Problem Set #8: Directed Graphs
SCC, Euler Tours, and Topological Sorting

Orysbay Ulykpan

Contents

1 Introduction

This report provides solutions and clear explanations for three tasks on directed graphs from Problem Set #8. The main purpose is educational: the text is written in simple language, with small logical steps, so that the reader can understand not only the answers but also the reasoning.

The tasks are:

- **Problem 1:** Strongly Connected Components (SCC) and reversing a directed graph.
- **Problem 2:** Euler tours in strongly connected directed graphs.
- **Problem 3:** Topological sorting for a given acyclic directed graph.

Assumptions used throughout:

- graphs are directed,
- graphs are stored using adjacency lists,
- V is the vertex set and E is the edge set.

2 Problem 1: SCC and Reversal

Let $G = (V, E)$ be a directed graph.

2.1 1.1 Computing $\mathrm{rev}(G)$ in $O(V + E)$ time

2.1.1 Meaning of $\mathrm{rev}(G)$

The reversed graph $\mathrm{rev}(G)$ keeps the same vertices but flips every edge direction. So if $(u, v) \in E$ in G, then (v, u) becomes an edge in $\mathrm{rev}(G)$. Intuitively, we keep the same nodes but reverse all arrows.

2.1.2 Input representation

We assume adjacency lists:

$$\mathrm{Adj}[u] = \{v \mid (u, v) \in E\}.$$

That is, for each vertex u we store all outgoing neighbors.

2.1.3 Construction procedure

We build adjacency lists for the reversed graph, called RevAdj.

1) For each vertex $x \in V$, initialize an empty list RevAdj$[x]$.
2) For each vertex $u \in V$:
   a) For every $v \in \mathrm{Adj}[u]$, add u into RevAdj$[v]$.

After the loop ends, RevAdj is exactly the adjacency list representation of $\mathrm{rev}(G)$.

2.1.4 Why the time is $O(V + E)$

- Initializing empty lists for all vertices costs $O(V)$.
- Each directed edge (u, v) appears exactly once in adjacency lists, so it is processed once.
- Processing an edge performs only constant work (one insertion).

Therefore the total time is $O(V) + O(E) = O(V + E)$.

2.2 1.2 Why $\mathrm{scc}(G)$ is always acyclic

2.2.1 What is an SCC

A strongly connected component is a maximal group of vertices C such that every pair of vertices in C can reach each other (there is a path both ways).

2.2.2 What is the SCC graph

The graph $\mathrm{scc}(G)$ (condensation graph) is formed by:

- compressing each SCC into one super-vertex,
- drawing an edge $C_1 \to C_2$ if in G there exists at least one edge from a vertex in C_1 to a vertex in C_2, where $C_1 \neq C_2$.

2.2.3 Claim

$\mathrm{scc}(G)$ cannot contain a directed cycle.

2.2.4 Reasoning

Assume, for contradiction, that SCCs form a directed cycle:

$$C_1 \to C_2 \to \cdots \to C_k \to C_1.$$

Inside each C_i, all vertices are mutually reachable. Because the cycle gives directed connections between components, we can travel from any component to the next and eventually return. As a result, every vertex in $C_1 \cup \cdots \cup C_k$ can reach every other vertex in this union.

That means the union is one strongly connected set, so it should be a single SCC. But SCCs are maximal and disjoint, so having a cycle of different SCCs is impossible. Hence $\mathrm{scc}(G)$ is acyclic (a DAG).

2.3 1.3 Why $\mathrm{scc}(\mathrm{rev}(G)) = \mathrm{rev}(\mathrm{scc}(G))$

We compare two constructions:

- First reverse G, then take SCCs: $\mathrm{scc}(\mathrm{rev}(G))$.
- First take SCCs, then reverse edges between components: $\mathrm{rev}(\mathrm{scc}(G))$.

Reversing all edges does not change which vertices are mutually reachable in both directions. If u reaches v and v reaches u in G, the same remains true after reversal (paths just go backwards). So the partition of vertices into SCCs stays the same.

However, any edge that went from component C_1 to component C_2 in G will go from C_2 to C_1 in $\mathrm{rev}(G)$. Therefore the SCC graph structure remains, but with all inter-component edges reversed:

$$\mathrm{scc}(\mathrm{rev}(G)) = \mathrm{rev}(\mathrm{scc}(G)).$$

2.4 1.4 Reachability in G vs reachability in $\mathrm{scc}(G)$

Let $S(x)$ be the SCC containing vertex x.

We want to justify:

$$u \text{ reaches } v \text{ in } G \iff S(u) \text{ reaches } S(v) \text{ in } \mathrm{scc}(G).$$

2.4.1 (⇒) From vertices to components

If there is a path $u = x_0 \to x_1 \to \cdots \to x_k = v$ in G, then each x_i lies in some SCC. Whenever the path moves from one SCC to a different SCC, that implies an edge between those SCCs in $\mathrm{scc}(G)$. So compressing the path gives a path from $S(u)$ to $S(v)$ in $\mathrm{scc}(G)$.

2.4.2 (⇐) From components back to vertices

If $\mathrm{scc}(G)$ has a path $S(u) = C_0 \to C_1 \to \cdots \to C_m = S(v)$, then for each step $C_i \to C_{i+1}$ there exists at least one edge in G going from some vertex in C_i to some vertex in C_{i+1}. Also, inside each SCC we can move between any two vertices. So we can:

- move inside C_0 from u to the exit vertex,
- cross to C_1 by the connecting edge,
- repeat until reaching C_m,
- then move inside C_m to v.

This builds a real path from u to v in G.

3 Problem 2: Euler Tour

Let $G = (V, E)$ be a strongly connected directed graph.

3.1 2.1 When does an Euler tour exist?

3.1.1 Definition

An Euler tour is a closed walk that uses each edge exactly once. Vertices may be visited multiple times; the key requirement is *edges exactly once*.

3.1.2 Existence condition

A directed graph has an Euler tour exactly when:

- it is strongly connected (so all edges belong to one reachable structure),
- for every vertex v, $\deg^-(v) = \deg^+(v)$ (in-degree equals out-degree).

3.1.3 Why the condition is necessary

In an Euler tour, every time we enter a vertex along some edge, we must also leave it along another edge. Because the tour is closed (starts and ends at the same vertex), no vertex can have more entrances than exits or vice versa. So for each vertex:

$$\deg^-(v) = \deg^+(v).$$

3.1.4 Why the condition is sufficient (idea)

If degrees are balanced, then whenever we arrive at a vertex, there is an unused outgoing edge to continue (unless we are completing a cycle). Strong connectivity ensures we can always reach remaining unused edges and merge cycles. This is the intuition behind Hierholzer's method.

3.2 2.2 An $O(E)$ algorithm to find an Euler tour

3.2.1 Hierholzer's method (high level)

1) Start from any vertex and keep following unused outgoing edges until returning to the start. This produces a cycle.
2) If all edges are used, we are done.
3) Otherwise, find a vertex on the current cycle that still has unused outgoing edges, build another cycle from it, and splice (merge) this new cycle into the main cycle.
4) Repeat until no unused edges remain.

3.2.2 Why the time is $O(E)$

Each edge becomes used exactly once and then never returns to the "unused" state. With a pointer per vertex to track the next unused outgoing edge in its adjacency list, each edge is processed constant-time overall. Thus, total work is proportional to E, i.e. $O(E)$.

4 Problem 3: Topological Sort

We are given the directed edges:

$$A \to B,\ A \to C,\ B \to C,\ B \to D,\ C \to E,\ D \to E,\ D \to F,\ G \to F,\ G \to E.$$

4.1 4.1 What is a topological order?

A topological order is a listing of all vertices such that for every directed edge $u \to v$, vertex u appears before v. A topological order exists if and only if the graph is acyclic (a DAG).

4.2 4.2 Compute in-degrees

We compute in-degree (number of incoming edges) for each vertex:

- A: 0.
- B: 1 (from A).
- C: 2 (from A and B).
- D: 1 (from B).
- E: 3 (from C, D, G).
- F: 2 (from D, G).
- G: 0.

So initially the zero in-degree vertices are $\{A, G\}$.

4.3 4.3 Kahn's algorithm (high level)

1) Compute in-degrees.
2) Put all vertices with in-degree 0 into a set (or queue).
3) Repeatedly remove one vertex u from the set, output it, then "delete" all outgoing edges from u and update in-degrees. Any vertex that becomes in-degree 0 is added to the set.

Different choices when multiple vertices are available produce different valid orders.

4.4 4.4 Topological sort with preference for A

We choose A first whenever possible.

Start: available $\{A, G\}$. Choose A.

Order: A. Remove $A \to B$ and $A \to C$:

$$\deg(B) : 1 \to 0, \quad \deg(C) : 2 \to 1.$$

Available: $\{G, B\}$.

Choose B. Order: A, B. Remove $B \to C$ and $B \to D$:

$$\deg(C) : 1 \to 0, \quad \deg(D) : 1 \to 0.$$

Available: $\{G, C, D\}$.

Choose D. Order: A, B, D. Remove $D \to E$ and $D \to F$:

$$\deg(E) : 3 \to 2, \quad \deg(F) : 2 \to 1.$$

Available: $\{G, C\}$.

Choose C. Order: A, B, D, C. Remove $C \to E$:

$$\deg(E) : 2 \to 1.$$

Available: $\{G\}$.

Choose G. Order: A, B, D, C, G. Remove $G \to F$ and $G \to E$:

$$\deg(F) : 1 \to 0, \quad \deg(E) : 1 \to 0.$$

Available: $\{F, E\}$.

Choose F, then E. Final order:

$$A, B, D, C, G, F, E.$$

4.5 4.5 Another valid order (starting with G)

Now choose G first.

Start: available $\{A, G\}$. Choose G.

Order: G. Remove $G \to F$ and $G \to E$:

$$\deg(F) : 2 \to 1, \quad \deg(E) : 3 \to 2.$$

Available: $\{A\}$.

Choose A. Order: G, A. Remove $A \to B$ and $A \to C$:

$$\deg(B) : 1 \to 0, \quad \deg(C) : 2 \to 1.$$

Available: $\{B\}$.

Choose B. Order: G, A, B. Remove $B \to C$ and $B \to D$:

$$\deg(C) : 1 \to 0, \quad \deg(D) : 1 \to 0.$$

Available: $\{C, D\}$.

Choose D. Order: G, A, B, D. Remove $D \to E$ and $D \to F$:

$$\deg(E) : 2 \to 1, \quad \deg(F) : 1 \to 0.$$

Available: $\{C, F\}$.

Choose C. Order: G, A, B, D, C. Remove $C \to E$:

$$\deg(E) : 1 \to 0.$$

Available: $\{F, E\}$.

Choose F, then E. Final order:

$$G, A, B, D, C, F, E.$$

4.6 4.6 Many correct answers exist

Topological sorting is often not unique. Whenever the algorithm has more than one zero in-degree vertex available, any choice is allowed. The only rule is: for each edge $u \to v$, the vertex u must appear earlier than v.

So different valid choices at intermediate steps produce different correct topological orders.

5 Conclusion

This report discussed three key topics for directed graphs:

- how to build $\text{rev}(G)$ efficiently and how SCCs behave under reversal,
- when Euler tours exist and how to find one in linear time,
- how Kahn's algorithm produces topological orders and why multiple answers are possible.

The reasoning was written in a simple step-by-step style, focusing on understanding definitions, algorithm ideas, and correctness arguments.