

Fundamental Algorithmic Techniques

VIII

November 24, 2025

Outline

Topological Sort

Cycles Detection

Connected Components

Minimum Spanning Trees

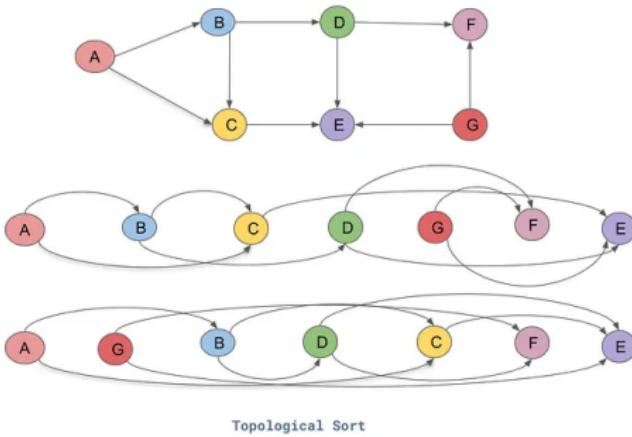
Graph Colouring Algorithms

Shortest Paths

Flow Networks

Topological Sort: DFS Approach

- Works only on **Directed Acyclic Graphs** (DAGs)
- Detects cycles (if any node is visited twice)
- Result is **not unique** in general
- All trees have a topological order
- Course prerequisites, Build/makefile dependencies, Class loading in Java



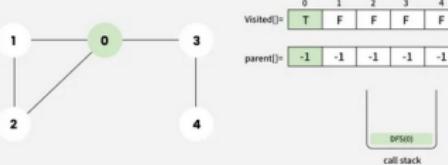
Algorithm:

Create empty stack S_{topo}

- 1 pick random starting node
- 2 run DPS until node has no children (cancel edges along the way)
- 3 push childrenless node into S_{topo}
- 4 perform DPS further if possible, if not start from new node.

Cycle Detection: DFS vs BFS — Complexity

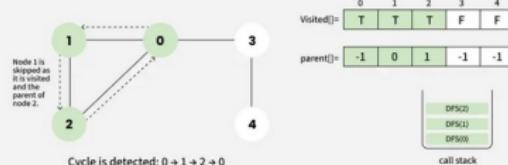
02 Start DFS from node 0: mark it as visited, and add DFS(0) to the call stack.



Detect Cycle in Undirected Graph

Depth-First Search step 0

05 Explore the neighbors of node 2: encounter neighbor 0, which is already visited and not the parent of 2 (as Parent[2] = 1)



Detect Cycle in Undirected Graph

Depth-First Search step 3

Both detect the cycle when exploring the back edge (e.g., $D \rightarrow A$):

since the target node is already visited and not the immediate parent (in undirected) or is on the recursion stack (in directed).

Complexity:

- Time: $O(V + E)$ for both
Every vertex and edge is processed at most once.
- Space: $O(V)$ for both
 - **DFS:** Call stack depth V (worst-case path).
 - **BFS:** Queue may hold up to $O(V)$ nodes (e.g., wide level).

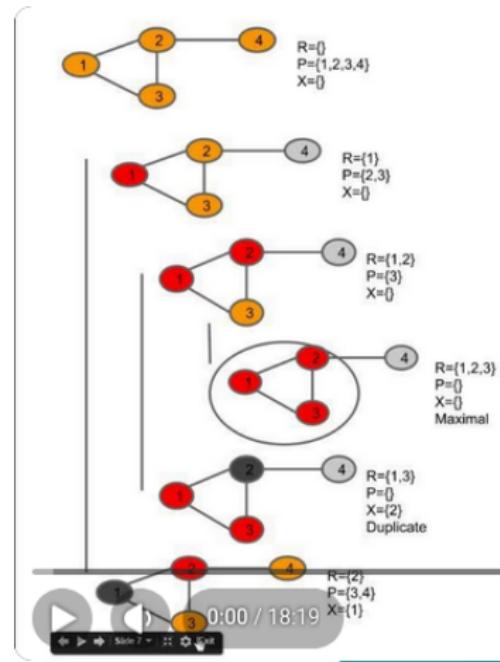
Bron–Kerbosch Algorithm: Maximal Clique Enumeration

Undirected graph $G = (V, E)$,
 $N(v) = \text{neighbors of } v \text{ in } G$,

Initial call: $\text{BronKerbosch1}(\emptyset, V, \emptyset)$

Pseudocode:

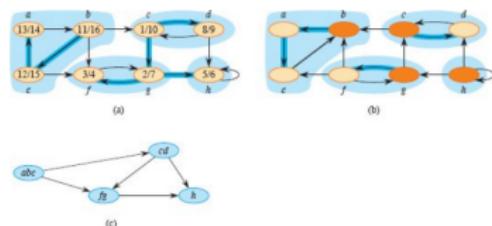
```
algorithm BronKerbosch1(R, P, X) is
    if P and X are both empty then
        report R as a maximal
        clique
    for each vertex v in P do
        BronKerbosch1(R ∪ {v}, P ∩
        N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```



Kosaraju's Algorithm - Strongly Connected Components

Kosaraju's Algorithm

- 1 **DFS on Original Graph:** Record finish times
- 2 **Transpose the Graph:** Reverse all edges
- 3 **DFS on Transposed Graph:** Process nodes in order of decreasing finish times to find SCCs

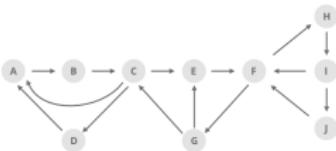


Two-pass DFS to find SCCs

Time Complexity: Depth First Search: $O(V + E)$

Space Complexity: Stack: $O(V)$

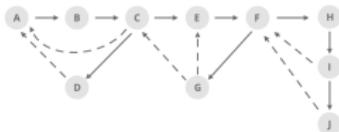
Tarjan's Algorithm for SCCs



Initially :

A	B	C	D	E	F	G	H	I	J
Disc	NIL								
Low	NIL								

Dfs Traversal :



A	B	C	D	E	F	G	H	I	J
Disc	1	2	3	4	5	6	7	8	9
Low	1	1	1	1	3	3	3	6	6

86

Index: Discovery order in DFS

Low-link: Smallest index
reachable via DFS (including
back edges)

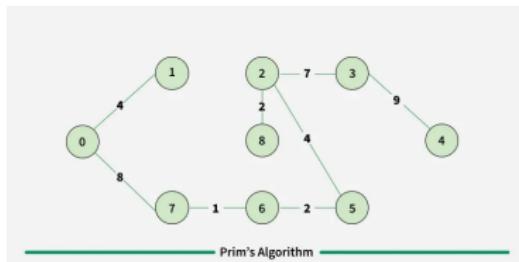
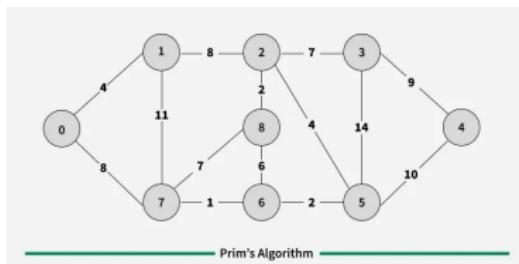
Problem: Random DFS order
→ ambiguous SCC boundaries

Solution: Use a stack to track
active nodes

- When $\text{low}[u] = \text{index}[u]$:
pop stack until u — those form
one SCC

Complexity: $O(V + E)$
node/edge visited once

Jarník's (Prim's) Algorithm



Prim's Algorithm: initial graph (top)
and MST

Steps

- 1 Start from arbitrary vertex for MST.
- 2 Till there are fringe vertices:
- 3 Find edges connecting tree & fringe vertices
- 4 Find the minimum among these edges
- 5 Add the chosen edge to the MST
- 6 Return the MST

Complexity

- Time: $\mathcal{O}(E \cdot \log V)$ with binary heap
- Space: $\mathcal{O}(V)$

Cuts and the Cut Property

Definition (Cut)

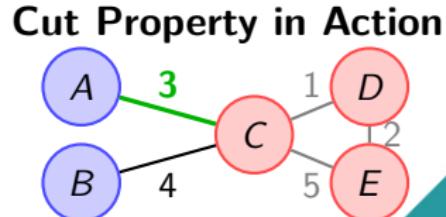
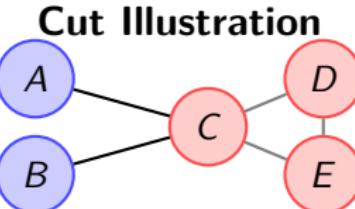
A *cut* $(S, V \setminus S)$ of an undirected graph $G = (V, E)$ is a partition of V into two non-empty disjoint subsets S and $V \setminus S$.

Cut-Crossing Edge

An edge $(u, v) \in E$ crosses the cut if exactly one of u or v is in S .

Cut Property / Theorem

For any cut $(S, V \setminus S)$, if edge e 's weight is the minimum among all edges crossing the cut, then e belongs to *some MST of G* .



Note: The green edge (weight 3) is the lightest crossing edge and must appear in *some MST*.

MST Building: Proof of Correctness

Key Invariant

At each step, the tree T maintained by the algorithm is a subset of some MST.

Proof by Induction

- **Base Case:** $T = \{v_0\}$ is trivially part of an MST.
- **Inductive Step:** Assume T is part of MST T^* . Let $e = (u, v)$ be the minimum-weight edge crossing the cut $(T, V \setminus T)$.
 - If $e \in T^*$: $T \cup \{e\}$ still part of T^* .
 - If $e \notin T^*$: $\exists e' \in T^*$ crossing same cut. Then $T^* - \{e'\} \cup \{e\}$ is also an MST (by cut property), since $w(e) \leq w(e')$.

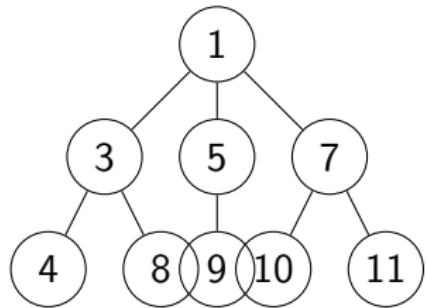
Conclusion

By induction, T is always part of an MST.

Final T is a spanning tree \Rightarrow it is an MST.

Jarník's Algorithm: Binary Heap Implementation

```
1: procedure JARNIKMST( $G = (V, E)$ )
2:    $Q \leftarrow$  empty min-heap                      ▷ Vertices with key values
3:   for  $v \in V$  do
4:      $key[v] \leftarrow \infty$ 
5:      $Q.insert(v, key[v])$ 
6:   end for
7:    $key[0] \leftarrow 0$                                 ▷ Start from vertex 0
8:   while  $Q$  is not empty do
9:      $u \leftarrow Q.extractMin()$ 
10:    for  $v \in Adj[u]$  and  $v \in Q$  do
11:      if  $w(u, v) < key[v]$  then
12:         $parent[v] \leftarrow u$ 
13:         $key[v] \leftarrow w(u, v)$ 
14:      end if
15:    end for
16:  end while
17: end procedure
```



Min-heap example
with 3 children for
root

Complexity: $O(E \log V)$ if using min heap
(Faster alternative with Fibonacci)

Kruskal's Algorithm - Greedy MST Construction

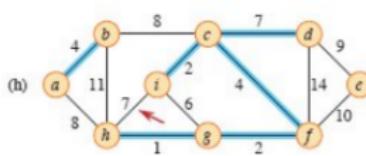
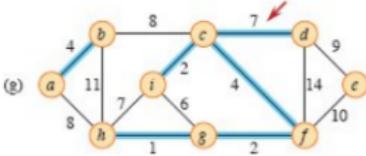
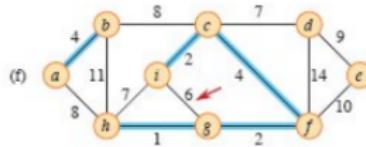
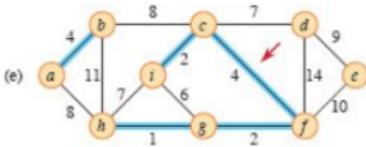
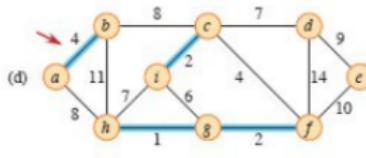
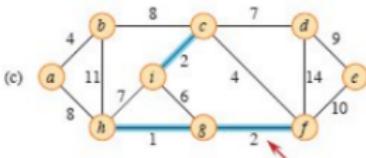
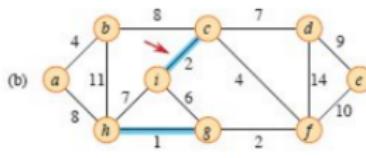
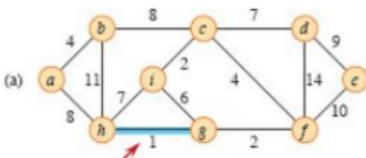
Kruskal's Algorithm Steps

- 1 **Initialize DSU:** Each vertex in its own component
- 2 **Sort edges:** By weight (ascending order)
- 3 **For each edge** (u, v) in sorted order:
- 4 **Check for cycle:** If $\text{find}(u) \neq \text{find}(v)$
- 5 **Add to MST:** Include edge if no cycle
- 6 **Union:** Merge components using $\text{union}(u, v)$
- 7 **Skip:** If same component (cycle detected)

Greedy Strategy

Always pick the smallest available edge that doesn't create a cycle

Kruskal Algorithm: Execution



Stepwise execution of Kruskal Algorithm

Graph Coloring – Map and Schedule Applications

Problem: Assign as **few colors as possible** to vertices so that no two adjacent vertices share the same color.

Example:

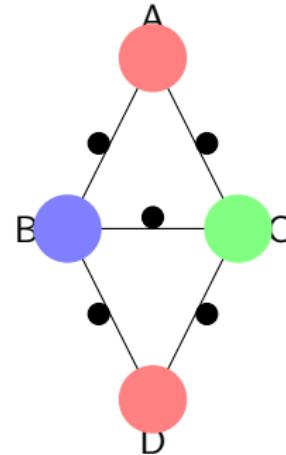
Vertices: Regions on a map or tasks needing resources

Edges: Conflicts

Chromatic Number: minimum colors needed: $\chi(G) = 3$ (NP Hard)

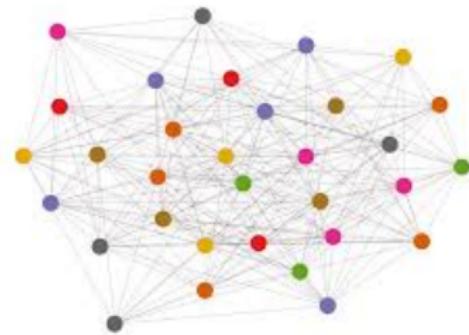
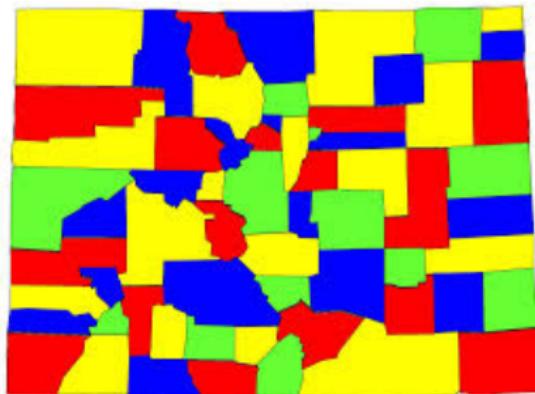
Real-world use cases:

- Scheduling exams
- Register allocation in compilers
- Frequency assignment in wireless networks



A 3-coloring: $A, D = \text{red}$; $B = \text{blue}$; $C = \text{green}$

Nice examples of graph colouring problems



Bipartite Graphs & Graph Coloring

Bipartite Graph

A graph whose vertices can be divided into two disjoint sets U and V , such that every edge connects a vertex in U to one in V .

Equivalently: 2-colorable — no two adjacent vertices share the same color.

When is a graph bipartite?

\iff No odd-length cycles.

Example: Trees, even cycles, hypercubes.

Beyond Two Colors: Four Color Theorem

Every planar graph (or map) is 4-colorable — no two adjacent regions need more than four colors.

First major theorem proven with computer assistance (1976, Appel & Haken)

Graph Coloring Algorithm: Greedy Coloring

Algorithm (Greedy Coloring):

1 Order vertices: v_1, v_2, \dots, v_n

2 For each v_i in order:

Assign the smallest color not used by its already-colored neighbors.

Key Properties:

Time complexity: $O(V + E)$

Not optimal — may use $> \chi(G)$ colors

Performance depends on vertex ordering

Worst case: $\chi(G) + 1$ colors

Heuristics: DSATUR, Largest First,
Smallest Last

Vertex	Neighbors' Colors	Color Assigned
v_1	—	1
v_2	{1}	2
v_3	{1,2}	3
v_4	{2,3}	1

Example run of greedy coloring

Flood Fill: More Than Just a Paint Tool

It's Graph Traversal on a Grid

Each pixel is a node; edges connect to 4 neighbors.

Flood fill = find connected component of same color.

Why Queue? Avoid Stack Overflow

Recursive DFS crashes on large regions (e.g., 1M pixels).

Queue → iterative BFS → safe, predictable memory use.

BFS vs DFS: Shape Matters!

BFS (Queue): Circular, even fill — *used in Photoshop*

DFS (Stack): Jagged, spiky fill — *faster on small areas*

Applications

- Medical imaging: Segment tumors or organs
- Computer vision: Object detection via region growing
- Game engines: Territory control, pathfinding

Dijkstra's Algorithm

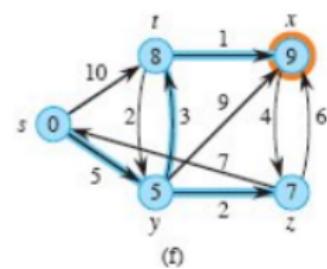
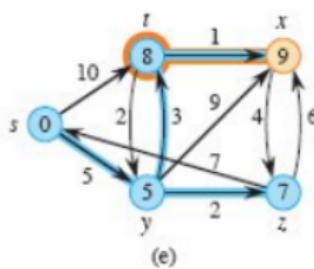
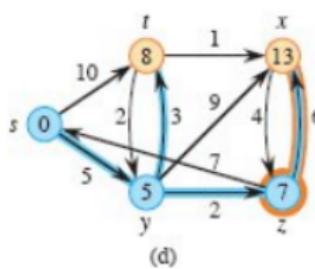
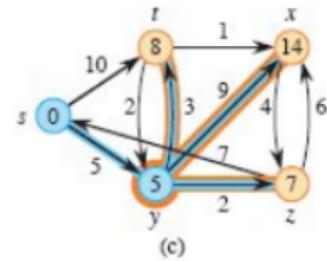
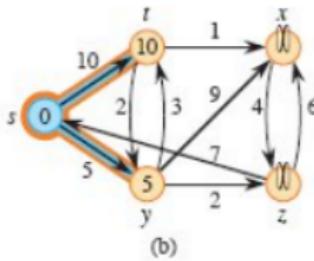
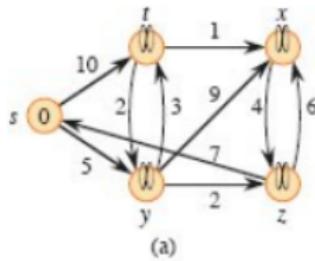
Goal: Find shortest paths from a source node to all other nodes in a weighted graph (non-negative weights).

Simple Steps:

- 1 Set distance to source = 0. Set all other distances to ∞ . Mark all nodes unvisited.
- 2 While there are unvisited nodes:
 - 3 Choose the unvisited node with the smallest known distance.
 - 4 For each neighbor of that node:
 - Add the edge weight to the current node's distance.
 - If this gives a shorter path to the neighbor, update its distance.
 - Mark the current node as visited.

Key idea: Greedily expand the closest unvisited node — guarantees optimal paths.

Dijkstra's Algorithm: Step-by-Step Execution



Dijkstra's algorithm: shortest path tree built step by step

Bellman-Ford: Shortest Paths with Negative Weights

Why Bellman-Ford?

- Handles **negative edge weights** (unlike Dijkstra)
- Detects **negative cycles** — paths with $-\infty$ weight
- Works on graphs where Dijkstra fails due to negative edges

Key Algorithmic Ideas

- Initialize all distances to ∞ , except source to 0
- Relaxation:** If $\text{dist}[u] + \text{wt} < \text{dist}[v]$, update $\text{dist}[v]$
- Repeat relaxation for **at most $V - 1$ iterations** (tree property and so not sensitive to neg. paths)
- Run V -th iteration to detect negative cycles

Complexity

- Time:** $O(VE)$ — $V - 1$ passes, E edges each
- Space:** $O(V)$ — distance array only
- vs Dijkstra:** $O(E \log V)$, but no negative edges allowed

Pseudocode

```
for i = 1 to V - 1 :  
    for each edge (u, v) with weight w :  
        if dist[u] + w < dist[v] :  
            dist[v] = dist[u] + w  
    for each edge (u, v) with weight w : // Negative cycle check  
        if dist[u] + w < dist[v] : return "Negative cycle detected"
```

Flow Network

A **flow network** is a directed graph $G = (V, E)$ with:

A **source** $s \in V$ and a **sink** $t \in V$

A **capacity function** $c : E \rightarrow \mathbb{R}_{\geq 0}$

A **flow function** $f : E \rightarrow \mathbb{R}_{\geq 0}$ satisfying:

1 **Capacity constraint:** $0 \leq f(u, v) \leq c(u, v)$

2 **Flow conservation:** $\sum_w f(w, u) = \sum_w f(u, w)$ for all $u \neq s, t$

Value of flow: $|f| = \sum_v f(s, v) - \sum_v f(v, s)$

Goal: Find a flow of **maximum value** from s to t .

Max-Flow Min-Cut Theorem & Duality

Cut: A partition (S, T) of V with $s \in S, t \in T$. **Capacity of cut:**

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

Max-Flow Min-Cut Theorem

$$\max_f |f| = \min_{(S, T)} c(S, T)$$

The maximum flow value equals the minimum cut capacity.

LP Duality Perspective

Max-flow is a linear program.

The dual of the max-flow LP corresponds to a fractional min-cut.

Strong duality \Rightarrow integral optimal solutions coincide.

Primal (Max-Flow) \leftrightarrow Dual (Min-Cut)