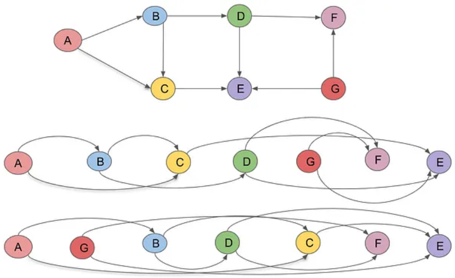
Fundamental Algorithmic Techniques VIII

November 28, 2025

Outline

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



Topological Sort

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.

Topological Sort: DFS Approach

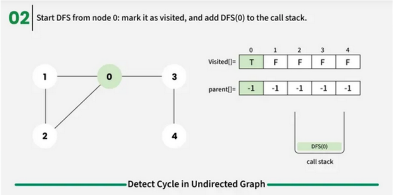
Algorithm:
DFS-Based Topological Sort
$O(V + E)$

1. Run DFS on the graph
2. When a vertex is **finished** (all descendants processed), push it onto stack S
3. The stack S (when popped) gives topological order

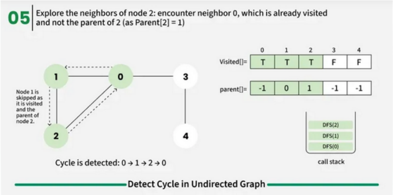
Key insight:
Vertices are ordered by *decreasing finishing times*

Cycle detection:
If back edge found during DFS $\Rightarrow$ cycle exists

Cycle Detection: DFS vs BFS — Complexity



Depth-First Search step 0



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).

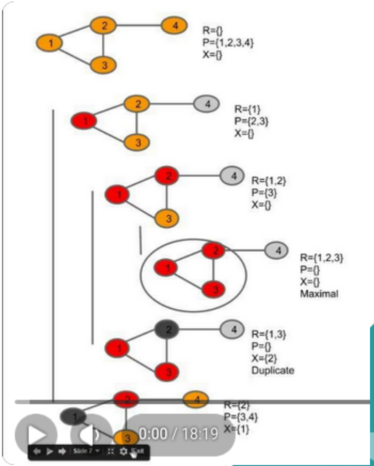
Bron–Kerbosch Algorithm: Maximal Clique Enumeration

Undirected graph $G = (V, E)$,
$N(v)$ = neighbors of v in G,

Initial call: 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}
```

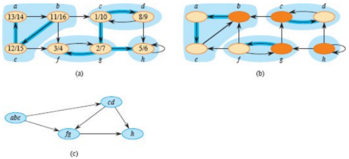


Kosaraju's Algorithm - Strongly Connected Components

Kosaraju's Algorithm

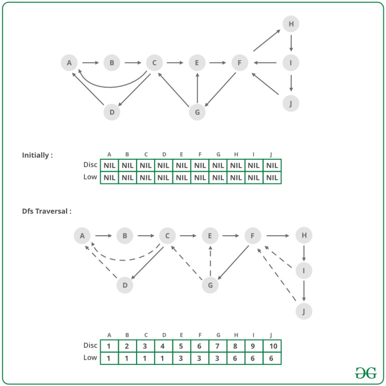
1. **DFS on Original Graph**: Record finish times
2. **Transpose the Graph**: Reverse all edges
3. **DFS on Transposed Graph**: Process nodes in order of decreasing finish times to find SCCs

- **Time Complexity**: Depth First Search: $\mathcal{O}(V + E)$
- **Space Complexity**: Stack: $\mathcal{O}(V)$



Two-pass DFS to find SCCs

Tarjan's Algorithm for SCCs



Index: Discovery order in DFS

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

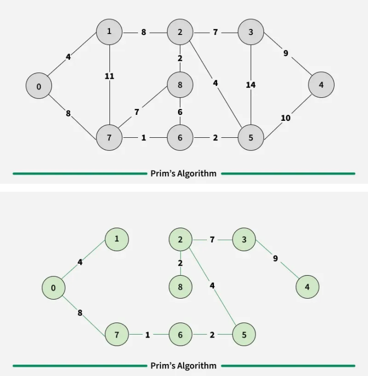
Problem: Random DFS order $\rightarrow$ ambiguous SCC boundaries

Solution: Use a stack to track active nodes

- When `low`$[u]$ = `index`$[u]$: pop stack until u — those form one SCC

Complexity: $\mathcal{O}(V + E)$ 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 $V \setminus S$.

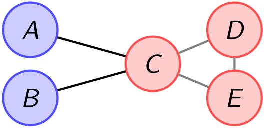
Cut-Crossing Edge

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

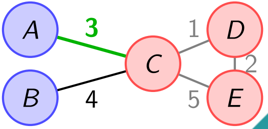
Cut Property / Theorem

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.**

Cut Illustration



Cut Property in Action



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 $w(e) \leq w(e')$.

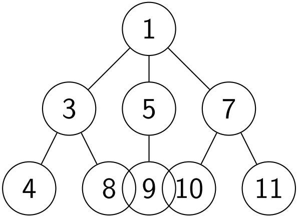
Conclusion

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

```
procedure JarnikMST(G = (V, E))
    Q ← empty min-heap                          ▷ Vertices with key values
    for v ∈ V do
        key[v] ← ∞
        Q.insert(v, key[v])
    end for
    key[0] ← 0                                   ▷ Start from vertex 0
    while Q is not empty do
        u ← Q.extractMin()
        for v ∈ Adj[u] and v ∈ Q do
            if w(u, v) < key[v] then
                parent[v] ← u
                key[v] ← w(u, v)
            end if
        end for
    end while
end procedure
```

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



Min-heap example with 3 children for root

Kruskal's Algorithm - Greedy MST Construction

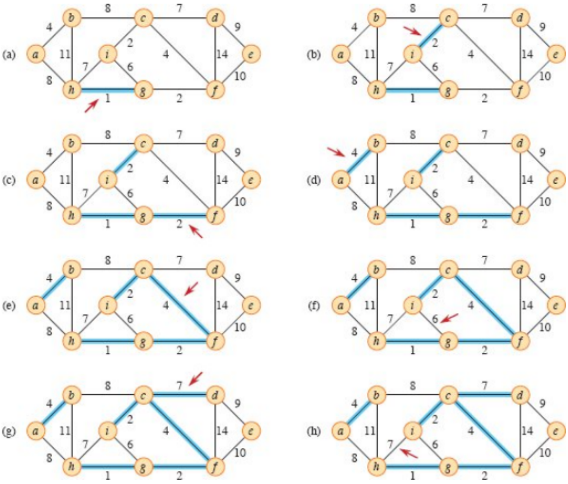
Kruskal's Algorithm Steps

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 `find(u)` $\neq$ `find(v)`
5. **Add to MST**: Include edge if no cycle
6. **Union**: Merge components using `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