Fundamental Algorithmic Techniques IX

November 24, 2025

Outline

Graph Colouring Algorithms

Shortest Paths

Flow Networks

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.

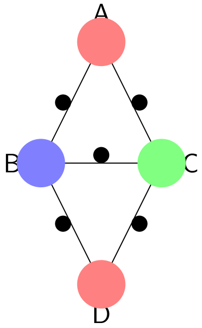
Example:

- Vertices: Regions on a map or tasks needing resources
- Edges: Conflicts

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

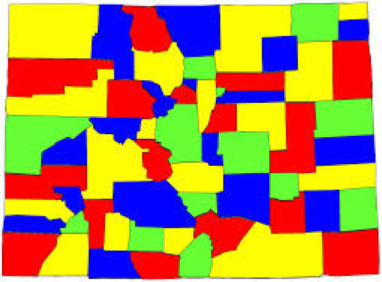
Real-world use cases:

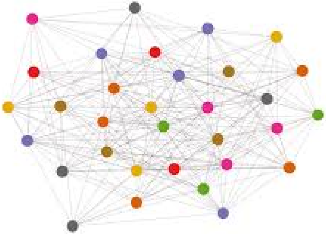
- Scheduling exams
- Register allocation in compilers
- Frequency assignment in wireless networks



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

Nice examples of graph colouring problems





Bipartite Graphs & 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: $v_1, v_2, \ldots, v_n$
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

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 $\rightarrow$ iterative BFS $\rightarrow$ 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

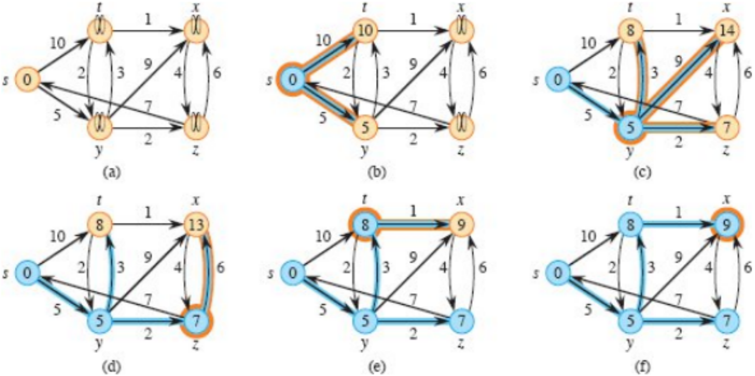
Goal: Find shortest paths from a source node to all other nodes in a weighted graph (non-negative weights).

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

1. **Time**: $O(VE)$ — $V - 1$ passes, E edges each
2. **Space**: $O(V)$ — distance array only
3. **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** $c : E \to \mathbb{R}_{\geq 0}$

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

$$\text{Primal (Max-Flow)} \quad \leftrightarrow \quad \text{Dual (Min-Cut)}$$