SCC, Topological Sort, and Dijkstra

Data Structures and Algorithms Tutorial

IIITH - S26

Kosaraju, Topological Sort, Dijkstra (C++)

Agenda

1. Strongly Connected Components
2. Practice Problems - SCC
3. Topological Sort
4. Practice Problems - Topological Sort
5. Dijkstra's Algorithm
6. Practice Problems - Dijkstra

Strongly Connected Components

What is a Strongly Connected Component?

Definition

- A Strongly Connected Component (SCC) is a maximal subgraph where every node can reach every other node
- Maximal: Can't add more nodes while keeping the property

In undirected graphs: Just "connected components" (DFS/BFS enough)

- Since edges work both ways, reachability is symmetric

In directed graphs: Need special algorithms!

- u can reach v doesn't mean v can reach u

Why do we care?

- Condensation graph (DAG of SCCs) is super useful
- Cycle detection, finding "groups" in social networks, etc.

SCC Intuition

What are the SCCs?

- Nodes that can reach each other form one SCC
- Any cycle is part of an SCC
- Nodes feeding into a cycle are NOT in the same SCC (unless they can also be reached from the cycle)

Key Property

- If you compress each SCC to a single node, you get a DAG
- No cycles in the condensation graph (by definition!)

Example

- Graph with cycle {1, 2, 3} and node 4 pointing to 1
- {1, 2, 3} is one SCC, {4} is another SCC
- Condensation: 4 $\longrightarrow$ {1,2,3} (a DAG)

Kosaraju's Algorithm

Two-Pass Algorithm

Pass 1: Order discovery

1. Do DFS on original graph
2. Push node to stack when BACKTRACKING (after visiting all neighbors)
3. Stack now has nodes in order of decreasing finish time

Pass 2: Find SCCs

1. Reverse all edges in graph
2. Pop nodes from stack one by one
3. Each DFS from unvisited node in reversed graph = one SCC!

Time: $O(V + E)$ for each pass = $O(V + E)$ total Space: $O(V + E)$ for reversed graph

Kosaraju: Implementation

Stack Order: Push when returning from DFS (postorder)

```
void dfs(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) dfs(v);
    }
    order.push_back(u);  // Push AFTER all neighbors
}
```

Second Pass: Process in reverse order

```
void dfs2(int u) {
    visited[u] = true;
    comp.push_back(u);
    for (int v : radj[u]) {  // Reversed graph
        if (!visited[v]) dfs2(v);
    }
}
```

Each DFS2 call gives one SCC!

Practice Problems - SCC

Problem : Strongly Connected Cities

Problem Statement Given a directed graph with n cities and m one-way roads, find the number of strongly connected components. (check code/kosaraju.cpp)

Intuition

- Direct application of Kosaraju
- Count how many times we start a new DFS in second pass

Solution

1. First DFS: Get finish order
2. Build reversed graph
3. Second DFS: Count SCCs

Time: $O(n + m)$ | Space: $O(n + m)$

Topological Sort

What is Topological Sort?

Definition

- A linear ordering of vertices such that for every directed edge (u, v), u comes before v in the ordering
- ONLY exists for DAGs (Directed Acyclic Graphs)

Why do we care?

- Task scheduling: some tasks must finish before others start
- Build dependencies: some files must be compiled before others
- Prerequisite chains in courses
- DP on DAGs (process in topological order)

Topological Sort: Kahn's Algorithm

Main Idea: Repeatedly remove nodes with no incoming edges

Algorithm

1. Compute in-degree for all nodes
2. Add all nodes with in-degree 0 to queue
3. While queue not empty:
   - Remove node u from queue
   - Add u to result
   - For each neighbor v: decrement in-degree, add to queue if 0

Time: $O(V + E)$ | Space: $O(V)$

Cycle Detection: If result size < V, graph has a cycle!

Topological Sort: DFS-Based Approach

Alternative: Use DFS with finish times

Algorithm

1. Do DFS on unvisited nodes
2. Push node to stack when BACKTRACKING (after all neighbors)
3. Reverse the stack to get topological order

Why reverse?

- Node with no outgoing edges finishes first (goes to bottom of stack)
- Source nodes finish last (go to top of stack)
- Reversing gives correct order!

Same intuition as Kosaraju's first pass

Topological Sort: Implementation

```
void topological_sort(int n) {
    queue<int> q;
    vector<int> result;
    for (int i = 1; i <= n; i++) {
        if (indeg[i] == 0) q.push(i);
    }

    while (!q.empty()) {
        int u = q.front(); q.pop();
        result.push_back(u);
        for (int v : adj[u]) {
            if (--indeg[v] == 0) q.push(v);
        }
    }
}
```

If result.size() < n: cycle exists!

Practice Problems - Topological Sort

Problem 1: Course Schedule

Problem Statement Given n courses and prerequisites, find an order to take all courses. If impossible, output "IMPOSSIBLE".

Link: https://cses.fi/problemset/task/1679

Intuition

- Direct topological sort application
- If cycle exists, impossible to complete all courses

Solution

1. Build graph from prerequisites
2. Run Kahn's algorithm
3. Check if all courses were processed

Time: $O(n + m)$ | Space: $O(n + m)$

Problem 2: Longest Flight Route

Problem Statement Given n cities and m flights forming a DAG, find the longest route from city 1 to city n.

Link: https://cses.fi/problemset/task/1680

Intuition

- DAG allows DP with topological sort
- Process nodes in topological order, update longest distance

Solution

1. Topological sort the DAG
2. dp[v] = max(dp[v], dp[u] + 1) for each edge $u \to v$
3. Reconstruct path using parent array

Time: $O(n + m)$ | Space: $O(n + m)$

Dijkstra's Algorithm

Dijkstra's Algorithm: Main Idea

Assumption: All edge weights are NON-NEGATIVE

Greedy Strategy

- Always process the unvisited node with smallest known distance
- Once we visit a node, its distance is FINAL (can't be improved later)
- This is why non-negative weights are crucial!
- Give analogy on proof about for each point the shortest path IS the shortest distance

Time: Depends on how we find minimum distance node

Dijkstra: Algorithm Steps

1. Initialize: dist[source] = 0, all others = infinity
2. Use priority queue (min-heap) of (distance, node) pairs
3. While PQ not empty:
   - Pop node u with minimum distance
   - If u already visited: skip
   - Mark u as visited
   - For each neighbor v with weight w:
     - If dist[u] + w < dist[v]:
       - Update dist[v] = dist[u] + w
       - Push (dist[v], v) to PQ

Note: Each node can be pushed multiple times, but only processed once (when popped with minimum distance)

Dijkstra Visualization

Dijkstra: Why Does It Work?

Invariant: When we pop node u from PQ, dist[u] is final

Proof:

- Suppose there exists a shorter path to u through unvisited nodes
- This path goes: $s \to \ldots \to x \to u$, where x is unvisited
- Since all edges ≥ 0: dist[s $\to$ x] < dist[s $\to$ u]
- But we popped u first, so dist[u] $\leq$ dist[x] (PQ property)
- Contradiction!

Negative weights break this: dist[s $\to$ x] could be < dist[s $\to$ u] even if popped first, due to negative edge later

Dijkstra: Complexity

Naive implementation: Scan all nodes for minimum

- Finding minimum: $O(V)$
- Total: $O(V^2 + E) = O(V^2)$

With priority queue:

- Each node pushed: $O(E)$ times (once per edge relaxation)
- Each pop: $O(\log V)$
- Total: $O((V + E) \log V)$

For sparse graphs ($E \approx V$): $O(V \log V)$ For dense graphs ($E \approx V^2$): $O(V^2 \log V)$

Dijkstra: Implementation

Don't mark visited when pushing to PQ

- Mark only when popping (might have stale entries)
- Check `if (visited[u]) continue;` at start of loop

Long long for distances

- Weights can sum up large: use `long long`

INF should be large enough

- `const long long INF = 1e18;` (for $n \leq 10^5$, weights $\leq 10^9$)

Reconstruction

- Store `parent[v] = u` when relaxing edge
- Walk backwards from destination to get path

Practice Problems - Dijkstra

Problem 1: Shortest Routes I

Problem Statement Given weighted graph with non-negative edges, find shortest distance from node 1 to all other nodes.

Link: https://cses.fi/problemset/task/1671

Solution

1. Build adjacency list
2. Dijkstra from source (node 1)
3. Output distances

Time: $O((n+m)\log n)$ | Space: $O(n+m)$

Problem 2: Flight Discount

Problem Statement You can use a discount coupon for ONE flight (50% off). Find minimum cost to reach city n from city 1. Link: https://cses.fi/problemset/task/1195

Intuition

- At each node, track TWO distances:
  - dist[v]: minimum cost to reach v WITHOUT using coupon
  - dist2[v]: minimum cost to reach v AFTER using coupon
- When using edge (u, v, w):
  - If haven't used coupon: can use it OR not use it
  - If already used: just pay full price

Solution Dijkstra with state (node, coupon_used) as key

Time: $O((n + m) \log n)$ | Space: $O(n + m)$

Homework Problems !

Planet Queries II

Flight Routes

Course Schedule II

Questions?

Thank you!