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

Simple and Detailed Explanations

Student Report

Contents

1 Introduction

This report contains solutions and explanations for three problems from *Problem Set #8* on directed graphs.[1]

The problems are:

- Problem 1: Strongly Connected Components (SCC) and graph reversal.
- Problem 2: Euler tour in a strongly connected directed graph.
- Problem 3: Topological sort on a small example graph.

The main goal of this report is to explain everything in **very simple language**, step by step, so that a reader who is new to these topics can still understand:

- what each definition means,
- what each algorithm does,
- why each statement is true.

Throughout the report we assume:

- graphs are directed,
- graphs are represented by adjacency lists (for algorithms),
- V is the set of vertices, E is the set of edges.

2 Problem 1: SCC and Reversal

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

2.1 1.1 Algorithm to Compute $\mathrm{rev}(G)$ in $O(V + E)$ Time

What is the reversed graph?

The reversed graph $\mathrm{rev}(G)$ is a graph where:

- the set of vertices is the same as in G,
- every edge has opposite direction.

Formally, if $(u, v) \in E$ is an edge in G, then in $\mathrm{rev}(G)$ we have edge (v, u).
So we simply **flip all arrows**.

How is the graph stored?

We assume that G is stored using adjacency lists. For each vertex u we have:

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

This means: for each vertex u we have a list of all neighbors v such that there is an edge from u to v.

[1] Original statements: Fundamental Algorithm Techniques, Problem Set #8.

Algorithm (step by step)

We want to build adjacency lists for the reversed graph, call them RevAdj.

1) For every vertex $u \in V$, create an **empty** list:

$$\text{RevAdj}[u] \leftarrow \emptyset.$$

2) For every vertex $u \in V$:

   a) Look at each neighbor v in $\text{Adj}[u]$.

   b) In the reversed graph there is an edge $v \to u$.

   c) So we add u to the list $\text{RevAdj}[v]$.

3) When we finish, the graph $\text{rev}(G)$ is defined by the lists RevAdj.

Why is the running time $O(V + E)$?

We look at:

- Step 1: We create one empty list for each vertex. This is $O(V)$.
- Step 2: For each vertex u, we walk through its adjacency list $\text{Adj}[u]$.
- The total number of elements over all adjacency lists is exactly E (each edge appears once).
- For each edge we do only a constant amount of work (one insertion into a list).

So the total running time is:

$$O(V) + O(E) = O(V + E).$$

2.2 1.2 Why is $\text{scc}(G)$ Always Acyclic?

What is an SCC (Strongly Connected Component)?

A strongly connected component is a set of vertices C such that:

- For any two vertices $u, v \in C$, there is a path from u to v **and** a path from v to u.
- The set C is **maximal**: we cannot add any other vertex to C and keep this property.

What is $\text{scc}(G)$?

The graph $\text{scc}(G)$ is sometimes called the *condensation graph*:

- Each SCC becomes a single “super-vertex”.
- If there is any edge from a vertex in SCC C_1 to a vertex in SCC C_2 (with $C_1 \neq C_2$), then in $\text{scc}(G)$ we draw an edge $C_1 \to C_2$.

Claim

The graph $\mathrm{scc}(G)$ has **no directed cycles**. In other words, it is a DAG.

Proof idea in simple words

Assume the opposite: suppose there *is* a directed cycle between SCCs:

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

What does this mean?

- From C_1 we can go to C_2 (using some edge between them).
- From C_2 we can go to C_3, and so on.
- From C_k we can go back to C_1.

Because each C_i is strongly connected inside, and we also have paths between them, we can move:

- from any vertex in C_1 to any vertex in C_2,
- from any vertex in C_2 to any vertex in C_3,
- ...
- and back from C_k to C_1.

So, in fact, **every** vertex in the union $C_1 \cup \cdots \cup C_k$ can reach every other vertex in this union.

That means $C_1 \cup \cdots \cup C_k$ is one big strongly connected set. So they should all be merged into one single SCC, not into several SCCs.

This contradicts the definition of SCCs as *maximal* strongly connected sets.

Therefore, our assumption was wrong, and $\mathrm{scc}(G)$ has no directed cycles.

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

We want to explain in simple words what this equality means.

Two steps we compare

There are two ways to build a graph:

Way 1: First reverse G to get $\mathrm{rev}(G)$, then build its SCC graph:

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

Way 2: First build $\mathrm{scc}(G)$, then reverse this SCC graph:

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

The equality says: these two resulting graphs are **the same** (up to the same SCC labels).

Why SCCs do not change when we reverse all edges

Take two vertices u and v.

- Suppose in G there is a path from u to v and another from v to u.
- In $\text{rev}(G)$ each path is just walked in the opposite direction.

So if u and v are strongly connected in G, they are also strongly connected in $\text{rev}(G)$.

And the opposite is also true: if they are strongly connected in $\text{rev}(G)$, then they are strongly connected in G.

This means that the partition of the vertex set into SCCs is the same for both G and $\text{rev}(G)$.

What happens with edges between SCCs

Now look at edges between SCCs:

- If there is an edge from SCC C_1 to SCC C_2 in G, then in $\text{rev}(G)$ all such edges point from C_2 to C_1.
- So the SCC graph of $\text{rev}(G)$ is exactly the SCC graph of G, but with all edges reversed.

This is exactly the same as saying:

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

2.4 1.4 Reachability and SCCs

For every vertex v in G, let $S(v)$ be the SCC that contains v.

We need to show:

$$u \text{ can reach } v \text{ in } G \iff S(u) \text{ can reach } S(v) \text{ in } \text{scc}(G).$$

This is a statement about how reachability between vertices in the original graph matches reachability between SCCs in the condensed graph.

Direction 1: u reaches $v \Rightarrow S(u)$ reaches $S(v)$

Assume there is a path:

$$u = x_0 \to x_1 \to \cdots \to x_k = v.$$

Each vertex x_i belongs to some SCC $S(x_i)$.

Now look at the sequence of SCCs along the path:

$$S(x_0), S(x_1), \ldots, S(x_k).$$

Two simple observations:

- When we move by an edge inside the same SCC, the SCC label does not change.
- When we move from $S(x_i)$ to a different SCC $S(x_{i+1})$, this means there is an edge between these two SCCs in $\text{scc}(G)$.

If we compress repeated SCCs (like $C, C, C \to C$) into one, we get a path from $S(u)$ to $S(v)$ in $\text{scc}(G)$.

So $S(u)$ can reach $S(v)$.

Direction 2: $S(u)$ reaches $S(v)$ $\Rightarrow$ u reaches v

Now assume there is a path between SCCs:

$$S(u) = C_0 \to C_1 \to \cdots \to C_m = S(v).$$

By definition of $\text{scc}(G)$:

- For each i, there is at least one edge (a_i, b_i) where $a_i \in C_i$ and $b_i \in C_{i+1}$.
- Inside each SCC C_i, every vertex can reach every other vertex.

We can build a path from u to v in G like this:

1) Start at u in SCC C_0.
2) Move inside C_0 to vertex a_0.
3) Use edge (a_0, b_0) to go to SCC C_1.
4) Move inside C_1 to vertex a_1, then go along edge (a_1, b_1) to C_2.
5) Continue this process until we reach SCC $C_m = S(v)$.
6) Inside C_m, move from b_{m-1} to v.

This gives a real path from u to v in the original graph.
So u can reach v in G.

Conclusion

We have shown both directions, so the statement is true:

$$u \text{ can reach } v \text{ in } G \iff S(u) \text{ can reach } S(v) \text{ in } \text{scc}(G).$$

3 Problem 2: Euler Tour

We have a strongly connected directed graph $G = (V, E)$.

3.1 2.1 When Does an Euler Tour Exist?

Definition

An **Euler tour** is a cycle that:

- uses every edge exactly once,
- can visit vertices multiple times.

We want to show:

$$G \text{ has an Euler tour} \iff \text{in-degree}(v) = \text{out-degree}(v) \text{ for every vertex } v.$$

If there is an Euler tour, then in-degree = out-degree

Suppose G has an Euler tour.
Think about what happens at a vertex v when we follow the tour:

- Every time we enter v along some edge, we must also leave v along another edge, except possibly at the very beginning and very end.
- But an Euler tour is a **cycle**, so it starts and ends at the same vertex.
- So for each vertex, the number of times we enter it equals the number of times we leave it.

Because the tour uses every edge exactly once, the number of incoming edges for v is equal to the number of outgoing edges:

$$\text{in-degree}(v) = \text{out-degree}(v).$$

If in-degree = out-degree (and G is strongly connected), then there is an Euler tour

Now assume:

- G is strongly connected,
- for every vertex v we have in-degree(v) = out-degree(v).

The idea is:

1) Start from any vertex s.
2) Always follow an unused outgoing edge from your current vertex.
3) Because out-degrees and in-degrees are balanced, you cannot "get stuck" in the middle of the graph with some edges entering the vertex but no edges leaving it.
4) Eventually, you must return to s and form a cycle (this is a closed walk).

If this cycle already uses all edges, then it is an Euler tour.
If not, there is at least one unused edge somewhere. By strong connectivity, this unused edge is reachable from some vertex on our cycle. We can:

- start from that vertex on the cycle,
- follow a new path using unused edges,
- eventually come back to the same vertex and form another cycle,
- then **merge** this new cycle into the old cycle.

Repeating this, we get one big cycle that uses every edge exactly once.
So an Euler tour exists.

3.2 2.2 $O(E)$-Time Algorithm to Find an Euler Tour

We can turn the previous idea into an algorithm, known as **Hierholzer's algorithm**.

Algorithm (informal)

1) Pick any start vertex s.

2) From s, walk along unused edges until you come back to s. This gives a cycle C.

3) While there is some vertex v on C that still has an unused outgoing edge:

    a) start a new walk from v using only unused edges,

    b) this walk will also return to v and form a new cycle C',

    c) merge C' into C at vertex v.

4) When there are no unused edges left, C is an Euler tour.

Why is the running time $O(E)$?

- Every edge is used exactly once when it is added to some cycle.
- We never revisit an edge as "unused".
- We can store a pointer for each vertex to the next unused outgoing edge in its adjacency list.

So each edge is processed a constant number of times, and the total running time is $O(E)$.

4 Problem 3: Topological Sort

We are given the following directed edges:

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

We need to:

- perform a topological sort starting from A,
- then start from another node and find one or two other valid orders.

4.1 What is a Topological Order?

A **topological order** is a list of all vertices such that:

for every edge (u, v), vertex u appears before v in the list.

This is possible only if the graph has no directed cycles (i.e., it is a DAG).

4.2 Compute In-Degrees

Vertices: A, B, C, D, E, F, G.

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

- A: no edges into $A \Rightarrow$ in-degree(A) = 0.
- B: edge from $A \Rightarrow$ in-degree(B) = 1.
- C: edges from A and $B \Rightarrow$ in-degree(C) = 2.
- D: edge from $B \Rightarrow$ in-degree(D) = 1.
- E: edges from C, D, $G \Rightarrow$ in-degree(E) = 3.
- F: edges from D, $G \Rightarrow$ in-degree(F) = 2.
- G: no edges into $G \Rightarrow$ in-degree(G) = 0.

So the initial vertices with in-degree 0 are A and G.

4.3 Algorithm: Kahn's Method (High-Level)

1) Compute in-degree for all vertices.
2) Put all vertices with in-degree 0 in a set (or queue).
3) While this set is not empty:
    a) pick any vertex u from the set,
    b) append u to the topological order,
    c) remove all edges going out of u,
    d) for each neighbor v of u, decrease in-degree(v) by 1,
    e) if some in-degree(v) becomes 0, add v to the set.

Different choices of vertices in step 3a can lead to different valid topological orders.

4.4 Topological Sort Starting from A

We will always choose A first when it is possible.

Step 1

Vertices with in-degree 0: $\{A, G\}$.

Choose A first.

Order so far: A.

Remove edges $A \to B$ and $A \to C$:

- in-degree(B): $1 \to 0$.
- in-degree(C): $2 \to 1$.

Now in-degree 0 vertices: $\{G, B\}$.

Step 2

From $\{G, B\}$ choose B (to stay near A in the order).
Order: A, B.
Remove edges $B \to C$ and $B \to D$:

- in-degree(C): $1 \to 0$.
- in-degree(D): $1 \to 0$.

Now in-degree 0 vertices: $\{G, C, D\}$.

Step 3

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

- in-degree(E): $3 \to 2$.
- in-degree(F): $2 \to 1$.

Now in-degree 0 vertices: $\{G, C\}$.

Step 4

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

- in-degree(E): $2 \to 1$.

Now in-degree 0 vertices: $\{G\}$.

Step 5

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

- in-degree(F): $1 \to 0$.
- in-degree(E): $1 \to 0$.

Now in-degree 0 vertices: $\{F, E\}$.

Step 6

Choose F.
Order: A, B, D, C, G, F.
No outgoing edges from F.
Now in-degree 0 vertices: $\{E\}$.

Step 7

Choose E.

Final order:

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

Check

Check that every edge goes from left to right in this list.

- $A \to B$: A before B.
- $A \to C$: A before C.
- $B \to C$: B before C.
- $B \to D$: B before D.
- $C \to E$: C before E.
- $D \to E$: D before E.
- $D \to F$: D before F.
- $G \to F$: G before F.
- $G \to E$: G before E.

So this is a valid topological order that "starts from A".

4.5 Topological Sort Starting from Another Node (e.g., G)

Now we choose G first when possible.

Step 1

Initial in-degree 0 vertices: $\{A, G\}$.

Choose G.

Order: G.

Remove edges $G \to F$ and $G \to E$:

- in-degree(F): $2 \to 1$.
- in-degree(E): $3 \to 2$.

Now in-degree 0 vertices: $\{A\}$.

Step 2

Choose A.

Order: G, A.

Remove edges $A \to B$ and $A \to C$:

- in-degree(B): $1 \to 0$.
- in-degree(C): $2 \to 1$.

Now in-degree 0 vertices: $\{B\}$.

Step 3

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

- in-degree(C): $1 \to 0$.
- in-degree(D): $1 \to 0$.

Now in-degree 0 vertices: $\{C, D\}$.

Step 4

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

- in-degree(E): $2 \to 1$.
- in-degree(F): $1 \to 0$.

Now in-degree 0 vertices: $\{C, F\}$.

Step 5

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

- in-degree(E): $1 \to 0$.

Now in-degree 0 vertices: $\{F, E\}$.

Step 6

Choose F.
Order: G, A, B, D, C, F.

Step 7

Choose E.
Final order:

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

This is another valid topological order.

4.6 Many Valid Orders

We can get other valid orders by changing choices when several vertices have in-degree 0. For example, we could choose C before D at some step, and so on.

The important point is:

- All edges must go from left to right.
- Any order that satisfies this is a correct topological sort.

5 Conclusion

In this report, we studied three topics about directed graphs:

- how to reverse a graph and how SCCs behave under reversal;
- when a directed graph has an Euler tour and how to find one in $O(E)$ time;
- how to perform topological sort and why different orders can be valid.

The explanations were intentionally written in very simple language, with many small steps, so that a beginner in graph algorithms can follow the logic and understand not only *what* to do, but also *why* each statement is true.