Algorithms for graph problems

Depth-First Search (DFS)

Here is a comprehensive guide to the Depth-First Search (DFS) algorithm, designed to be visual and easy to grasp without using complex mathematical notation.

What is Depth-First Search (DFS)?

Imagine you are in a massive maze. You want to find the exit. You stand at a fork in the road.

- **Strategy A:** You look down the left path, then the right path, just peeking a few steps in. (This is Breadth-First Search).
- **Strategy B (DFS):** You pick the left path and keep walking. If you hit another fork, you pick left again. You keep going until you hit a dead end. Only then do you retrace your steps (backtrack) to the last fork and try the right path.

Depth-First Search is Strategy B. It explores as far as possible along each branch before backtracking. It is an algorithm used to traverse or search tree and graph data structures.

Why Do We Need It?

DFS is fundamental for solving problems where you need to visit every location or find a specific path. Common uses include:

1. **Solving Puzzles:** Mazes, Sudoku, and crosswords (finding a solution by trying one path until it fails).
2. **Finding Connected Components:** Checking if every computer in a network is connected to every other computer.
3. **Pathfinding:** "Is there a path from Point A to Point B?" (Not necessarily the shortest one, but *a* path).
4. **Cycle Detection:** Checking if a series of dependencies loops back on itself (like A depends on B, B depends on C, and C depends on A).

Visual Walkthrough: How It Works

Let's trace a DFS on a simple graph.

The Rules:

1. Visit a node.
2. Mark it as "Visited" (so you don't go in circles).

3. Go to an unvisited neighbor and repeat.
4. If stuck (no unvisited neighbors), **Backtrack** (return to the previous node).

The Graph:

```
     (A)
    /   \
  (B)   (C)
  / \     \
(D) (E)   (F)
```

Step 1: Start at **A**. Mark A as visited.

- *Current Node:* A
- *Stack (History):* [A]

```
     [A]  <-- Current
    /   \
  (B)   (C)
```

Step 2: A has neighbors B and C. Let's pick **B** (convention usually goes left first).

- *Current Node:* B
- *Stack (History):* [A, B]

```
     (A)
    /
  [B]  <-- Current (Go Deep!)
  / \
(D) (E)
```

Step 3: B has neighbors D and E. We pick **D**.

- *Current Node:* D
- *Stack (History):* [A, B, D]

```
     (A)
    /
  (B)
  /
[D]  <-- Current
```

Step 4: D has no neighbors (Dead End!). We **Backtrack** by popping D off the stack and returning to B.

- *Current Node:* B
- *Stack (History):* [A, B]

```
     (A)
    /
  [B]  <-- Back at B. Any other paths? Yes, (E).
```

```
   / \
 (D) (E)
  ^
(Visited)
```

Step 5: From B, we visit the remaining neighbor **E**.

- *Current Node:* E
- *Stack (History):* [A, B, E]

```
      (A)
     /
   (B)
     \
     [E] <-- Current
```

Step 6: E is a dead end. Backtrack to B. B has no more unvisited neighbors. Backtrack to A.

- *Current Node:* A
- *Stack (History):* [A]

```
      [A] <-- Back at A. Any other paths? Yes, (C).
     /   \
   (B)   (C)
(All visited)
```

Step 7: Visit **C**. Then Visit **F**.

- Final Path Traveled: A -> B -> D -> E -> C -> F.

Complexity Analysis

We measure algorithms by Time (how long it takes) and Space (how much memory it uses).

Time Complexity: O(V + E)

- **V** = Vertices (Nodes/Dots)
- **E** = Edges (Lines/Connections)

Why? Because in the worst case, you visit every Node once and travel across every Edge once.

```
Time Taken Visualization:
|
| (Visiting every dot) + (Checking every line)
|
| [ VVVVVVV ] + [ EEEEEEEEEE ]
```

```
|__________________________________
             Total Work
```

Space Complexity: O(V) Why? Because of the **Stack** (or Recursion). In the worst-case scenario (a long straight line), your recursion stack will hold all the vertices at once before it can backtrack.

```
Space Used (The Stack) in Worst Case:

Graph: A -- B -- C -- D -- E

Memory Stack:
| [E] | < Top
| [D] |
| [C] |
| [B] |
| [A] | < Bottom
+-----+
Width is small, Height = Number of Vertices (V)
```

Algorithm Code

Here is how you write it. The **Recursive** method is the most common and easiest to write for interviews.

Python (Recursive)

```python
# Graph represented as a dictionary (Adjacency List)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}

visited = set() # To keep track of visited nodes

def dfs_recursive(node):
    # 1. Base Case: If already visited, stop.
    if node in visited:
        return

    # 2. Mark the current node as visited
```

```
        print(f"Visiting: {node}")
        visited.add(node)

        # 3. Explore neighbors
        for neighbor in graph[node]:
            dfs_recursive(neighbor)

# Start the process
print("DFS Path:")
dfs_recursive('A')
```

JavaScript (Recursive)

```
const graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
};

const visited = new Set();

function dfsRecursive(node) {
    // 1. If already visited, stop
    if (visited.has(node)) {
        return;
    }

    // 2. Mark and Process
    console.log(`Visiting: ${node}`);
    visited.add(node);

    // 3. Explore neighbors
    const neighbors = graph[node];
    for (let neighbor of neighbors) {
        dfsRecursive(neighbor);
    }
}

console.log("DFS Path:");
dfsRecursive('A');
```

Thinking in LeetCode: "The Flood Fill Strategy"

When you see a LeetCode problem involving a **Grid (Matrix)** or **Islands**, think DFS immediately.

Example Scenario: "Number of Islands" (Given a grid of '1's (land) and '0's (water), count the islands).

Mental Model: Imagine you are looking at a map. You drop a bucket of paint on a piece of land. The paint flows to all connected land but stops at water.

1. **Iterate:** Loop through every cell in the grid.
2. **Trigger:** If you see a "1" (Land) that hasn't been visited, increase your `IslandCount` by 1.
3. **DFS (The Paint):** Immediately trigger DFS on that cell.

- The DFS turns that cell into a "0" (or marks it visited).
- It then looks Up, Down, Left, Right.
- It moves to any neighbor that is a "1" and calls DFS on them.

4. **Result:** When the DFS finishes, it has "erased" that entire island. The loop continues to find the next island.

```
Grid before DFS:      Grid after DFS starts at (0,0):
1 1 0 0               x x 0 0
1 1 0 0      --->     x x 0 0
0 0 1 0               0 0 1 0  (The 'x's are marked visited)
```

Comparison: DFS vs. BFS

Feature	DFS (Depth-First)	BFS (Breadth-First)
Concept	Maze Solver (boldly go forward)	Radar / Ripple (scan everywhere nearby)
Data Structure	**Stack** (LIFO - Last In, First Out)	**Queue** (FIFO - First In, First Out)
Best For	Puzzles, Connecting components	Finding the **Shortest Path**
Memory	Good (O(h) where h is height)	Heavy (O(w) where w is width/widest part)

Final Tip: Use DFS when you need to search everything or just check for existence. Use BFS if you specifically need the *shortest* path in an unweighted graph.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a fundamental graph traversal algorithm. Think of it like a ripple in a pond: it starts at a center point and expands outward in concentric circles, visiting everything close by before moving further away.

1. The Problem It Solves

BFS is primarily used for finding the **shortest path** in an unweighted graph (a graph where the distance between all connected nodes is equal).

If you are looking for a key in a house, a random search might take you to the basement, then the attic, then the kitchen. BFS checks the room you are in, then the adjacent rooms, then the rooms adjacent to those. It guarantees that if a solution exists, you will find the one closest to the start.

Common scenarios:

- Finding the shortest route on a map (with equal road lengths).
- Social networks: "Who are the friends of my friends?" (2nd degree connections).
- Web crawlers: Analyzing links on a website level-by-level.

---

2. What is BFS?

BFS explores a graph **layer by layer**.

1. Start at a chosen node (Level 0).
2. Visit all immediate neighbors (Level 1).
3. Visit all neighbors of those neighbors (Level 2).
4. Repeat until all reachable nodes are visited.

It uses a **Queue** data structure (First-In, First-Out) to keep track of which node to visit next.

---

3. Visual Walkthrough

Let's look at this simple graph:

```
      [A]          <-- Start Node
     /   \
   [B]   [C]       <-- Neighbors of A
     \     \
     [D]   [E]     <-- Neighbors of B and C
```

Step 1: Initialization We start at `A`. We put `A` in the Queue and mark it as "Visited".

```
Queue: [A]
Visited: {A}
```

Step 2: Processing A We take `A` out of the queue. We look at its neighbors (`B` and `C`). We add them to the queue.

```
Processing: A
Queue: [B, C]     <-- B and C are added to the back
Visited: {A, B, C}
```

Step 3: Processing B We take `B` out. We look at its neighbor (`D`). We add `D` to the queue.

```
Processing: B
Queue: [C, D]     <-- C was already there, D joins the back
Visited: {A, B, C, D}
```

Step 4: Processing C We take `C` out. We look at its neighbor (`E`). We add `E` to the queue.

```
Processing: C
Queue: [D, E]
Visited: {A, B, C, D, E}
```

Step 5: Processing D & E We take `D` out. It has no unvisited neighbors. We take `E` out. It has no unvisited neighbors.

```
Queue: []
Status: Done.
Traversal Order: A -> B -> C -> D -> E
```

---

4. Complexity Analysis

- **V**: Vertices (Nodes)
- **E**: Edges (Connections)

Time Complexity: O(V + E) We visit every node once `O(V)` and for every node, we check its connections `O(E)`. Since we process the whole graph, the time taken grows linearly with the size of the graph.

Space Complexity: O(V) In the worst case, the Queue might have to store the majority of the nodes at once (imagine a "star" graph where the center node is connected to 1,000 other nodes; after processing the center, all 1,000 neighbors go into the queue).

```
Worst Case Space (Star Graph)
      [Center]
      /  /  |  \  \
    [1][2] [3] [4][5]...

Queue becomes: [1, 2, 3, 4, 5...] (Huge width)
```

---

5. Code Implementation

Python

```python
from collections import deque

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])

    # Mark start as visited immediately
    visited.add(start_node)

    result = []

    while queue:
        # Dequeue from the front (FIFO)
        current_node = queue.popleft()
        result.append(current_node)

        # Check all neighbors
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return result

# Example Graph
adj_list = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}

print(bfs(adj_list, 'A'))
# Output: ['A', 'B', 'C', 'D', 'E']
```

JavaScript

```javascript
function bfs(graph, startNode) {
    let visited = new Set();
    let queue = [startNode];
    visited.add(startNode);

    let result = [];

    while (queue.length > 0) {
        // Dequeue from front
        let currentNode = queue.shift();
        result.push(currentNode);

        let neighbors = graph[currentNode];
        for (let neighbor of neighbors) {
            if (!visited.has(neighbor)) {
                visited.add(neighbor);
                queue.push(neighbor);
            }
        }
    }
    return result;
}

const adjList = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
};

console.log(bfs(adjList, 'A'));
// Output: ['A', 'B', 'C', 'D', 'E']
```

---

6. LeetCode Mental Model

When solving problems, look for these triggers to use BFS:

1. **"Shortest Path" / "Minimum Steps":** If the problem asks for the minimum number of steps to convert X to Y (e.g., Word Ladder), use BFS.
2. **"Level Order":** Anything requiring you to process a tree row-by-row.
3. **"Rotting Oranges" / "Virus Spread":** Problems where a state

spreads to neighbors simultaneously each turn.

Strategy for "Grid" Problems (Islands, Mazes): Instead of an adjacency list, your "neighbors" are just the cells `(row+1, col)`, `(row-1, col)`, `(row, col+1)`, and `(row, col-1)`.

```
Grid Navigation Visual:
[ ][ ][ ]
[ ][X][ ]  <- X is current
[ ][ ][ ]

Neighbors are Up, Down, Left, Right.
Check bounds (don't go off grid) and check visited set.
```

---

7. Comparison with Other Algorithms

Feature	**BFS (Breadth-First)**	**DFS (Depth-First)**
Data Structure	Queue (FIFO)	Stack (LIFO) or Recursion
Visual Style	Expands in a circle (Wave)	Snakes deep into one path
Shortest Path?	**Yes** (Guaranteed for unweighted)	**No** (Might take a long detour)
Memory	High (Stores width of "wave")	Low (Stores length of path)
Best For	Finding closest target	Mazes, Puzzles requiring all solutions

Analogy:

- **BFS:** Scanning a crowd of people by looking at everyone in the front row, then everyone in the second row.
- **DFS:** Walking up to one person, asking who they know, walking to that person, asking who *they* know, until you hit a dead end.

Multi-Source Breadth First Search (Multi-Source BFS)

Multi-Source Breadth-First Search (BFS)

1. The Problem: Why Do We Need It?

Imagine you drop a single stone into a calm pond. Ripples expand outward from that one point. This is **Standard BFS**. It tells you the shortest distance from *one* starting point to anywhere else.

Now, imagine you drop **three** stones into the pond at the exact same time at different locations. Ripples expand from *all three* points simultaneously. If you want to know how soon a ripple reaches a specific leaf floating on the water, you care about the ripple that gets there *first*.

The Problem: You have a graph (or grid) and a set of "source" nodes. You need to find the shortest distance from *any* source node to every other node in the graph.

If you ran a Standard BFS from Source A, then another from Source B, and another from Source C, and then compared the results, it would be incredibly slow and inefficient. **Multi-Source BFS** solves this in a single pass.

---

2. What Is It?

Multi-Source BFS is a slight modification of the standard BFS algorithm.

- **Standard BFS:** Initialize the Queue with **one** starting node with distance 0.
- **Multi-Source BFS:** Initialize the Queue with **all** starting nodes with distance 0 simultaneously.

Functionally, you can imagine a "fake" invisible node (a Super Source) connected to all your actual starting nodes with an edge weight of 0. You run standard BFS from this Super Source.

---

3. Visual Walkthrough

Let's visualize a **virus spreading** on a grid.

- `S` = Source (Infected)
- `.` = Empty (Uninfected)
- `X` = Wall (Blockage)

The Grid:

```
(0,0) (0,1) (0,2) (0,3)
  S     .     S     .
  .     .     .     .
  .     X     .     .
```

Step 0: Initialization We identify all `S` nodes. We add them to our Queue. We set their distance to 0. All other nodes are "unvisited" (infinity).

- **Queue:** `[(0,0), (0,2)]`
- **Visited/Distance Map:**

```
0   inf   0   inf
inf inf   inf inf
inf inf   inf inf
```

Step 1: Process Distance 0 Nodes We pop `(0,0)` and `(0,2)` one by one. We look at their neighbors (Up, Down, Left, Right). If a neighbor is valid and not visited, we infect it (set distance to 1) and add it to the queue.

- Pop `(0,0)`: Neighbors are `(0,1)` and `(1,0)`.
- Pop `(0,2)`: Neighbors are `(0,1)` (already seen via 0,0? No, let's say we process 0,0 first, so 0,1 gets dist 1), `(0,3)`, `(1,2)`.
- **Queue:** `[(0,1), (1,0), (0,3), (1,2)]` (These are the "1"s)
- **Grid State (Current Distances):**

```
0     1     0     1
1     inf   1     inf
inf   X     inf   inf
```

Step 2: Process Distance 1 Nodes The "ripples" expand again. We process the nodes currently in the queue. Their unvisited neighbors become distance 2.

- Pop `(0,1)`: Neighbors `(1,1)` becomes 2.
- Pop `(1,0)`: Neighbor `(2,0)` becomes 2.
- Pop `(0,3)`: Neighbor `(1,3)` becomes 2.
- Pop `(1,2)`: Neighbors `(1,1)` (already visited), `(1,3)` (already visited), `(2,2)` becomes 2.
- **Queue:** `[(1,1), (2,0), (1,3), (2,2)]`
- **Grid State:**

```
0     1     0     1
1     2     1     2
2     X     2     inf
```

Step 3: Process Distance 2 Nodes Final expansion.

- Pop `(1,1)`: Neighbors `(1,0)` (visited), `(1,2)` (visited), `(0,1)` (visited), `(2,1)` (It's a wall 'X', ignore).
- …
- Pop `(2,2)`: Neighbor `(2,3)` becomes 3.
- **Final Grid:**

```
0     1     0     1
1     2     1     2
2     X     2     3
```

This tells us exactly how many steps it takes for the virus to reach any cell, regardless of which source it came from.

---

4. Implementation

Python Code Using `collections.deque` for an efficient O(1) pop from the left.

```python
from collections import deque

def multi_source_bfs(grid):
    rows = len(grid)
    cols = len(grid[0])

    # Queue for BFS
    queue = deque()

    # Visited set or distance matrix initialized to infinity
    # Using a distance matrix allows us to track steps
    dist = [[-1 for _ in range(cols)] for _ in range(rows)]

    # 1. INITIALIZATION: Add ALL sources to the queue
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 'S':  # 'S' is a source
                queue.append((r, c))
                dist[r][c] = 0

    # Directions: Up, Down, Left, Right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 2. STANDARD BFS LOOP
    while queue:
        curr_r, curr_c = queue.popleft()

        current_distance = dist[curr_r][curr_c]

        for dr, dc in directions:
            new_r, new_c = curr_r + dr, curr_c + dc

            # Check bounds
            if 0 <= new_r < rows and 0 <= new_c < cols:
                # Check if not visited (dist is -1) and not a wall
                if dist[new_r][new_c] == -1 and grid[new_r][new_c] != 'X':
                    dist[new_r][new_c] = current_distance + 1
                    queue.append((new_r, new_c))

    return dist
```

JavaScript Code

```
function multiSourceBFS(grid) {
    const rows = grid.length;
    const cols = grid[0].length;
    const queue = [];

    // Initialize distance matrix with -1 (representing infinity/unvisited)
    const dist = Array.from({ length: rows }, () => Array(cols).fill(-1));

    // 1. INITIALIZATION
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (grid[r][c] === 'S') {
                queue.push([r, c]);
                dist[r][c] = 0;
            }
        }
    }

    const directions = [[-1, 0], [1, 0], [0, -1], [0, 1]];
    let head = 0; // Optimization for queue pointer rather than shifting array

    // 2. BFS LOOP
    while (head < queue.length) {
        const [currR, currC] = queue[head++];
        const currentDist = dist[currR][currC];

        for (const [dr, dc] of directions) {
            const newR = currR + dr;
            const newC = currC + dc;

            if (newR >= 0 && newR < rows && newC >= 0 && newC < cols) {
                if (dist[newR][newC] === -1 && grid[newR][newC] !== 'X') {
                    dist[newR][newC] = currentDist + 1;
                    queue.push([newR, newC]);
                }
            }
        }
    }
    return dist;
}
```

---

5. Complexity Analysis

Let **V** be the number of vertices (or cells R * C) and **E** be the number of edges.

Time Complexity: O(V + E)

- Why? Even though we have multiple sources, every node is added to the queue and processed **at most once**.
- In a grid, E is roughly 4 * V, so it simplifies to O(V) or **O(R * C)**.

Space Complexity: O(V)

- Why? In the worst case (e.g., a completely filled grid), the queue might hold a significant portion of the nodes. We also store the `dist` array which is size V.
- In a grid: **O(R * C)**.

Visualization of Complexity

```
| Node Processing (Time) | Memory Usage (Space)   |
|------------------------|------------------------|
| [X] Processed Once     | [Q] Queue Storage      |
| [X] Processed Once     | [D] Distance Matrix    |
| [X] Processed Once     |                        |
| ...                    |                        |
| Total: Linear Prop.    | Total: Linear Prop.    |
| to Grid Size           | to Grid Size           |
```

---

6. LeetCode Strategy: How to Think

When you see a problem, ask yourself these questions to decide if you need Multi-Source BFS:

1. **"Nearest X to Y":** Does the problem ask for the distance to the *nearest* specific item (like nearest gate, nearest zero, nearest safe zone)?
2. **Simultaneous Start:** Does the process happen everywhere at once? (e.g., rotten oranges rotting neighbors simultaneously).
3. **Parallel Flooding:** Can I visualize the problem as water flooding from multiple leaks?

Classic LeetCode Examples:

- **01 Matrix (LeetCode 542):** Find distance of nearest 0 for each cell.
- *Strategy:* Push all '0' cells into queue first.
- **Rotting Oranges (LeetCode 994):** Minimum time until all oranges rot.
- *Strategy:* Push all initially rotten oranges into queue.

- **Map of Highest Peak (LeetCode 1765):** Assign heights such that water cells are 0 and height diff is at most 1.
- *Strategy:* Push all water cells into queue.

---

7. Comparison with Other Algorithms

Feature	Single-Source BFS	Multi-Source BFS	Dijkstra	Floyd-Warshall
Input Sources	One (e.g., Start)	Many (e.g., All Monsters)	One	All Pairs
Weights	Unweighted (1)	Unweighted (1)	Weighted (non-negative)	Weighted
Best For	Shortest path A to B	Shortest path {Set A} to rest	Shortest path with costs	All-to-All paths
Speed	Fast O(V+E)	Fast O(V+E)	Slower O(E log V)	Very Slow O(V^3)

> **Key Takeaway:** Multi-Source BFS is just BFS, but you "cheat" by starting the race with multiple runners already on the track.

Disjoint Set Union: Union-Find Algorithm

Here is a detailed, easy-to-understand guide to the Disjoint Set Union (DSU), also known as the Union-Find algorithm.

---

1. Why Do We Need It? (The Problem)

Imagine you are managing a social network. You have a massive list of people, and occasionally, two people become "friends." Friends of friends are also considered connected.

The Challenge: You frequently get two types of queries:

1. **Connect:** "Alice and Bob just became friends."
2. **Query:** "Are Alice and Charlie connected (directly or indirectly)?"

If you use standard Graph algorithms like BFS or DFS (Breadth-First Search) to answer "Are they connected?", it takes **O(N)** time (linear time) for *every* query. If you have millions of users and millions of queries, BFS is too slow.

The Solution: We need a way to connect items and check connectivity almost **instantly**. This is what Union-Find does. It solves the **Dynamic Connectivity Problem**.

---

2. What Is It?

Union-Find is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

Think of it as a collection of "gangs" or "clubs."

- Initially, everyone is in their own club of one.
- **Union:** Merging two clubs together.
- **Find:** Asking "Which club does this person belong to?" (represented by the club's leader).

If Person A and Person B have the same leader, they are in the same club.

---

3. How It Works

We represent these sets as **Trees**.

- Each element is a node.
- Each subset is a tree.
- The "Leader" (or Representative) is the **root** of the tree.

Key Optimizations To make this algorithm lightning fast, we use two tricks:

1. **Path Compression (The "Shortcut"):** When we look for the leader of a node, we make that node point *directly* to the leader so we don't have to traverse the whole chain next time. **Before Find(D):** A (Leader) -> B -> C -> D **After Find(D):** A (Leader) -> B A (Leader) -> C A (Leader) -> D *(Now C and D point directly to A)*
2. **Union by Rank/Size (The "Balance"):** When merging two trees, we always attach the **shorter** tree under the **taller** tree. This prevents the tree from becoming a long, thin line (linked list), keeping it flat and fast.

---

4. Visual Walkthrough

Let's say we have 5 elements: `0, 1, 2, 3, 4`.

Step 1: Initialization Everyone is their own parent (leader).

```
[0]    [1]    [2]    [3]    [4]
 ^      ^      ^      ^      ^
 |      |      |      |      |
Leader Leader Leader Leader Leader
```

- Parent Array: `[0, 1, 2, 3, 4]`

Step 2: Union(0, 1) Connect 0 and 1. We make 0 the leader of 1.

```
   0 (Leader)
  /
 1

[2]   [3]   [4]
```

- Parent Array: `[0, 0, 2, 3, 4]`

Step 3: Union(2, 3) Connect 2 and 3. We make 2 the leader of 3.

```
   0          2 (Leader)
  /          /
 1          3

[4]
```

- Parent Array: `[0, 0, 2, 2, 4]`

Step 4: Union(1, 4) We find the leader of 1 (which is 0). We find the leader of 4 (which is 4). We attach 4 to 0.

```
      0 (Leader)     2 (Leader)
     / \            /
    1   4          3
```

- Parent Array: `[0, 0, 2, 2, 0]`

Step 5: Union(3, 4) This connects the two big components.

1. Find leader of 3 -> **2**
2. Find leader of 4 -> **0**
3. Attach 2 (shorter tree) under 0 (taller tree).

```
      0 (Ultimate Leader)
    / | \
   1  4  2
         |
         3
```

- Now, `Find(3)` will return 0. `Find(1)` returns 0. They are connected.

---

5. Complexity Analysis

Using both Path Compression and Union by Rank, the time complexity is nearly constant.

Time Complexity: `O(alpha(N))`

- `alpha` is the **Inverse Ackermann function**.
- For all practical values of N (even up to the number of atoms in the universe), `alpha(N)` is **less than 5**.
- It is essentially `O(1)` (constant time) for practical purposes.

Space Complexity: `O(N)`

- We need an array to store the parent of each node.

Visualization of Speed:

```
Time taken per operation
^
|
|   O(N) - Linear (DFS/BFS)
|   /
|  /
| /
|/
|----------------------- O(log N) - Logarithmic (Binary Search)
|
|_______________________ O(alpha(N)) - Union Find (The floor!)
+-----------------------> Number of Elements
```

---

6. Implementation

Here is the efficient implementation with Path Compression and Union by Rank.

Python

```python
class UnionFind:
    def __init__(self, size):
        # Initially, each node is its own parent
        self.parent = list(range(size))
        # Rank helps to keep the tree flat (height approximation)
        self.rank = [1] * size

    def find(self, p):
        # Path Compression:
        # If p is not its own parent, recursively find the root
        # and update p's parent to directly point to the root.
        if self.parent[p] != p:
            self.parent[p] = self.find(self.parent[p])
        return self.parent[p]

    def union(self, p, q):
        rootP = self.find(p)
        rootQ = self.find(q)

        # If they are already in the same set, do nothing
        if rootP == rootQ:
            return False

        # Union by Rank: Attach smaller tree to larger tree
        if self.rank[rootP] > self.rank[rootQ]:
            self.parent[rootQ] = rootP
        elif self.rank[rootP] < self.rank[rootQ]:
            self.parent[rootP] = rootQ
        else:
            self.parent[rootQ] = rootP
            self.rank[rootP] += 1

        return True

# Usage
dsu = UnionFind(5)
dsu.union(0, 1)
dsu.union(2, 3)
dsu.union(1, 4)
print(dsu.find(3) == dsu.find(4)) # False (before connecting components)
dsu.union(3, 4)
print(dsu.find(3) == dsu.find(4)) # True (after connecting)
```

JavaScript

```
class UnionFind {
    constructor(size) {
        // Create array [0, 1, 2, ..., size-1]
        this.parent = Array.from({ length: size }, (_, i) => i);
        this.rank = new Array(size).fill(1);
    }

    find(p) {
        // Path Compression
        if (this.parent[p] !== p) {
            this.parent[p] = this.find(this.parent[p]);
        }
        return this.parent[p];
    }

    union(p, q) {
        const rootP = this.find(p);
        const rootQ = this.find(q);

        if (rootP === rootQ) return false;

        // Union by Rank
        if (this.rank[rootP] > this.rank[rootQ]) {
            this.parent[rootQ] = rootP;
        } else if (this.rank[rootP] < this.rank[rootQ]) {
            this.parent[rootP] = rootQ;
        } else {
            this.parent[rootQ] = rootP;
            this.rank[rootP] += 1;
        }
        return true;
    }
}
```

---

7. How to Apply to LeetCode Problems

Problem Idea: "Redundant Connection" (LeetCode 684)

- **The Prompt:** You are given a set of edges. A graph is a tree if it has no cycles. Find an edge that, if removed, makes the graph a tree (i.e., find the edge that created a cycle).

Strategy: Iterate through every edge `[u, v]` in the input.

1. Check `find(u)` and `find(v)`.
2. If `find(u) == find(v)`, it means `u` and `v` are **already connected** by a previous path. Adding this edge `[u, v]` would create a loop (a cycle). **This is your answer!**
3. If they are not connected, `union(u, v)` and continue.

Other Identifiers: Use Union-Find when you see keywords like:

- "Number of connected components"
- "Group the items together"
- "Equivalence classes"
- "Graph validity" (checking for cycles)

---

8. Comparison with Other Algorithms

Feature	Union-Find	DFS / BFS
Primary Use	Dynamic Connectivity (adding edges live)	Static Connectivity (graph is fixed)
Handling Cycles	Excellent (Detects cycle instantly)	Good (Detects via visited set)
Path Finding	Cannot find the *path* (only if connected)	Can find the actual path between nodes
Complexity	Almost O(1) per operation	O(V + E) per traversal
Implementation	Array-based (Very compact)	Stack/Queue + Adjacency List

Single Source Shortest Path: Djikstra's Algorithm

Here is a comprehensive guide to **Dijkstra's Algorithm**, explained using plain text, ASCII visualizations, and code, strictly adhering to your preference for non-mathematical notation.

---

1. The Problem: Why Do We Need This?

Imagine you are driving and using a GPS. You are at **Point A** (Home) and want to get to **Point B** (Work).

- There are many roads (edges) connecting many intersections (nodes).
- Each road has a "cost" or "weight." This could be **distance**, **time** (traffic), or **tolls**.

You don't just want *any* path; you want the **cheapest** path (shortest distance or fastest time).

The Challenge: A simple breadth-first search (BFS) assumes every road takes the same amount of time (weight = 1). In the real world, highways are faster than dirt roads. We need an algorithm that respects these different weights.

---

2. What is Dijkstra's Algorithm?

Dijkstra's Algorithm finds the shortest path from a single starting point (Source) to **all other points** in a graph with **non-negative weights**.

Key Analogy: Imagine dumping a bucket of water on the starting node. The water spreads along the pipes (edges). It will reach closer nodes first before flowing to further ones. Dijkstra's algorithm calculates exactly when the water reaches every other node.

Core Principles:

1. **Greedy Approach:** Always process the closest "unvisited" node next.
2. **Relaxation:** If we find a shortcut to a node through the current node, we update its distance.

- *Example:* If I thought it took 10 minutes to get to Node X, but I found a path through Node Y that takes only 8 minutes, I "relax" the edge and update Node X's distance to 8.

---

3. Visual Walkthrough (ASCII)

Let's find the shortest path from **Start Node (A)** to all other nodes.

The Graph:

- Nodes: A, B, C, D, E
- Weights are numbers on the lines.

```
    (4)
A --------> B
|           | \
|           |  \ (3)
| (2)       | (1)
v           v    \
C --------> D ---> E
    (5)      (1)
```

Connections (Weights):

- A -> B (4)
- A -> C (2)
- C -> B (1) <– Note: This creates a shortcut to B!
- C -> D (5)
- B -> D (1)
- B -> E (3)
- D -> E (1)

Initialization

We track two things:

1. **Shortest Distance:** Known distance from A to a node (starts at 'infinity' for everyone except A).

2. **Priority Queue:** A list of nodes to visit, sorted by shortest distance.

Start State:

- Current Node: **A** (Distance: 0)
- Unvisited: {B, C, D, E}

```
DISTANCES:
[ A: 0 ]
[ B: inf ]
[ C: inf ]
[ D: inf ]
[ E: inf ]
```

Step 1: Visit Node A

Look at neighbors of A: **B** and **C**.

- Distance to B = A's dist (0) + weight (4) = **4**. (4 is less than inf, so update).
- Distance to C = A's dist (0) + weight (2) = **2**. (2 is less than inf, so update).

Mark A as visited. Who is next? We look at the unvisited nodes with the smallest distance: B(4) vs C(2). **C is smaller.**

```
STATUS UPDATE:
Visited: {A}
Queue: [ C: 2, B: 4 ]  <-- C is at the front because 2 < 4

DISTANCES:
[ A: 0 ]
[ B: 4 ]
[ C: 2 ]  <-- Next to visit
[ D: inf ]
[ E: inf ]
```

Step 2: Visit Node C (Distance: 2)

Neighbors of C: **B** and **D**.

- **Check B:**
- Current known distance to B is **4** (via A->B).
- New path via C: (Distance to C) + (C->B) = 2 + 1 = **3**.
- **3 < 4**, so we found a shortcut! **Update B to 3.**

- **Check D:**
- New path via C: (Distance to C) + (C->D) = 2 + 5 = **7**.
- Update D to 7.

Mark C as visited. Who is next? Smallest in Queue: B(3) vs D(7). **B is smaller.**

```
STATUS UPDATE:
Visited: {A, C}
Queue: [ B: 3, D: 7 ]

DISTANCES:
[ A: 0 ]
[ B: 3 ]  <-- Updated from 4! (Shortcut found)
[ C: 2 ]
[ D: 7 ]
[ E: inf ]
```

Step 3: Visit Node B (Distance: 3)

Neighbors of B: **D** and **E**.

- **Check D:**
- Current known distance to D is **7** (via C).
- New path via B: (Distance to B) + (B->D) = 3 + 1 = **4**.
- **4 < 7**, shortcut found! **Update D to 4.**
- **Check E:**
- New path via B: (Distance to B) + (B->E) = 3 + 3 = **6**.
- Update E to 6.

Mark B as visited. Who is next? Smallest in Queue: D(4) vs E(6). **D is smaller.**

```
STATUS UPDATE:
Visited: {A, C, B}
Queue: [ D: 4, E: 6 ]

DISTANCES:
[ A: 0 ]
[ B: 3 ]
[ C: 2 ]
[ D: 4 ]  <-- Updated from 7!
[ E: 6 ]
```

Step 4: Visit Node D (Distance: 4)

Neighbors of D: **E**.

- **Check E:**
- Current known distance to E is **6**.
- New path via D: (Distance to D) + (D->E) = 4 + 1 = **5**.
- **5 < 6**, shortcut found! **Update E to 5.**

Mark D as visited. Only E is left.

```
STATUS UPDATE:
Visited: {A, C, B, D}
Queue: [ E: 5 ]

DISTANCES:
[ A: 0 ]
[ B: 3 ]
[ C: 2 ]
[ D: 4 ]
[ E: 5 ]  <-- Updated from 6!
```

Final Result

Shortest paths from A:

- A: 0
- B: 3 (Path: A -> C -> B)
- C: 2 (Path: A -> C)
- D: 4 (Path: A -> C -> B -> D)
- E: 5 (Path: A -> C -> B -> D -> E)

---

4. Complexity Analysis (ASCII)

Here is how the algorithm scales.

- **V** = Number of Vertices (Nodes)
- **E** = Number of Edges (Connections)

Time Complexity

Using a Min-Priority Queue (Heap): **O(E * log V)**

Think of it as: For every road (Edge), we might have to sort the list of cities (log V).

```
TIME SPENT
|
|         (Edges * log(Nodes))
|       ________________________
|      |                        |
|      | Processing Neighbors   |
|      | & Updating Queue       |
|      |________________________|
|__________________________________
           Input Size
```

Space Complexity

O(V + E) We need to store the graph (V + E) and the distances/priority queue (V).

```
MEMORY USED
|
|    [Graph Storage]    [Distances Array]
|    (Nodes + Edges)       (Nodes)
|       O(V + E)            O(V)
|__________________________________________
```

---

5. Code Implementation

Python

We use heapq because it is efficient.

```python
import heapq

def dijkstra(graph, start_node):
    # 1. Initialize distances to infinity, start_node to 0
    # Dictionary to store shortest distance to each node
    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0

    # 2. Priority Queue: stores tuples of (current_distance, node)
    # We start with the source node
    priority_queue = [(0, start_node)]

    while priority_queue:
        # Pop the node with the smallest distance
        current_dist, current_node = heapq.heappop(priority_queue)

        # Optimization: If we found a shorter way to this node already, skip
        if current_dist > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_dist + weight

            # If a shorter path is found
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # Push to queue with new priority
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example Graph (from the ASCII diagrams)
graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'D': 1, 'E': 3},
    'C': {'B': 1, 'D': 5},
    'D': {'E': 1},
    'E': {}
}

print(dijkstra(graph, 'A'))
# Output: {'A': 0, 'B': 3, 'C': 2, 'D': 4, 'E': 5}
```

JavaScript

JavaScript does not have a built-in Priority Queue, so for simple problems, we can use an array and sort it (less efficient) or implement a MinHeap class (efficient). Below is a conceptual version using a simple array sort for readability.

```
function dijkstra(graph, startNode) {
    // 1. Initialize distances
    let distances = {};
    for (let node in graph) {
        distances[node] = Infinity;
    }
    distances[startNode] = 0;

    // 2. Priority Queue (Simple Array for demo)
    // In production, use a proper MinHeap data structure
    let pq = [[0, startNode]];

    while (pq.length > 0) {
        // Sort to simulate Priority Queue (Smallest distance first)
        pq.sort((a, b) => a[0] - b[0]);

        // Get node with smallest distance
        let [currentDist, currentNode] = pq.shift();

        if (currentDist > distances[currentNode]) continue;

        // Check neighbors
        let neighbors = graph[currentNode];
        for (let neighbor in neighbors) {
            let weight = neighbors[neighbor];
            let newDist = currentDist + weight;

            if (newDist < distances[neighbor]) {
                distances[neighbor] = newDist;
                pq.push([newDist, neighbor]);
            }
        }
    }
    return distances;
}

const graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'D': 1, 'E': 3},
    'C': {'B': 1, 'D': 5},
    'D': {'E': 1},
    'E': {}
};

console.log(dijkstra(graph, 'A'));
```

---

6. LeetCode Strategy: How to use this

When you see a problem on LeetCode, ask yourself these questions to see if Dijkstra is the right tool:

1. **Is it a graph?** (Are there states/nodes and connections?)
2. **Are there weights?** (Does moving from A to B cost something different than C to D?)
3. **Are weights non-negative?** (Dijkstra fails if weights are negative).
4. **Are you looking for a minimum cost/time/path?**

Mental Template for Solving:

"I need to track the 'best so far' cost for every node. I will use a Min-Heap to always process the cheapest option next. If I find a cheaper way to a node, I update it and add it to the heap."

Common LeetCode variations:

- **Network Delay Time:** Find how long it takes for a signal to reach all nodes. (Classic Dijkstra).
- **Cheapest Flights Within K Stops:** Dijkstra, but with a constraint on the number of edges used (often requires a slight modification or Bellman-Ford if straightforward).
- **Path with Minimum Effort:** The "weight" might be the absolute difference in height between two cells in a grid.

---

7. Comparison with Other Algorithms

Algorithm	Best Used For	Key Difference	Weights Allowed
BFS (Breadth-First Search)	Unweighted Graphs	Explores layer by layer. Like Dijkstra but assumes all edge weights are 1.	N/A (assumes 1)
Dijkstra	Weighted Graphs	Uses a Priority Queue to prioritize lower costs.	Must be **Positive**
Bellman-Ford	Graphs with Negative Weights	Can detect negative cycles. Much slower than Dijkstra.	Positive & **Negative**
A* (A-Star)	Pathfinding to a specific Target	Uses a "Heuristic" (guess) to guide the search toward the goal faster.	Positive

Summary Algorithm Choice Flowchart (ASCII)

```
Is the graph Weighted?
      |
      NO -----> Use BFS (Simple Queue)
      |
      YES
      |
      v
Are there Negative Weights?
      |
      YES ----> Use Bellman-Ford
      |
      NO
      |
      v
Dijkstra's Algorithm (Min-Heap)
```

Single Source Shortest Path: Bellman Ford Algorithm

Here is a detailed guide to the Bellman-Ford Algorithm, designed to be clear and visual without using complex mathematical notation.

---

1. The Problem: Why do we need Bellman-Ford?

In graph theory, the **Single Source Shortest Path (SSSP)** problem asks: *"If I start at Node A, what is the shortest path to every other node in the graph?"*

Most people use **Dijkstra's Algorithm** for this. However, Dijkstra's algorithm fails if the graph has **negative edge weights**.

- **Example:** Imagine a game where traveling usually costs energy (positive weight), but sliding down a magic chute *gives* you energy back (negative weight). Dijkstra cannot handle the magic chute correctly.

The Solution: The **Bellman-Ford Algorithm** can calculate shortest paths even if edges have negative weights. It can also detect **Negative Weight Cycles** (infinite loops where you keep gaining energy forever).

---

2. What is it and How does it work?

Bellman-Ford is based on the **Relaxation Principle**. It is slower than Dijkstra but more versatile.

The Core Idea: "Relaxation"

To "relax" an edge means to check if we can improve the shortest path to a node by going through a specific neighbor.

- Current best distance to Node B is infinity.
- Current best distance to Node A is 5.

- There is an edge from A -> B with weight 2.
- **Relaxation:** Is `(Distance to A) + (Weight A->B) < (Current Distance to B)`?
- Is `5 + 2 < infinity`?
- Yes! Update distance to Node B to 7.

The Algorithm Logic

1. **Initialize:** Set the distance to the Source Node to 0 and all other nodes to Infinity.
2. **Iterate:** Repeat the following process **V - 1** times (where V is the number of vertices/nodes):

- Look at **every single edge** in the graph.
- Try to relax that edge.

3. **Check for Cycles:** Run the relaxation one more time. If any distance still changes, it means there is a **Negative Weight Cycle**.

---

3. Complexity Analysis

Here is the breakdown of the efficiency using ASCII visualizations.

Time Complexity

Since we iterate (V-1) times and check all E edges each time:

```
Time Complexity: O(V * E)

+------------------------+
| Loop V times           |  <-- Outer Loop (Vertices)
|   +------------------+ |
|   | Check E edges    | |  <-- Inner Loop (Edges)
|   +------------------+ |
+------------------------+
```

- **Best Case:** O(E) (If the edges happen to be ordered perfectly, though standard implementation is always O(V*E)).
- **Worst Case:** O(V * E).

Space Complexity

We only need to store the distance to each node.

```
Space Complexity: O(V)

[ Dist A | Dist B | Dist C | ... | Dist V ]
```

---

4. Visual Walkthrough (ASCII)

Let's find the shortest path from **Node A**.

The Graph:

- Nodes: A, B, C
- Edges:
- A -> B (weight: 4)
- A -> C (weight: 2)
- C -> B (weight: -10) **(Negative Edge!)**

Step 0: Initialization

Set Source (A) to 0, others to Infinity (Inf).

```
 (0)            (Inf)
  A ------------> B
  |             ^
  | (2)         | (-10)
  |             |
  v             |
  C ------------+
(Inf)
```

Iteration 1 (Relax all edges)

1. **Check A -> B (weight 4):**

- Is `Dist(A) + 4 < Dist(B)`?
- `0 + 4 < Inf`? **YES**.
- Update B to 4.

2. **Check A -> C (weight 2):**

- Is `Dist(A) + 2 < Dist(C)`?
- `0 + 2 < Inf`? **YES**.
- Update C to 2.

3. **Check C -> B (weight -10):**

- Is `Dist(C) + (-10) < Dist(B)`?
- `2 + (-10) < 4`?
- `-8 < 4`? **YES**.
- Update B to -8.

End of Iteration 1:

```
(0)              (-8)
 A ------------> B
 |             ^
 | (2)         | (-10)
 |             |
 v             |
 C ------------+
(2)
```

Note: We normally repeat this V-1 times (2 times here). In the second iteration, no values would change because the paths are already optimal.

Final Result:

- Path A -> B is cost -8 (Route: A -> C -> B).
- Path A -> C is cost 2.

---

5. Implementation

Python Code

```python
def bellman_ford(graph, num_nodes, start_node):
    # Step 1: Initialize distances
    # Using float('inf') to represent infinity
    distances = {node: float('inf') for node in range(num_nodes)}
    distances[start_node] = 0

    # Step 2: Relax edges V-1 times
    # 'graph' is a list of tuples: (u, v, weight)
    for _ in range(num_nodes - 1):
        for u, v, weight in graph:
            if distances[u] != float('inf') and distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    # Step 3: Check for Negative Cycles
    for u, v, weight in graph:
        if distances[u] != float('inf') and distances[u] + weight < distances[v]:
            print("Graph contains negative weight cycle")
            return None

    return distances

# Example Usage
# Nodes: 0, 1, 2
# Edges: 0->1 (4), 0->2 (2), 2->1 (-10)
edges = [
    (0, 1, 4),
    (0, 2, 2),
    (2, 1, -10)
]
print(bellman_ford(edges, 3, 0))
# Output: {0: 0, 1: -8, 2: 2}
```

JavaScript Code

```
function bellmanFord(edges, numNodes, startNode) {
    // Step 1: Initialize distances
    let distances = new Array(numNodes).fill(Infinity);
    distances[startNode] = 0;

    // Step 2: Relax edges V-1 times
    for (let i = 0; i < numNodes - 1; i++) {
        for (let [u, v, weight] of edges) {
            if (distances[u] !== Infinity && distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;
            }
        }
    }

    // Step 3: Check for Negative Cycles
    for (let [u, v, weight] of edges) {
        if (distances[u] !== Infinity && distances[u] + weight < distances[v]) {
            console.log("Graph contains negative weight cycle");
            return null;
        }
    }

    return distances;
}

// Example Usage
const edges = [
    [0, 1, 4],
    [0, 2, 2],
    [2, 1, -10]
];
console.log(bellmanFord(edges, 3, 0));
// Output: [ 0, -8, 2 ]
```

6. How to Use this for LeetCode

When you see a graph problem on LeetCode, use this mental checklist to decide if Bellman-Ford is the right tool:

1. **Does the problem mention costs/weights that can be negative?**

- *Yes:* Immediate Bellman-Ford signal.

2. **Does the problem ask to find a "cycle" that reduces cost?**

- *Yes:* This is a Negative Cycle Detection problem.

3. **Is the graph small?**

- Bellman-Ford is O(V*E). If V > 5,000, this might Time Out (TLE). Dijkstra is faster (O(E log V)) but stricter on weights.

Specific LeetCode Problem Strategy:

- **Problem:** "Cheapest Flights Within K Stops"
- **Strategy:** This is a modified Bellman-Ford. Instead of running the loop `V-1` times, you run it exactly `K+1` times. This restricts the path to only finding routes that use `K` stops or fewer.

---

7. Comparison Table

Feature	Bellman-Ford	Dijkstra	BFS (Breadth-First Search)
Edge Weights	Handles Positive & Negative	Must be Non-Negative	Unweighted (all edges = 1)
Time Complexity	Slow: O(V * E)	Fast: O(E log V)	Very Fast: O(V + E)
Detects Cycles?	Yes (Negative Cycles)	No	No (for weights)
Use Case	Financial graphs (arbitrage), small graphs with weird rules.	GPS, Maps, Network routing.	Peer-to-peer networks, web crawlers.

Practice Problem with negative weights: "The Time Travel Paradox"

The Scenario: Imagine you are navigating a network of star systems. Most routes take time (positive cost). However, there is a "wormhole" that actually sends you back in time (negative cost).

The Question: Determine if there is a path in this star map where you can keep looping forever to travel infinitely far back in the past. In Computer Science terms: **"Does this graph contain a Negative Weight Cycle?"**

---

1. The Graph Setup (ASCII)

- **Nodes (Star Systems):** 0, 1, 2
- **Edges (Routes):**
- 0 -> 1 (Cost: 1)
- 1 -> 2 (Cost: -5) <– *The Wormhole*
- 2 -> 0 (Cost: 3)

Cycle Math: If you go 0 -> 1 -> 2 -> 0: `1 + (-5) + 3 = -1` Every time you complete this loop, the total cost decreases by 1. This is a **Negative Cycle**.

---

2. Step-by-Step Algorithm Walkthrough

We have **3 Nodes** (V=3). We must run the relaxation loop **V-1 times** (2 times). Finally, we run it **1 extra time** to detect the cycle.

Initialization: `Distances: [0, Inf, Inf]` (Start at Node 0)

Iteration 1 (Standard Bellman-Ford Pass) We check all edges:

1. **Edge 0->1 (weight 1):**

- `Dist[0] + 1 < Dist[1]`? -> `0 + 1 < Inf`? **Yes.**
- Update `Dist[1]` to **1**.
- *State: [0, 1, Inf]*

2. **Edge 1->2 (weight -5):**

- `Dist[1] + (-5) < Dist[2]`? -> `1 - 5 < Inf`? **Yes.**
- Update `Dist[2]` to **-4**.
- *State: [0, 1, -4]*

3. **Edge 2->0 (weight 3):**

- `Dist[2] + 3 < Dist[0]`? -> `-4 + 3 < 0`? **Yes.**
- Update `Dist[0]` to **-1**.
- *State: [-1, 1, -4]*

Iteration 2 (Standard Bellman-Ford Pass) We check all edges again using the new values `[-1, 1, -4]`:

1. **Edge 0->1 (weight 1):**

- `Dist[0] + 1 < Dist[1]`? -> `-1 + 1 < 1`? **Yes.**

- Update `Dist[1]` to **0**.
- *State: [-1, 0, -4]*

2. **Edge 1->2 (weight -5):**

- `Dist[1] - 5 < Dist[2]`? -> `0 - 5 < -4`? **Yes.**
- Update `Dist[2]` to **-5**.
- *State: [-1, 0, -5]*

3. **Edge 2->0 (weight 3):**

- `Dist[2] + 3 < Dist[0]`? -> `-5 + 3 < -1`? **Yes.**
- Update `Dist[0]` to **-2**.
- *State: [-2, 0, -5]*

The Detection Step (Iteration 3) According to the algorithm, after V-1 iterations, we should be done. **If we can still relax an edge, a cycle exists.**

Let's test **Edge 0->1** again:

- Current `Dist[0]` is **-2**.
- Current `Dist[1]` is **0**.
- Check: `Dist[0] + 1 < Dist[1]`?
- `-2 + 1 < 0`? -> `-1 < 0`? **YES!**

Conclusion: Because we could still reduce the cost in the extra step, we have proven a **Negative Weight Cycle** exists. Infinite time travel is possible!

---

3. Solution Code

Here is how you would write a function specifically to **detect** this boolean condition.

Python Solution

```python
def has_negative_cycle(num_nodes, edges, start_node):
    # Step 1: Initialize
    dist = {i: float('inf') for i in range(num_nodes)}
    dist[start_node] = 0

    # Step 2: Relax V-1 times
    for _ in range(num_nodes - 1):
        for u, v, weight in edges:
            if dist[u] != float('inf') and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Step 3: Cycle Detection Check
    # We iterate through all edges ONE more time
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            print(f"Cycle detected! Node {u} to {v} can still be reduced.")
            return True # Cycle exists!

    return False # Safe graph

# Test with our Time Travel graph
# Nodes: 0, 1, 2
# Edges: (0->1, w=1), (1->2, w=-5), (2->0, w=3)
my_edges = [
    (0, 1, 1),
    (1, 2, -5),
    (2, 0, 3)
]

print(f"Negative Cycle Detected? {has_negative_cycle(3, my_edges, 0)}")
```

JavaScript Solution

```javascript
function hasNegativeCycle(numNodes, edges, startNode) {
    let dist = new Array(numNodes).fill(Infinity);
    dist[startNode] = 0;

    // Step 2: Relax V-1 times
    for (let i = 0; i < numNodes - 1; i++) {
        for (let [u, v, weight] of edges) {
            if (dist[u] !== Infinity && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Step 3: Cycle Detection Check
    for (let [u, v, weight] of edges) {
        if (dist[u] !== Infinity && dist[u] + weight < dist[v]) {
            console.log(`Cycle detected between ${u} and ${v}`);
            return true; // Cycle exists!
        }
    }

    return false;
}

const myEdges = [
    [0, 1, 1],
    [1, 2, -5],
    [2, 0, 3]
];

console.log("Negative Cycle Detected?", hasNegativeCycle(3, myEdges, 0));
```

Minimum Spanning Tree (MST): Prim's Algorithm

What Problem Does It Solve?

Imagine you are a city planner who needs to connect several neighborhoods with fiber optic cables.

- **The Goal:** Every neighborhood must be connected to the network (directly or indirectly).
- **The Constraint:** You want to minimize the total cost of the cable used.
- **The Catch:** You don't need redundant loops. If Neighborhood A is connected to B, and B to C, you don't need a direct cable from A to C if the cost is higher.

This is the **Minimum Spanning Tree (MST)** problem. You have a "graph" of points (vertices) and possible connections (edges) with costs (weights). You need to select a subset of edges that connects everyone with the lowest total weight and no cycles.

What is Prim's Algorithm?

Prim's Algorithm is a "greedy" strategy to find this MST. It builds the tree one node at a time, always picking the cheapest connection available from the nodes you have *already visited* to a node you *haven't visited yet.*

Core Logic:

1. Start at an arbitrary node.
2. Look at all edges connecting your current "tree" to the outside world.
3. Pick the shortest (cheapest) edge that leads to a new node.
4. Add that node and edge to your tree.
5. Repeat until all nodes are included.

Visual Walkthrough

Let's trace Prim's algorithm on a simple graph.

The Graph: Nodes: A, B, C, D, E

Weights:

- A-B: 4
- A-C: 2
- B-C: 3
- B-D: 2
- B-E: 3
- C-D: 4
- C-E: 5
- D-E: 1

Step 1: Start at Node A

- **Visited:** {A}
- **Available Edges from Visited:**
- A -> B (cost 4)
- A -> C (cost 2)
- **Decision:** A -> C is cheaper (2 vs 4). Choose A-C.

```
    A
   /
 (2)
 /
C
```

Step 2: Tree is now {A, C}

- **Visited:** {A, C}
- **Available Edges from {A, C} to Unvisited:**
- A -> B (cost 4)
- C -> B (cost 3)
- C -> D (cost 4)
- C -> E (cost 5)
- **Decision:** C -> B is the cheapest (cost 3). Choose C-B.
- *Note: We ignore A->B (cost 4) because we found a cheaper way to get to B via C.*

```
    A
   /
 (2)
 /
C ---(3)--- B
```

Step 3: Tree is now {A, C, B}

- **Visited:** {A, C, B}
- **Available Edges from {A, C, B} to Unvisited:**
- C -> D (cost 4)
- C -> E (cost 5)
- B -> D (cost 2)
- B -> E (cost 3)
- **Decision:** B -> D is the cheapest (cost 2). Choose B-D.

```
    A
   /
  (2)
 /
C ---(3)--- B
            |
           (2)
            |
            D
```

Step 4: Tree is now {A, C, B, D}

- **Visited:** {A, C, B, D}
- **Available Edges from {A, C, B, D} to Unvisited:**
- C -> E (cost 5)
- B -> E (cost 3)
- D -> E (cost 1)
- **Decision:** D -> E is the cheapest (cost 1). Choose D-E.

```
    A
   /
  (2)
 /
C ---(3)--- B
            |
           (2)
            |
            D ---(1)--- E
```

Result: All nodes {A, B, C, D, E} are visited. Total Cost: $2 + 3 + 2 + 1 =$ **8**

Complexity Analysis

Here is the breakdown of efficiency using a Min-Priority Queue (the standard implementation).

V = Number of Vertices (Nodes) E = Number of Edges

Metric	Complexity	Explanation
Time	O(E * log V)	For every edge, we might push/pop from the priority queue. The heap operations take logarithmic time relative to the number of vertices.
Space	O(V + E)	We need to store the graph (adjacency list) and the arrays to track weights and visited status.

Ascii Visualization of Complexity:

```
Time Complexity Scale:
Low -----------------------------------------> High
       O(1)      O(V)      O(E log V)     O(V^2)
                            ^
                            |
                   Prim's (Binary Heap)
```

Implementation

Here is how you would code this efficiently using a Min-Heap (Priority Queue).

Python Code

```python
import heapq

def prims_algorithm(n, edges):
    # n: number of nodes
    # edges: list of [u, v, weight]

    # Build Adjacency List
    adj = {i: [] for i in range(n)}
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w)) # MST is for undirected graphs

    # Min-Heap to pick smallest weight edge
    # Format: (weight, node_to)
    min_heap = [(0, 0)] # Start at node 0 with cost 0

    visited = set()
    total_cost = 0
    mst_edges = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)

        # If we already visited this node, skip it
        if u in visited:
            continue

        visited.add(u)
        total_cost += weight
        mst_edges += 1

        # If we have visited all nodes, we can stop early
        if mst_edges == n:
            break

        # Check neighbors
        for v, w in adj[u]:
            if v not in visited:
                heapq.heappush(min_heap, (w, v))

    return total_cost if len(visited) == n else -1

# Example Usage
# Nodes 0, 1, 2, 3, 4 represent A, B, C, D, E
edges_data = [
    [0, 1, 4], [0, 2, 2], [1, 2, 3],
    [1, 3, 2], [1, 4, 3], [2, 3, 4],
    [2, 4, 5], [3, 4, 1]
]
print(prims_algorithm(5, edges_data)) # Output: 8
```

JavaScript Code

```
class MinPriorityQueue {
  // Simple implementation of a Min Priority Queue for visualization
  constructor() { this.heap = []; }

  enqueue(element, priority) {
    this.heap.push({ element, priority });
    this.heap.sort((a, b) => a.priority - b.priority); // O(N log N) for simplicity here
  }

  dequeue() { return this.heap.shift(); }
  isEmpty() { return this.heap.length === 0; }
}

function primsAlgorithm(n, edges) {
    const adj = new Map();
    for (let i = 0; i < n; i++) adj.set(i, []);

    for (const [u, v, w] of edges) {
        adj.get(u).push([v, w]);
        adj.get(v).push([u, w]);
    }

    const pq = new MinPriorityQueue();
    pq.enqueue(0, 0); // Start at node 0, cost 0

    const visited = new Set();
    let totalCost = 0;

    while (!pq.isEmpty()) {
        const { element: u, priority: weight } = pq.dequeue();

        if (visited.has(u)) continue;

        visited.add(u);
        totalCost += weight;

        if (visited.size === n) break;

        for (const [v, w] of adj.get(u)) {
            if (!visited.has(v)) {
                pq.enqueue(v, w);
            }
        }
    }

    return visited.size === n ? totalCost : -1;
}
```

```
// Example Usage
const edgesData = [
    [0, 1, 4], [0, 2, 2], [1, 2, 3],
    [1, 3, 2], [1, 4, 3], [2, 3, 4],
    [2, 4, 5], [3, 4, 1]
];
console.log(primsAlgorithm(5, edgesData)); // Output: 8
```

Algorithm Comparisons

It is helpful to know when to use Prim's versus others.

Feature	Prim's Algorithm	Kruskal's Algorithm	Dijkstra's Algorithm
Goal	Minimum Spanning Tree (MST)	Minimum Spanning Tree (MST)	Shortest Path from Source
Strategy	Grow a single tree from a source node.	Sort all edges by weight and merge disjoint sets.	Grow path from source, minimizing total distance from start.
Best For	**Dense Graphs** (lots of edges per node).	**Sparse Graphs** (fewer edges).	Finding shortest route (GPS).
Data Structure	Priority Queue + Visited Set	Union-Find (Disjoint Set) + Sorting	Priority Queue + Distance Array
Analogy	A mold spreading to nearest food.	Building bridges between islands until all connected.	Driving directions.

Applying to LeetCode: "Min Cost to Connect All Points"

A classic problem is **LeetCode 1584: Min Cost to Connect All Points**.

The Setup: You are given an array of coordinates `points = [[x1,y1], [x2,y2]...]`. You need to connect all points. The cost is the Manhattan distance: `|x1 - x2| + |y1 - y2|`.

How to Think About It:

1. **Identify the structure:** “Connect all points” + “Min Cost” = Minimum Spanning Tree.
2. **Dense Graph:** Since *every* point can connect to *every other* point, this is a “Complete Graph” (Dense).
3. **Selection:** Prim’s is usually preferred over Kruskal’s for dense graphs because Kruskal’s requires sorting O(E log E) edges. In a complete graph, E is roughly V^2, making sorting very expensive. Prim’s avoids processing every single edge explicitly if implemented optimally.

Strategy:

- Treat point 0 as your starting node.
- Calculate the distance from point 0 to all others and push to a Min-Heap.
- Pop the closest point, add it to visited, and calculate distances from *that* point to remaining unvisited points.

Would you like me to generate a solution specifically for that LeetCode problem?

Minimum Spanning Tree (MST): Kruskal’s Algorithm

Here is a detailed guide to Minimum Spanning Trees (MST) and Kruskal’s Algorithm.

The Problem: Why Do We Need This?

Imagine you are a city planner who needs to connect several neighborhoods with fiber-optic cables.

- **Goal:** You must ensure *every* neighborhood is connected to the network.
- **Constraint:** You have a limited budget, so you want to use the least amount of cable possible.
- **Rule:** You don’t need redundant loops (cycles). If Neighborhood A connects to B, and B connects to C, A is already connected to C. Adding a direct cable from A to C is unnecessary expense.

This is the **Minimum Spanning Tree** problem. You need a subset of connections (edges) that connects all points (vertices) with the minimum total cost (weight).

What is Kruskal’s Algorithm?

Kruskal’s Algorithm is a **greedy** strategy to solve the MST problem. It builds the spanning tree by adding edges one by one into a growing spanning tree.

The Strategy:

1. **Be Cheap:** Always pick the smallest weight edge available in the entire graph.
2. **Be Safe:** Only add that edge if it doesn't create a loop (cycle). If it creates a loop, discard it.
3. **Repeat:** Keep doing this until all vertices are connected.

To handle the "cycle detection" efficiently, Kruskal's uses a data structure called **Disjoint Set Union (DSU)** or **Union-Find**. This keeps track of which vertices are already in the same group.

Visual Walkthrough

Let's visualize this with a graph.

The Graph:

- **Vertices (Nodes):** 0, 1, 2, 3
- **Edges (Connections):**
- 0 to 1 (Weight: 10)
- 0 to 2 (Weight: 6)
- 0 to 3 (Weight: 5)
- 1 to 3 (Weight: 15)
- 2 to 3 (Weight: 4)

Step 1: Sort All Edges First, we list all edges and sort them from smallest weight to largest.

```
|   Source  | Destination |  Weight  |
|-----------|-------------|----------|
|     2     |      3      |     4    |  <-- Smallest
|     0     |      3      |     5    |
|     0     |      2      |     6    |
|     0     |      1      |    10    |
|     1     |      3      |    15    |  <-- Largest
```

Step 2: Processing Edges

Iteration 1: Pick edge (2, 3) with weight 4.

- Are 2 and 3 connected? No.
- **Action:** Add edge. Join sets {2} and {3}.

```
Visual:
(0)     (1)

(2)=====(3)  <-- Edge added
```

Iteration 2: Pick edge (0, 3) with weight 5.

- Are 0 and 3 connected? No.

- **Action:** Add edge. Join set {0} with set {2, 3}.

```
Visual:
(0)     (1)
  \
   \
(2)=====(3)
```

Iteration 3: Pick edge (0, 2) with weight 6.

- Are 0 and 2 connected?
- Trace: 0 is connected to 3, and 3 is connected to 2.
- Yes, they are in the same group.
- **Action:** Discard (SKIP). Adding this would create a triangle cycle (0-2-3).

```
Visual (No Change):
(0)     (1)
  \
   \
(2)=====(3)
   ^
   |
 Attempted to add connection here (0-2), but rejected.
```

Iteration 4: Pick edge (0, 1) with weight 10.

- Are 0 and 1 connected? No.
- **Action:** Add edge. Join set {1} with set {0, 2, 3}.

```
Visual:
(0)-----(1)
  \
   \
(2)=====(3)
```

Iteration 5: Pick edge (1, 3) with weight 15.

- Are 1 and 3 connected? Yes (via 0).
- **Action:** Discard (SKIP).

Final Result:

- Edges: (2,3), (0,3), (0,1)
- Total Minimum Cost: 4 + 5 + 10 = **19**

Complexity Analysis

Here is the breakdown of the efficiency.

- **V** = Number of Vertices
- **E** = Number of Edges

Type	Complexity	Explanation
Time	**O(E log E)**	Sorting the edges takes O(E log E). The Union-Find operations take nearly constant time, usually written as O(E * alpha(V)), which is very fast. The sorting is the bottleneck.
Space	**O(V + E)**	We need space to store the list of edges O(E) and the parent array for Union-Find O(V).

Implementation

We will use a `UnionFind` helper class to manage the groups efficiently.

Python

```python
class UnionFind:
    def __init__(self, n):
        # Initially, each node is its own parent
        self.parent = list(range(n))
        # Rank helps keep the tree flat during union
        self.rank = [0] * n

    def find(self, i):
        # Path compression: Point directly to the root
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            # Union by rank: attach smaller tree to larger tree
            if self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            elif self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            else:
                self.parent[root_j] = root_i
                self.rank[root_i] += 1
            return True # Union was successful
        return False # Cycle detected

def kruskals(num_vertices, edges):
    """
    edges: List of [source, dest, weight]
    """
    mst_weight = 0
    mst_edges = []

    # Step 1: Sort edges by weight (ascending)
    # x[2] corresponds to the weight
    edges.sort(key=lambda x: x[2])

    uf = UnionFind(num_vertices)

    for u, v, w in edges:
        # Step 2: Check if adding edge creates a cycle
        if uf.union(u, v):
            mst_weight += w
            mst_edges.append([u, v, w])

    return mst_weight, mst_edges

# Example Usage
# Nodes: 0, 1, 2, 3
edges_list = [
    [0, 1, 10],
    [0, 2, 6],
    [0, 3, 5],
    [1, 3, 15],
    [2, 3, 4]
]

weight, tree = kruskals(4, edges_list)
print("MST Weight:", weight)
print("Edges in MST:", tree)
```

JavaScript

```
class UnionFind {
    constructor(n) {
        this.parent = Array.from({ length: n }, (_, i) => i);
        this.rank = new Array(n).fill(0);
    }

    find(i) {
        if (this.parent[i] !== i) {
            this.parent[i] = this.find(this.parent[i]);
        }
        return this.parent[i];
    }

    union(i, j) {
        let rootI = this.find(i);
        let rootJ = this.find(j);

        if (rootI !== rootJ) {
            if (this.rank[rootI] > this.rank[rootJ]) {
                this.parent[rootJ] = rootI;
            } else if (this.rank[rootI] < this.rank[rootJ]) {
                this.parent[rootI] = rootJ;
            } else {
                this.parent[rootJ] = rootI;
                this.rank[rootI] += 1;
            }
            return true; // Edge added
        }
        return false; // Cycle detected
    }
}

function kruskals(numVertices, edges) {
    let mstWeight = 0;
    let mstEdges = [];

    // Step 1: Sort edges by weight
    edges.sort((a, b) => a[2] - b[2]);

    const uf = new UnionFind(numVertices);

    // Step 2: Iterate and build MST
    for (let [u, v, w] of edges) {
        if (uf.union(u, v)) {
                mstWeight += w;
                mstEdges.push([u, v, w]);
            }
        }

    return { mstWeight, mstEdges };
}

// Example Usage
const edges = [
    [0, 1, 10], [0, 2, 6], [0, 3, 5],
    [1, 3, 15], [2, 3, 4]
];

const result = kruskals(4, edges);
console.log("MST Weight:", result.mstWeight);
console.log("MST Edges:", result.mstEdges);
```

Strategy for LeetCode Problems

When you see a problem on LeetCode, use Kruskal's if you spot these clues:

1. **"Connect all points":** The problem asks to link cities, computers, or nodes.
2. **"Minimum Cost":** You need to minimize the sum of weights (cost, distance, time).
3. **Sparse Graphs:** Kruskal's is often easier to implement and slightly faster on graphs with fewer edges compared to vertices (Sparse Graphs).

Mental Model for Solving: Think of the problem as a "Merging Islands" scenario.

- Initially, every node is its own island.
- Bridges (edges) are available at different prices.
- Buy the cheapest bridges first.
- If a bridge connects two nodes that are already on the same island (via other bridges), don't buy it.

Example Problem: "Min Cost to Connect All Points"

- **Input:** Coordinates of points.
- **Step 1:** Calculate the Manhattan distance between every pair of points (these are your edge weights).
- **Step 2:** Sort all these distances.
- **Step 3:** Run Kruskal's loop.

Comparison: Kruskal's vs. Prim's

Both algorithms solve the exact same problem, but they approach it differently.

Feature	Kruskal's Algorithm	Prim's Algorithm
Approach	**Edge-centric:** Focuses on the edges. Sorts them and picks the best one anywhere in the graph.	**Vertex-centric:** Focuses on growing a single tree from a starting node. Grows like a mold spreading out.
Data Structure	Uses **Union-Find** (Disjoint Set).	Uses a **Priority Queue** (Min-Heap).
Best For	**Sparse Graphs:** (Lots of nodes, fewer edges). It is simpler to implement with an edge list.	**Dense Graphs:** (Lots of edges connecting everything). Since it doesn't need to sort *all* edges upfront, it can be faster here.
Connectivity	Can generate a "forest" (multiple disconnected trees) if the graph is disconnected.	Only generates a single tree from the start node.