Home

DATA STRUCTURES

Algorithms

WEEK 08: ELEMENTARY GRAPHS (LAB)

Bingjie Xu

2026-02-26

Home

INTRO

Home

LEARNING OBJECTIVES

1. Implement graph traversal algorithms (BFS, DFS) in C++.
2. Determine reachability between nodes in a graph.
3. Perform topological sorting on directed acyclic graphs (DAGs).
4. Identify connected components using BFS/DFS in C++.

Home

OVERVIEW OF LAB ACTIVITIES

This lab consists of three main activities:

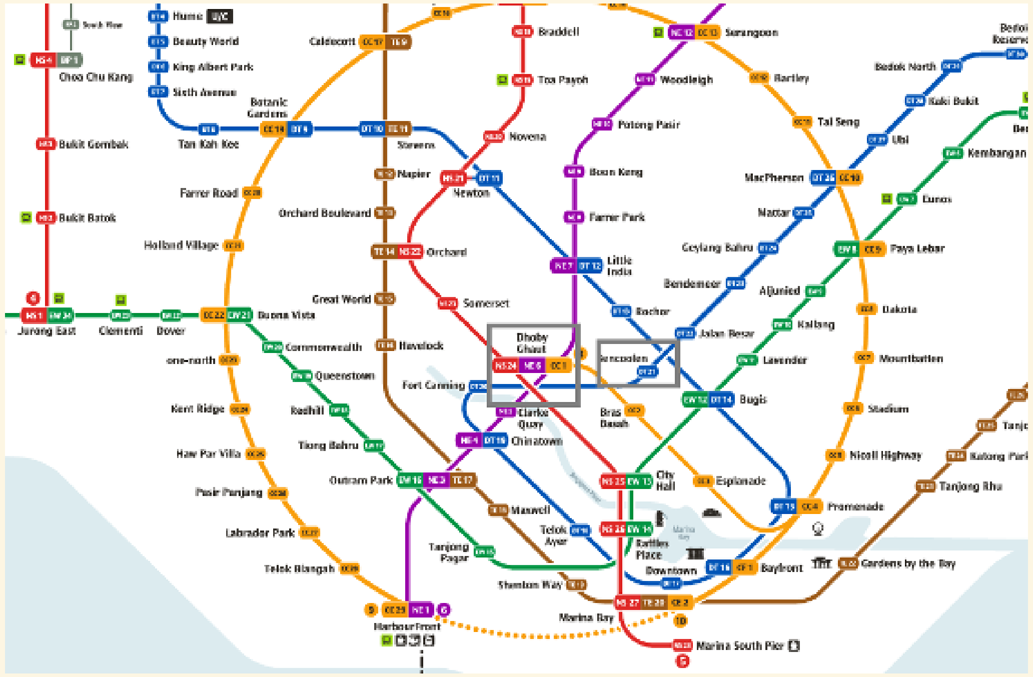
1. **Reachability in an Unweighted Graph** (20 marks)
    - Implement a function to determine if a path exists between two vertices in a directed graph using BFS or DFS.
    - Submit your C++ implementation via Gradescope.
2. **Topological Sorting of a DAG** (10 marks)
    - Model the process of getting dressed in the morning as a directed acyclic graph (DAG).
    - Draw the DAG and compute a valid topological ordering using DFS.
    - Submit your DFS and topological order via xSITe Dropbox.
3. **Connected Components** (30 marks)
    - Implement an algorithm to identify all connected components in an undirected graph using BFS or DFS.
    - Submit your C++ implementation via Gradescope.

Home

REACHABILITY IN AN UNWEIGHTED GRAPH

APPLICATION

A real-world application of reachability queries is determining if you can travel from one station to another in a transportation network. For example, in the Singapore (MRT) system, you might want to know if there's a path from Dhoby Ghaut to Bencoolen.

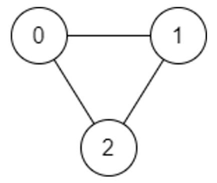


In this graph representation: **Vertices** represent MRT stations. **Edges** represent train lines connecting stations. A **reachability query** answers: "Can I get from station s to station t"?

Home

EXERCISE

You are given an **unweighted undirected graph** $G = (V, E)$ in form of edges. Your task is to decide whether there is a path from a source vertex s to a target vertex t.



Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true

1. Download **valid_path.zip** from xSITe. valid_path.cpp contains a stub for function `bool validPath(int n, vector<vector<int>> &edges, int source, int destination);`
2. Build an adjacency list representation of the graph from the edge list.
3. Implement `reachable` using **BFS** or **DFS** so that it runs in $O(|V| + |E|)$.
4. Make and Test your implementation locally using the provided test.txt.

Submission (Gradescope, 20 marks). Upload your completed `valid_path.cpp` on Gradescope.

Home

CONSTRAINTS

- The number of vertices $|V|$ is between 1 and $2 * 10^5$ (inclusive).
- The number of edges $|E|$ is between 0 and $2 * 10^5$ (inclusive).
- Vertices are labeled from 0 to $|V| - 1$.
- There are no duplicate edges.

ACTIVITY 1: KEY POINTS

- Build an adjacency list representation of the graph from the edge list.
- Use **BFS** (with a queue) or **DFS** (with a stack or recursion) to explore all reachable vertices from the source.
- Mark vertices as visited to avoid revisiting them.
- Return `true` immediately when the destination is found.
- Return `false` if all reachable vertices have been explored (queue/stack is exhausted or recursion completes) without finding the destination.

TOPOLOGICAL SORT

Home

USE CASES OF TOPOLOGICAL SORT

Topological sorting is useful in many real-world applications:

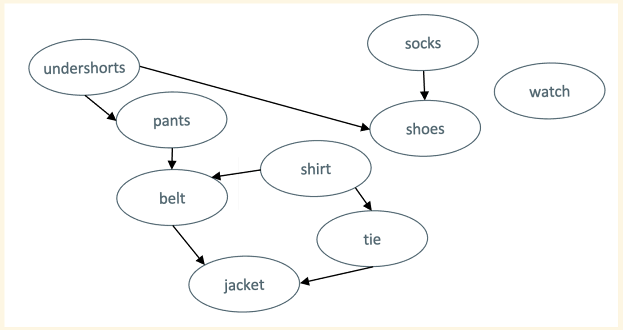
- **Task Scheduling**: Determining the order of tasks with dependencies (e.g., building software, course prerequisites)
- **Event Ordering**: Finding a valid sequence of events that respects precedence constraints
- **Dependency Resolution**: Resolving dependencies in package managers (e.g., npm, pip)
- **Course Prerequisites**: Determining a valid course enrollment sequence
- **Build Systems**: Compiling source files in the correct order based on dependencies

Home

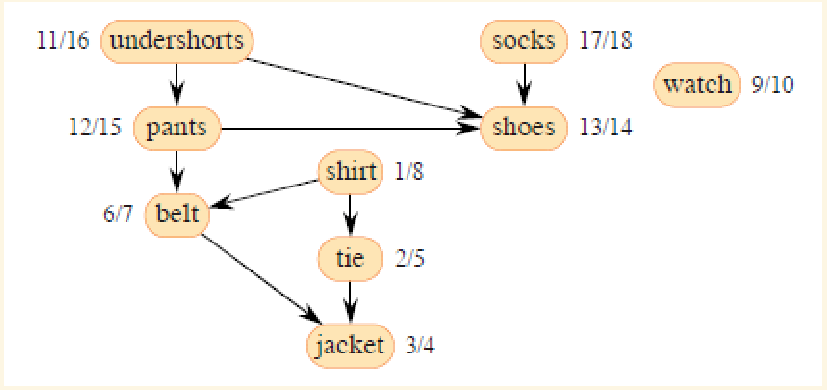
DIRECTED ACYCLIC GRAPH (DAG)

A **Directed Acyclic Graph (DAG)** is a directed graph with no cycles. This property is essential for topological sorting - if a graph contains cycles, no valid topological ordering exists.

Example of a Directed Acyclic Graph (DAG):



One of many valid DFS trees on the DAG, assuming `shirt` is the source vertex:



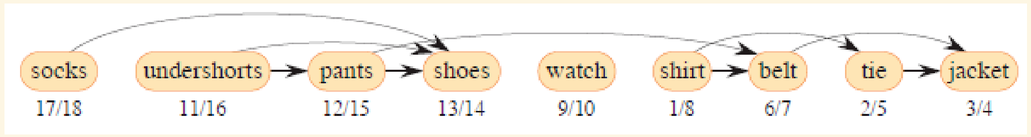
Home

TOPOLOGICAL SORT

Topological sort (or **topological ordering**) is a linear ordering of the vertices of a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.

Topological sort can be performed using **DFS**. The key insight is to process nodes in **reverse order of finishing times** during DFS.

The resulted topological sorting:



The valid topological order **might not be unique** because DFS can produce different orderings depending on the starting node and the order of neighbor selection. However, any topological order produced will always remain valid (a node will appear before its dependent nodes).

Home

TOPOLOGICAL SORT PSEUDO CODE

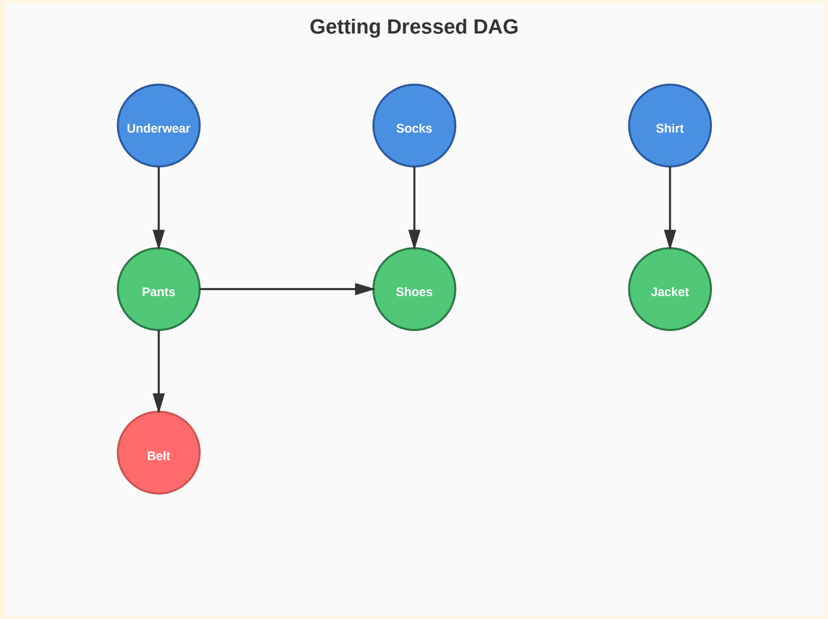
```
TOPOLOGICAL-SORT(G):
    1. Initialize an empty list L for the topological order
    2. For each vertex v in G:
        3. If v is not visited:
            4. DFS-VISIT(G, v, L)
    5. Return L (reversed)

DFS-VISIT(G, v, L):
    1. Mark v as visited
    2. For each neighbor u of v:
        3. If u is not visited:
            4. DFS-VISIT(G, u, L)
    5. Add v to the front of L (or push to stack)
```

Home

EXERCISE: GETTING DRESSED DAG

Model “getting dressed in the morning” as a **DAG**.



1. Choose source vertex `Underwear`.
2. Use DFS-based traversal approach.
3. Compute a valid **topological sorting**.

Submission (xSITe Dropbox, 10 marks). Upload your DFS spanning tree with visiting and finishing times, and the corresponding topological order.

Home

ACTIVITY 2: KEY POINTS

- In a valid topological order all edges go from left to right.
- If your algorithm detects a cycle, your graph is not a DAG and must be fixed.
- DFS-based topological sort processes nodes in reverse finishing time order.

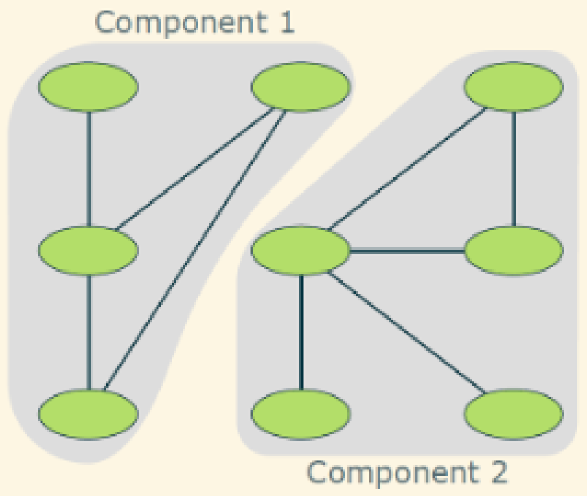
Home

CONNECTED COMPONENTS IN AN UNDIRECTED GRAPH

Home

CONNECTED COMPONENTS

A **connected component** of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices in the set. In other words, all vertices in a connected component are reachable from each other.



Connected Components Example: The graph above has 2 connected components.

Home

USE CASES OF CONNECTED COMPONENTS

Connected components have numerous real-world applications:

- **Network Analysis**: Identifying isolated subnetworks or clusters in computer networks, social networks, or communication systems
- **Image Processing**: Finding connected regions in binary images (e.g., identifying objects, characters in OCR)
- **Geographic Information Systems (GIS)**: Finding connected landmasses, road networks, or water bodies
- **Puzzle Solving**: Determining if all pieces of a puzzle are connected or if there are separate groups
- **Game Development**: Detecting if game objects or regions are connected (e.g., pathfinding, territory control)

Home

EXERCISE

You are given an **undirected graph** with n vertices and a list of edges. Your task is to count the number of connected components in the graph.

1. Download **connected_components.zip** from xSITe. The file `connected_component.cpp` contains a stub for function `int countComponents(int n, vector<vector<int>> &edges);`
2. Implement `countComponents` to return the number of connected components in the graph.
3. Use BFS or DFS to explore each connected component.
4. Make and Test your implementation locally using the provided test.txt.

Submission (Gradescope, 30 marks). Upload your completed `connected_component.cpp` file on Gradescope.

Home

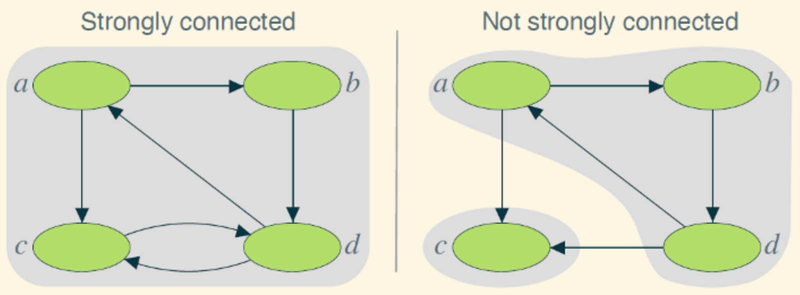
ACTIVITY 3: KEY POINTS

- Build adjacency-list representation of graph from the edge list.
- Maintain a `visited` array and `current_component_id`.
- For each unvisited vertex, start BFS/DFS, label all reachable vertices, then increment `current_component_id`.

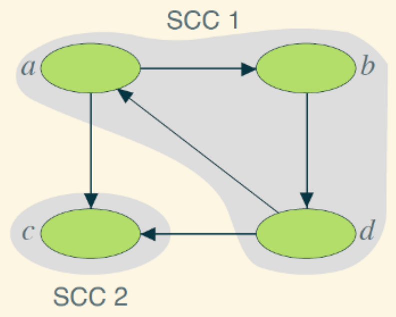
Home

STRONGLY CONNECTED COMPONENTS (SCC)

For **directed graphs**, we use the concept of **Strongly Connected Components (SCC)**. A strongly connected component is a maximal set of vertices such that for every pair of vertices u and v in the set, there is a directed path from u to v and from v to u.



Strongly Connected Component Example: A single SCC in a directed graph.



Multiple SCCs Example: A directed graph with multiple strongly connected components.

Home

CONCLUSION

WRAP-UP

By the end of this lab you should be able to:

1. Implement **reachability** queries in directed graphs.
2. Model constraints as DAGs and compute a **topological order**.
3. Compute **connected components** in undirected graphs using BFS/DFS.

Home

OUTLOOK

Elementary Graph Algorithms
: Representations, breadth-first and depth-first search, topological sorting, connected components

Disjoint Sets and Minimum Spanning Trees
: Union-find data structures, Kruskal’s and Prim’s minimum-spanning-tree algorithms

Single-Source Shortest Paths
: Dijkstra’s and Bellman-Ford algorithms

Dynamic Programming and Greedy Algorithms
: Versatile optimization techniques