any other edge. No special-case logic is required.

Algorithm outline (Hierholzer's algorithm)

1. Choose any vertex v with nonzero outdegree (or any vertex if all have edges).
2. Starting from v, follow unused outgoing edges arbitrarily, marking each edge as used. Since indegree equals outdegree everywhere, you will eventually return to v, forming a cycle C.
3. If C contains all edges of the digraph, output C as the Eulerian cycle and terminate.
4. Otherwise, find a vertex u on C that still has unused outgoing edges.
5. Starting from u, form a new cycle C' using only unused edges (again, you will return to u).
6. *Splice* C' into C at vertex u: replace u in C with the entire sequence from C'.
7. Repeat steps 4–6 until all edges are used.

If $\text{indegree}(v) = \text{outdegree}(v)$ for all vertices v, then every time you enter a vertex via an edge, there is always at least one unused outgoing edge available. This ensures that a cycle can always be completed.

Implementation Strategy: Marking Used Edges

Since the `Digraph` class does not provide unique edge identifiers, you must track which edges have been used. The recommended approach:
Use vectors for adjacency lists with index tracking:
Change the adjacency representation from linked lists to vectors in `Digraph` class. The definition of `_adj` should be:

```
vector<vector<int>> _adj;  // adjacency lists as vectors
```

and the `adj(int v)` method should return a reference to the vector:

```
const vector<int>& adj(int v) const {
    return _adj[v];
}
```

Data Structure Setup

Your `EulerianCycle` class will need at minimum:

```
class EulerianCycle {
private:
    Digraph G;              // the input digraph
    vector<int> next;       // next[v] = index of next unused edge from v
    deque<int> cycle;       // stores the final Eulerian cycle

    // Your methods here

public:
    EulerianCycle(...) {
        // 1. Initialize G and next
        // 2. Check if Eulerian cycle exists (degree condition)
        // 3. If yes, find the cycle
        // 4. If no, set appropriate state
    }

    bool hasEulerianCycle() const {
        // Return whether cycle exists
    }

    deque<int> getCycle() const {
        // Return the cycle
    }
};
```

Key Implementation Hints

1. **Degree Checking:** Before attempting to find a cycle, verify that every vertex has equal indegree and outdegree. You'll need to count degrees by iterating through all adjacency lists.
2. **Edge Traversal:** The `next` array tracks which edge to use next from each vertex. When at vertex v:
   - Access the next edge: `G.adj(v)[next[v]]`
   - Mark it as used by incrementing: `next[v]++`
   - Follow the edge to the next vertex
3. **Building the Cycle:** Consider using a recursive or iterative approach to follow edges. Think about:

- When do you add a vertex to the cycle?
- Should you add to the front or back of the deque?
- How do you know when to stop?

4. **Automatic Splicing:** If you structure your traversal correctly, the splicing happens naturally. When you finish exploring all edges from a vertex, that's when you should add it to your cycle structure.

Approach Options

You have two main approaches to choose from:
Option 1: Recursive DFS

- Follow edges recursively until no more unused edges from current vertex
- Add vertices to the cycle as you backtrack
- Natural and elegant, but uses call stack

Option 2: Iterative with Explicit Stack

- Maintain your own stack of vertices
- Explicitly track the current path
- More control, but requires careful stack management

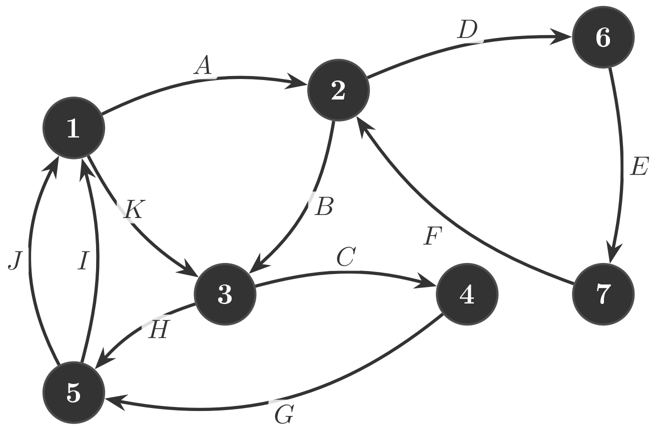
Think about: What order should vertices be added to ensure the final sequence forms a valid cycle?

Example

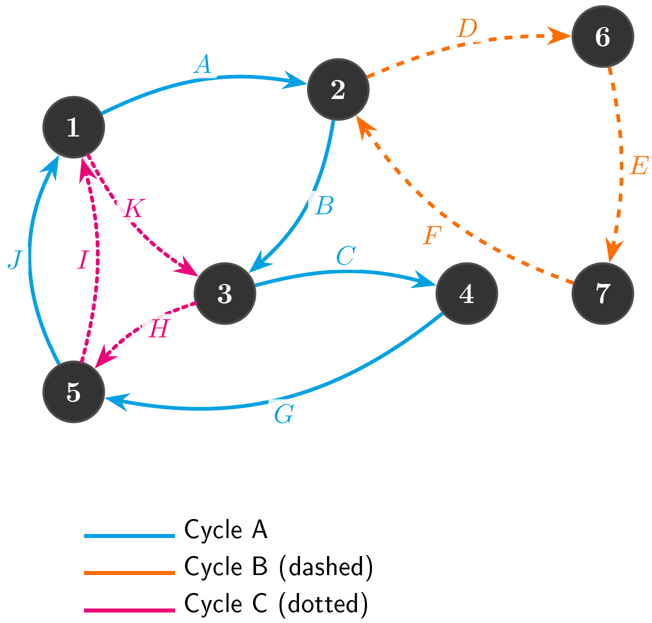
Consider the following digraph with 7 vertices and 11 edges:

Edge	**From → To**	**Edge**	**From → To**
A	$1 \to 2$	G	$4 \to 5$
B	$2 \to 3$	H	$3 \to 5$
C	$3 \to 4$	I	$5 \to 1$
D	$2 \to 6$	J	$5 \to 1$
E	$6 \to 7$	K	$1 \to 3$
F	$7 \to 2$		

Note that all vertices have equal indegree and outdegree, confirming an Eulerian cycle exists.



We will find an Eulerian cycle using Hierholzer's algorithm:



- **Step 1: Initial Cycle.** Starting at vertex 1, we follow unused edges to form a cycle:

$$1 \xrightarrow{A} 2 \xrightarrow{B} 3 \xrightarrow{C} 4 \xrightarrow{G} 5 \xrightarrow{J} 1.$$

  This gives our first cycle:

$$C_1 = (1, 2, 3, 4, 5, 1).$$

- **Step 2: Expand from Vertex 2.** Vertex 2 still has unused outgoing edges. Starting a new walk from vertex 2:

$$2 \xrightarrow{D} 6 \xrightarrow{E} 7 \xrightarrow{F} 2.$$

  We splice this subcycle into C_1 at vertex 2, producing:

$$C_2 = (1, 2, 6, 7, 2, 3, 4, 5, 1).$$

- **Step 3: Expand from Vertex 3.** Vertex 3 has unused outgoing edges. We form another cycle:

$$3 \xrightarrow{H} 5 \xrightarrow{I} 1 \xrightarrow{K} 3.$$

  Splicing this into C_2 at vertex 3 yields:

$$C_3 = (1, 2, 6, 7, 2, 3, 5, 1, 3, 4, 5, 1).$$

- **Step 4: Completion.** All edges have now been traversed exactly once. The final sequence C_3 represents a valid Eulerian cycle, visiting every directed edge in the graph precisely once and returning to the starting vertex.

Note: The repeated vertices (such as the multiple occurrences of 1, 2, 3, and 5) naturally arise from splicing subcycles and are characteristic of the algorithm's construction process.

Testing Your Implementation

Correctness checks:

1. Verify that $\text{indegree}(v) = \text{outdegree}(v)$ for all vertices v.
2. Ensure the cycle contains exactly $|E|$ edges (same as `G.E()`).

3. Confirm the first and last vertices are identical.
4. Verify each directed edge appears exactly once.

Test cases to consider:

- Single vertex with self-loop
- Complete graph with 3 vertices
- Graph with parallel edges
- Graph with no Eulerian cycle (unequal degrees)
- The example graph from this document