MTH 4300/4299

More on graphs: Union find, topological sort

Jaime Abbariao

What is Union Find?

- Data structure for tracking disjoint sets
- Efficiently performs two main operations:
  - **Find**: Determine which set an element belongs to
  - **Union**: Merge two sets together
- Also known as Disjoint Set Union (DSU)

Key Operations

1. **Find(x)**: Returns representative of set containing x
2. **Union(x, y)**: Merges sets containing x and y
3. **Connected(x, y)**: Checks if x and y are in same set

Basic Implementation

```
class UnionFind {
private:
    std::vector<int> parent;
    std::vector<int> rank;
public:
    UnionFind(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;  // Each element is its own parent
initially
        }
    }
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // Path compression
        }
        return parent[x];
    }
```

Union Implementation

```
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};
```

Connected implementation

```
bool connected(int x, int y) {
    return find(x) == find(y);
}
```

Path Compression Visualization

Before Path Compression:

Chain: $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4$

Height: $O(n)$

After Path Compression:

All point to root: $0 \longleftarrow 1,2,3,4$

Height: $O(1)$

Union by Rank Example

Initial State:

- Set A: $\{0\}$
- Set B: $\{1, 2\}$

Ranks: A=0, B=1

Union Operation:

- Attach smaller rank to larger
- A becomes child of B

Result: $\{0, 1, 2\}$

Final Structure:

- Root: 1
- Children: 0, 2

Balanced tree

Applications Example: Network Connectivity

```
// Check if network components are connected
UnionFind network(6);  // 6 computers

// Add connections
network.unite(0, 1);  // Computer 0 ↔ 1
network.unite(1, 2);  // Computer 1 ↔ 2
network.unite(3, 4);  // Computer 3 ↔ 4

// Check connectivity
bool connected_0_2 = network.connected(0, 2);  // true
bool connected_0_3 = network.connected(0, 3);  // false

// Bridge networks
network.unite(2, 3);  // Connect the two components
bool now_connected = network.connected(0, 4);  // true
```

Time Complexity

Why $\alpha(n)$ (Inverse Ackermann)?

- Path compression + union by rank create nearly flat trees
- Without optimizations: $O(n)$ worst case (linear chain)
- With optimizations: $\alpha(n) \leq 5$ for all practical input sizes
- Therefore: effectively $O(1)$ constant time
- **Find**: $O(\alpha(n)) \approx O(1)$ with path compression
- **Union**: $O(\alpha(n)) \approx O(1)$ with union by rank
- **Space**: $O(n)$

What is Topological Sort?

- Linear ordering of vertices in directed acyclic graph (DAG)
- For every directed edge u $\longrightarrow$ v, u appears before v in ordering
- Used for: dependency resolution, task scheduling, course prerequisites

Prerequisites

- Graph must be a DAG (no cycles)
- If cycle exists, topological sort is impossible

Algorithm: Kahn's Algorithm (BFS-based)

1. Calculate in-degree for each vertex
2. Start with vertices having in-degree 0
3. Process vertices one by one:
   - Remove vertex from graph
   - Decrease in-degree of all neighbors
   - Add neighbors with in-degree 0 to queue

Implementation: Setup Phase

```
std::vector<int> topologicalSort(int numVertices,
                                 std::vector<std::vector<int>>&
edges) {
    // Build adjacency list and calculate in-degrees
    std::vector<std::vector<int>> adj(numVertices);
    std::vector<int> inDegree(numVertices, 0);

    for (auto& edge : edges) {
        int from = edge[0], to = edge[1];
        adj[from].push_back(to);
        inDegree[to]++;
    }

    // Find all vertices with in-degree 0
    std::queue<int> queue;
    for (int i = 0; i < numVertices; i++) {
        if (inDegree[i] == 0) {
```

Implementation: Setup Phase (ii)

```
            queue.push(i);
        }
    }
```

Implementation: Processing Phase

```
    std::vector<int> result;

    // Process vertices
    while (!queue.empty()) {
        int vertex = queue.front();
        queue.pop();
        result.push_back(vertex);

        // Reduce in-degree of neighbors
        for (int neighbor : adj[vertex]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                queue.push(neighbor);
            }
        }
    }
```

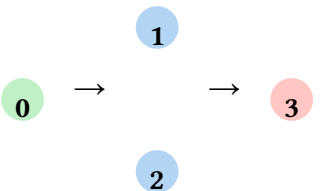
Implementation: Processing Phase (ii)

```
    // Check if all vertices were processed (no cycles)
    if (result.size() != numVertices) {
        return {}; // Cycle detected
    }

    return result;
}
```

Visual Example: Build Dependencies

Dependency Graph:



Dependencies: $0 \longrightarrow 1$, $0 \longrightarrow 2$, $1 \longrightarrow 3$, $2 \longrightarrow 3$

Topological Order: $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3$ **or** $0 \longrightarrow 2 \longrightarrow 1 \longrightarrow 3$

Kahn's Algorithm Steps

Step 1: In-degrees

- Node 0: 0
- Node 1: 1
- Node 2: 1
- Node 3: 2

Queue: [0]

Step 2: Process 0

- Remove 0
- Update: 1⟶0, 2⟶0

Queue: [1, 2] Result: [0]

Step 3: Process 1, 2

- Remove 1, 2
- Update: 3⟶0

Queue: [3] Result: [0, 1, 2]

Example: Software Build Dependencies

```
// Dependencies: 0->1, 0->2, 1->3, 2->3
// Meaning: task 0 must complete before 1 and 2
//          tasks 1 and 2 must complete before 3
std::vector<std::vector<int>> dependencies = {
    {0, 1}, {0, 2}, {1, 3}, {2, 3}
};

auto order = topologicalSort(4, dependencies);
// Possible result: [0, 1, 2, 3] or [0, 2, 1, 3]
```

Applications

- **Course scheduling**: Prerequisites define dependencies
- **Build systems**: Compile dependencies
- **Project management**: Task dependencies
- **Package managers**: Installation order