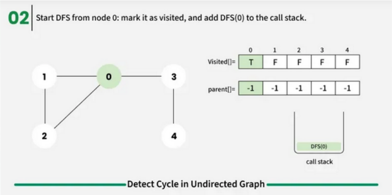
Fundamental Algorithmic Techniques VIII

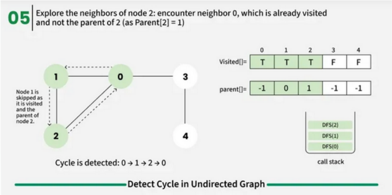
November 14, 2025

Outline

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., $D \to A$):
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).

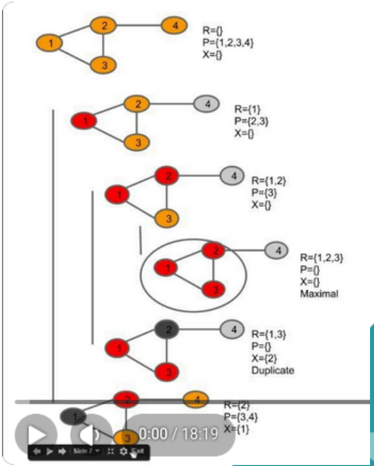
Bron–Kerbosch Algorithm: Maximal Clique Enumeration

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

Initial call: `BronKerbosch1(∅, V, ∅)`

Pseudocode:

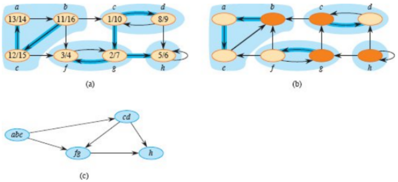
```
algorithm BronKerbosch1(R, P, X) is
    if P and X are both empty then
        report R as a maximal
clique
    for each vertex v in P do
        BronKerbosch1(R ∪ {v}, P ∩
N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```



Kosaraju Algorithm

$O(V + E)$

Kosaraju's Algorithm - Finding 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: O(V + E)

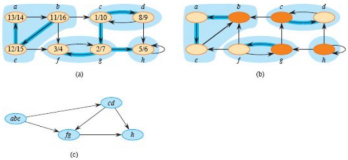
Space Complexity: O(V)

Key Insight: Transpose

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 Algorithm

$O(V + E)$

Kruskal's Algorithm - Greedy MST Construction

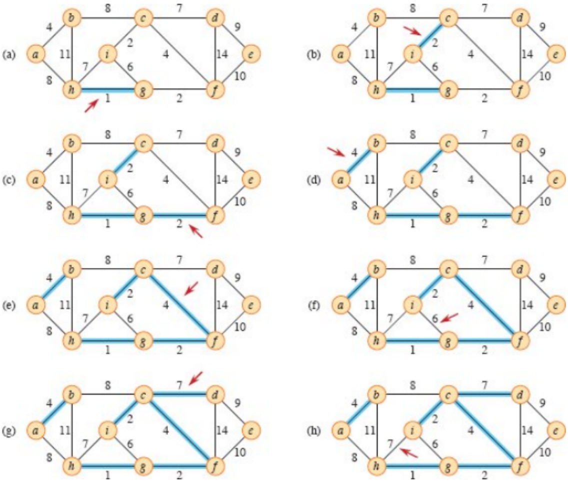
Kruskal's Algorithm Steps

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

Prim Algorithm

$O(V + E)$

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:

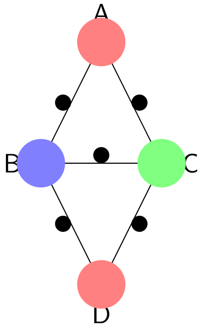
Vertices: Regions on a map or tasks needing resources

Edges: Conflicts

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

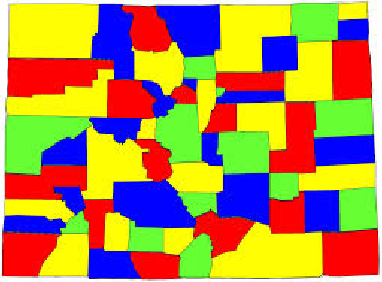
Real-world use cases:

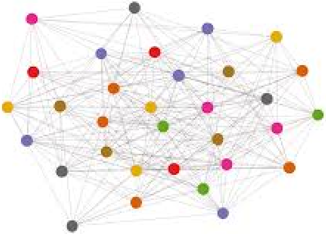
- Scheduling exams
- Register allocation in compilers
- Frequency assignment in wireless networks



A 3-coloring: A,D=red; B=blue; C=green

Nice examples of graph colouring problems





Bipartite Graphs

Graphs with 2 colours so that no two adjacent colours

Four Color Theorem: Any planar map can be colored with 4 colors

Graph Coloring Algorithm: Greedy Coloring

Algorithm (Greedy Coloring):

1. Order vertices: $v_1, v_2, \ldots, v_n$
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 Assig
v_1	—	1
v_2	$\{1\}$	2
v_3	$\{1,2\}$	3
v_4	$\{2,3\}$	1

Example run of greedy coloring

Bellman-Ford Algorithm

Dijkstra's Algorithm

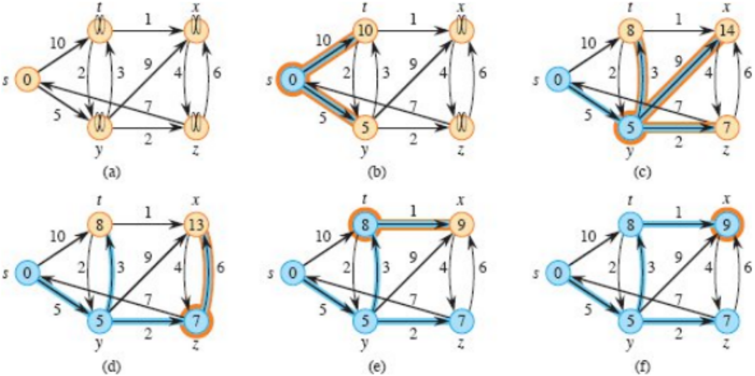
Goal: Find shortest paths from a source node to all other nodes in a weighted graph (non-negative weights).

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