Union Find

Union Find data structure: group nodes and check if two things are connected.

- `union(x, y)` → puts x and y in the same group
- `find(x)` → tells us which group x belongs to — or who the “boss” of that group is

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]  # Each node starts as its own parent
        self.count = n  # Number of connected components

    def find(self, x):
        '''
        Find the root of x with path compression
        '''
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression: point x directly
to its root
        return self.parent[x]

    def union(self, x, y):
        '''
        Union the sets that x and y belong to
        '''
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # x and y are already in the same set — union fails (cycle
detected)

        self.parent[rootY] = rootX  # Attach rootY under rootX (or the other way around)
        self.count -= 1
        return True  # Union succeeded

    def connected(self, x, y):
        '''
        Check if x and y are in the same set
        '''
        return self.find(x) == self.find(y)

    def count(self):
        '''
        Return the number of connected components
        '''
```

```
        return self.count
```

What is Union by Rank?

Union by Rank: decide who becomes the parent when merging two sets.

- **Always attach the shorter tree to the taller one**

What Is “Rank”?

In Union Find, rank is a number we use to estimate how “tall” or “large” a tree is.

At the beginning, each node is its own root, so rank = 1.

Why Do We Need This?

We have 7 nodes, and we merge like this:

```
union(1, 2)
union(2, 3)
union(3, 4)
union(4, 5)
union(5, 6)
union(6, 7)
```

And we always attach the new node to the previous one. You’ll end up with a long, skinny chain like this:

```
1   # root in the group
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
           \
            7
```

What Happens With Union by Rank?

```
        4
      /   \
     2     6
    / \   / \
   1   3 5   7
```

How Do We Code It?

Add a `rank[]` array to track the rank of each root. Inside `union()`, we compare ranks:

- If `rank[rootX] < rank[rootY]`, attach X under Y.
- If `rank[rootX] > rank[rootY]`, attach Y under X.
- If equal, pick one as root, and increase its rank by 1.

Here's the code with Union by Rank + Path Compression:

```
class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]
        self.rank = [1] * (n + 1)
        self.count = n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX == rootY:
            return False  # Already in the same group

        # Union by rank
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
```

```
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1

        self.count -= 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

    def count(self):
        return self.count
```

Dijkstra Algorithm

Shortest path problem:

- Single-source shortest path (SSSP) — From *one* starting node to *all* other nodes.
- Multi-source shortest path — From *every* node to *every* other node.

What Problem Does Dijkstra Solve?

- You're given a graph with non-negative edge weights
- You're given a starting node
- You want to find the shortest distance from the start node to every other node

keep in mind:

- Edge weights must be non-negative
- Requires one starting node (solves Single Source Shortest Path)
- Works with both directed and undirected graphs

Core idea

Greedy: Each time starting from the "currently known shortest node", try to update its neighbors; repeat this process until the shortest path is determined for all points.

1. Pick the node that is **closest** to the starting point (among all nodes not yet visited).
2. Tag it as visited.
3. **Update the distances** to its neighboring nodes.

Repeat this process until all nodes have been taged visited with their shortest distances.

Basic Structure

1. Pick the node that is closest to the starting point (among all nodes not yet visited).
2. Tag it as visited.
3. Update the distances to its neighboring nodes.

2 main things:

- keep track of the shortest known distance from the start to each node.

  > use a A `minDist[]` array.
  >
  > Before we start, each element in this array is infinity, and the distance from start to itself is 0.

- mark node as visited if we've visited a node.

  > use a `visited[]` array to track this information.
  >
  > All values start as `False`.

2 important operations:

- How you find the closest node to the starting node?

  > Difference between basic Dijkstra algorithm and heap opetimization Dijkstra algorithm

- How you update the distance to current node's neighboring nodes?

  > how we build the graph?

Naive Dijkstra

1. Find the closest unvisited node — use for loop to find the current shortest point.

2. Mark it as visited.
3. Try to update its neighbors — see if going through this node gives a better path to others.

```
def dijkstra(n, graph, start):
    INF = float('inf')
    dist = [INF] * n          # min distance from start to each node
    dist[start] = 0           # start node's distance to itself is 0

    visited = [False] * n     # whether the shortest path to this node is already finalized

    for _ in range(n):        # Do this n times — we’ll finalize at most n nodes
        u = -1
        # Find the unvisited node with the smallest distance
        for i in range(n):
            if not visited[i] and (u == -1 or dist[i] < dist[u]):
                u = i

        visited[u] = True  # Lock in this node's distance

        # Try to update all neighbors of u
        for v in range(n):
            if graph[u][v] != INF:  # There's an edge from u to v
                dist[v] = min(dist[v], dist[u] + graph[u][v])

    return dist
```

The time complexity here is **O(n²)**.

Dijkstra with Heap Optimization

- Sparse graph: about `O(n)` edges (like a tree or a road map)
- Dense graph: up to `O(n²)` edges (like a complete graph)

Priority queue (a min-heap): A heap can find the smallest value quickly.

1. Find the closest unvisited node — Select the node closest to the source point and add the edge directly to the top heap (using the heap to automatically sort) Every time we take out an edge from the top of the heap, it is naturally the edge of the node closest to the source point..
2. Mark it as visited.
3. Try to update its neighbors.

```
import heapq
from collections import defaultdict
```

```python
def dijkstra(n, graph, start):
    """
    Heap-optimized Dijkstra algorithm.
    :param n: number of nodes (1-indexed)
    :param graph: adjacency list -> graph[u] = [(v, weight), ...]
    :param start: starting node
    :return: dist[i] = shortest distance from start to node i
    """
    dist = [float('inf')] * (n + 1)
    dist[start] = 0

    visited = [False] * (n + 1)
    heap = [(0, start)]  # (distance, node)

    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True

        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(heap, (dist[v], v))

    return dist
```

Build graph

There are two common ways to represent graphs: **Adjacency Matrix** and **Adjacency List**.

Structure	When to Use	Works Well With...	Time Complexity
Adjacency List	Sparse graphs	Heap-optimized Dijkstra	`O(m log n)`
Adjacency Matrix	Dense graphs	Naive Dijkstra	`O(n²)`

Adjacency matrix

```python
graph = [[float('inf')] * (n) for _ in range(n)]

for u, v, w in times:
    graph[u][v] = w
```

Adjacency List

```python
from collections import defaultdict

graph = defaultdict(list)

for u, v, w in times:
    graph[u].append((v, w))
```