Competitive
Programming and
Mathematics
Society

Disjoint Set Unions

Uses of Union-Find

CPMSoc x Citadel

Looking Forward To

1. **Gentle Intro to Disjoint Set Unions**
   - Motivation - Application to Kruskal's
   - Implentation: Naive and Improved
   - Roads Not Only in Berland
2. **Citadel Takeover**
   - Who We Are
   - Fireside Chat
   - Citadel's Favourite DSU Questions
3. **Networking!**

Citadel Wants To Know You!

Motivation: Kruskal’s Algorithm

- Recap: Kruskal’s Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.

Motivation: Kruskal's Algorithm

- Recap: Kruskal's Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.
    - **Problem**: How can we efficiently determine if adding an edge (a, b) creates a cycle?

Motivation: Kruskal's Algorithm

- Recap: Kruskal's Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.
    - **Problem**: How can we efficiently determine if adding an edge (a, b) creates a cycle?
    - We could detect whether there is a path from a to b using a regular DFS, but this would be a linear overhead $O(V + E)$ for *every* edge we consider, which will **blow up** our time complexity to $O(E(V + E))$.

Motivation: Kruskal's Algorithm

- Recap: Kruskal's Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.
  - **Problem**: How can we efficiently determine if adding an edge (a, b) creates a cycle?
  - We could detect whether there is a path from a to b using a regular DFS, but this would be a linear overhead $O(V + E)$ for *every* edge we consider, which will **blow up** our time complexity to $O(E(V + E))$.
- We want a data structure to record which connected component each vertex is part of as we build our minimum spanning tree
  - **Key Result:** Adding an edge (a, b) is valid if and only if a and b are in different connected components.

Motivation: Kruskal's Algorithm

- Recap: Kruskal's Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.
  - **Problem**: How can we efficiently determine if adding an edge (a, b) creates a cycle?
  - We could detect whether there is a path from a to b using a regular DFS, but this would be a linear overhead $O(V + E)$ for *every* edge we consider, which will **blow up** our time complexity to $O(E(V + E))$.
- We want a data structure to record which connected component each vertex is part of as we build our minimum spanning tree
  - **Key Result:** Adding an edge (a, b) is valid if and only if a and b are in different connected components.
    - If node a and b are already in the same connected component, there must already be a path from a to b and hence, adding the edge (a, b) will create two distinct paths from a to b which creates a cycle.

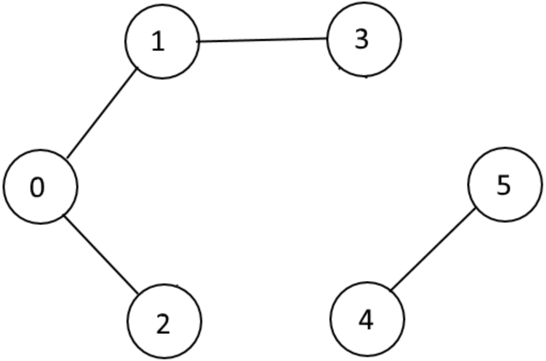
Motivation: Kruskal’s Algorithm

- Recap: Kruskal’s Algorithm finds the minimum spanning tree of a connected, undirected graph by starting with the empty graph (without edges) and iteratively adding the shortest (yet to be added) edge that does not create a cycle.
    - **Problem**: How can we efficiently determine if adding an edge (a, b) creates a cycle?
    - We could detect whether there is a path from a to b using a regular DFS, but this would be a linear overhead $O(V + E)$ for *every* edge we consider, which will **blow up** our time complexity to $O(E(V + E))$.
- We want a data structure to record which connected component each vertex is part of as we build our minimum spanning tree
    - **Key Result:** Adding an edge (a, b) is valid if and only if a and b are in different connected components.
        - If node a and b are already in the same connected component, there must already be a path from a to b and hence, adding the edge (a, b) will create two distinct paths from a to b which creates a cycle.
    - If we do add the edge (a, b), we are combining their two distinct connected components into one, which we need way of recording.

Solution - Disjoint Sets

- A disjoint set refers to a collection of sets in which every pair of sets has no common elements.
- For example, the sets $\{1, 2, 3\}$ and $\{4, 5, 6\}$ are disjoint because they do not share any elements. However, the sets $\{1, 2, 3\}$ and $\{3, 4, 5\}$ are not disjoint because they have the common element 3.
- DSU will provide two main functions that the user can use
  - $find\_set$, which lets the user know which connected component a vertex belongs to.
  - $union\_set$, which connects two **disjoint** connected components, analogous to adding an edge between the two connected components.

Visualisation



Disjoint Sets

Set Id: 0, elements: [0, 1, 2, 3]

Set Id: 4, elements: [4, 5]

DSU's application in Kruskal's

We will use disjoint sets to represent connected components in our graph.

DSU's application in Kruskal's

We will use disjoint sets to represent connected components in our graph.

1. We initialise every vertex to be its own connected component, which is clearly the case when no edges have yet been added.

DSU's application in Kruskal's

We will use disjoint sets to represent connected components in our graph.

1. We initialise every vertex to be its own connected component, which is clearly the case when no edges have yet been added.
2. When considering whether to add an edge (a, b), we use our disjoint sets to determine whether a and b are already in the same connected component.
    - ie. if $find\_set(a) == find\_set(b)$, then a and b are already in the same connected component so the edge (a, b) should not be added as it will create a cycle.
    - Otherwise, the edge (a, b) can be added

DSU's application in Kruskal's

We will use disjoint sets to represent connected components in our graph.

1. We initialise every vertex to be its own connected component, which is clearly the case when no edges have yet been added.
2. When considering whether to add an edge (a, b), we use our disjoint sets to determine whether a and b are already in the same connected component.
    - ie. if $find\_set(a) == find\_set(b)$, then a and b are already in the same connected component so the edge (a, b) should not be added as it will create a cycle.
    - Otherwise, the edge (a, b) can be added
3. If we find it's valid to add edge (a, b), then we need to merge the two connected components of a and b.
    - More specifically, we will use the union operation $union\_set(a, b)$ (more on this very soon!)

DSU's application in Kruskal's

We will use disjoint sets to represent connected components in our graph.

1. We initialise every vertex to be its own connected component, which is clearly the case when no edges have yet been added.
2. When considering whether to add an edge (a, b), we use our disjoint sets to determine whether a and b are already in the same connected component.
   - ie. if $find\_set(a) == find\_set(b)$, then a and b are already in the same connected component so the edge (a, b) should not be added as it will create a cycle.
   - Otherwise, the edge (a, b) can be added
3. If we find it's valid to add edge (a, b), then we need to merge the two connected components of a and b.
   - More specifically, we will use the union operation $union\_set(a, b)$ (more on this very soon!)
4. We keep repeating steps **2** and **3** until we've added $|V| - 1$ edges.

Implementation

```
void kruskals() {
        sort(edgelist.begin(), edgelist.end());
        // initialise each vertex to be in its own connected component
        initialise_set();
        vector<pair<int, int>> tree_edges;
        for (auto edge : edgelist) {
            int u = edge.v1; int v = edge.v2;
            if (find_set(u) != find_set(v)) { // check if u, v belong to the
                                              // same connected component.
                // if so, merge their two connected components.
                union_set(u, v);
                tree_edges.push_back({u, v});
            }
        }
    }
}
```

Naive Implementation

Let's think about how we could implement a DSU data structure!

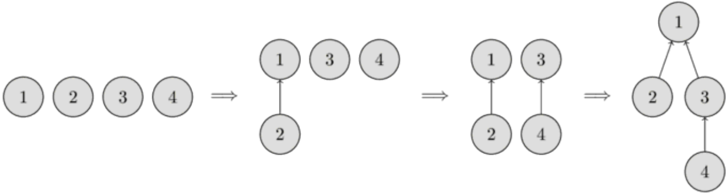
We want to be able to support the following functions:

```
// Creates a new set containing v
void make_set(int v);

// Returns the id (or representative) of the set containing v
int find_set(int v);

// Merge the sets containing a and b
void union_sets(int a, int b);
```

Naive Implementation

We can represent each of the sets as trees, with the root of the tree being the "representative" of the set.

Thus, we can store this tree in an array called parent that stores a reference to each node's immediate ancestor in the tree.



Naive Implementation

This leads us to the following implementation:

```
void make_set(int v) {
    parent[v] = v;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
}
```

Naive Implementation

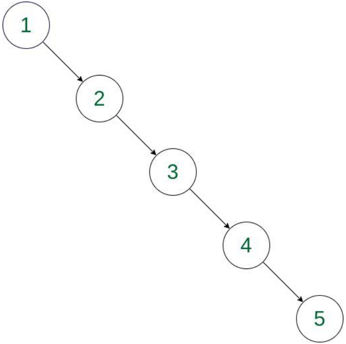
This is a good start, but it is too inefficient.

- make_set() runs in $O(1)$ time.
- union_sets() runs in $O(1)$ time too.
- find_set() actually runs in $O(n)$ time in the worst case. Can we think of an example where this might happen?

Naive Implementation

This is a good start, but it is too inefficient.

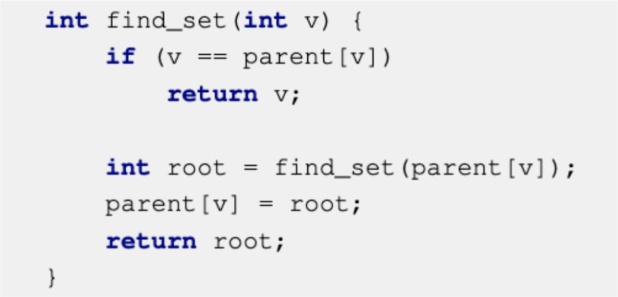
- make_set() runs in $O(1)$ time.
- union_sets() runs in $O(1)$ time too.
- find_set() actually runs in $O(n)$ time in the worst case. Consider the following:



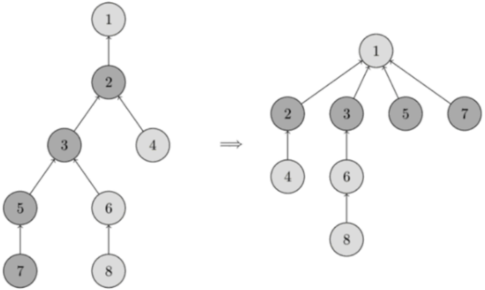
We can make find_set() more efficient with some optimisations.

Optimization 1: Path Compression

- Path compression helps speed up the find_set() operation
- Instead of traversing the whole depth of the DSU tree every time(which may become quite deep!), we link the elements straight to the representative(root) node p
- This way we only traverse a deep branch once, and while doing so link the elements directly to the root. Next time we find_set() it will be a lot faster!

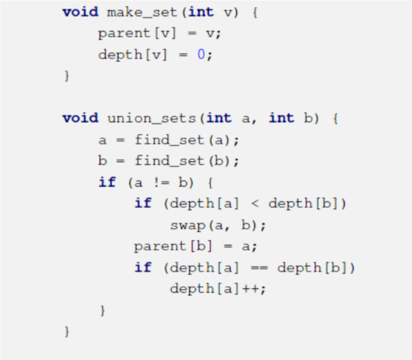


```
int find_set(int v) {
    if (v == parent[v])
        return v;

    int root = find_set(parent[v]);
    parent[v] = root;
    return root;
}
```

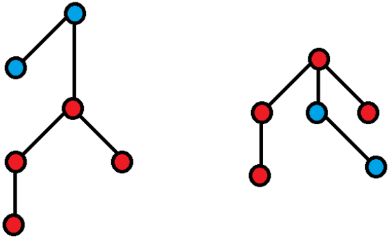


Optimization 2: Union by Height

- Union by size ensures when we merge two sets, we can get branches that are as short as possible
- The Idea is to link shallow trees to deep trees
- Example below, we compare the height of the red set and blue set. Then merge the shallower set onto the deeper set which results in tree with less max depth.



```
void make_set(int v) {
    parent[v] = v;
    depth[v] = 0;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (depth[a] < depth[b])
            swap(a, b);
        parent[b] = a;
        if (depth[a] == depth[b])
            depth[a]++;
    }
}
```



Question Time

Berland Government decided to improve relations with neighbouring countries. First of all, it was decided to build new roads so that from each city of Berland and neighbouring countries it became possible to reach all the others. There are n cities in Berland and neighbouring countries in total and exactly $n - 1$ two-way roads. Because of the recent financial crisis, the Berland Government is strongly pressed for money, so to build a new road it has to close some of the existing ones. Every day it is possible to close one existing road and immediately build a new one. Your task is to determine how many days would be needed to rebuild roads so that from each city it became possible to reach all the others and to draw a plan for the closure of old roads and building of new ones.

Input:

4

1 2

2 3

1 3

Output: 1

Observations

- Which roads are the most **optimal** roads to disconnect?

Observations

- Which roads are the most **optimal** roads to disconnect?
  - For a cycle, we can remove ANY edge on that cycle without disconnecting any cities from each other. Hence, we should only close roads that is part of a cycle.

Observations

- Which roads are the most **optimal** roads to disconnect?
    - For a cycle, we can remove ANY edge on that cycle without disconnecting any cities from each other. Hence, we should only close roads that is part of a cycle.
- Do we have to remove all cycles from the graph?

Observations

- Which roads are the most **optimal** roads to disconnect?
  - For a cycle, we can remove ANY edge on that cycle without disconnecting any cities from each other. Hence, we should only close roads that is part of a cycle.
- Do we have to remove all cycles from the graph?
  - **Yes!** Remember the final graph only has $n - 1$ edges which have to connect n nodes. This forms a tree and by the tree definition, the graph must have **no** cycle. So we must remove all cycles from this graph.

Observations

- Which roads are the most **optimal** roads to disconnect?
  - For a cycle, we can remove ANY edge on that cycle without disconnecting any cities from each other. Hence, we should only close roads that is part of a cycle.
- Do we have to remove all cycles from the graph?
  - **Yes!** Remember the final graph only has $n - 1$ edges which have to connect n nodes. This forms a tree and by the tree definition, the graph must have **no** cycle. So we must remove all cycles from this graph.
  - This means we must close all roads so that the resulting graph is now cycle-less.

Observations

- Can we now always add the edges back so that all the countries are connected?

Observations

- Can we now always add the edges back so that all the countries are connected?
    - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,

Observations

- Can we now always add the edges back so that all the countries are connected?
  - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?

Observations

- Can we now always add the edges back so that all the countries are connected?
    - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?

Observations

- Can we now always add the edges back so that all the countries are connected?
  - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?
  - DSU

Observations

- Can we now always add the edges back so that all the countries are connected?
  - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?
  - DSU
  - If the current edge in the input does not create a cycle, we simply add that edge into our DSU by running the union set command to join the two vertices together.

Observations

- Can we now always add the edges back so that all the countries are connected?
  - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?
  - DSU
  - If the current edge in the input does not create a cycle, we simply add that edge into our DSU by running the union set command to join the two vertices together.
  - If the current edge in the input does create a cycle, we do not add that edge in (since it will create a cycle), and then mark that edge as one which we will close from our existing road network.

Observations

- Can we now always add the edges back so that all the countries are connected?
    - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?
    - DSU
    - If the current edge in the input does not create a cycle, we simply add that edge into our DSU by running the union set command to join the two vertices together.
    - If the current edge in the input does create a cycle, we do not add that edge in (since it will create a cycle), and then mark that edge as one which we will close from our existing road network.
- How do we now figure out which edges to add in such that all countries will be connected and that there will not exist a cycle?

Observations

- Can we now always add the edges back so that all the countries are connected?
  - Yep! We are always guaranteed that there exists a way to add the roads back in such that the remaining graph is connected. With every road, we can connect two different connected components and in the end,
- Which data structure can we first use to determine which edges in our existing graph will create a cycle?
  - DSU
  - If the current edge in the input does not create a cycle, we simply add that edge into our DSU by running the union set command to join the two vertices together.
  - If the current edge in the input does create a cycle, we do not add that edge in (since it will create a cycle), and then mark that edge as one which we will close from our existing road network.
- How do we now figure out which edges to add in such that all countries will be connected and that there will not exist a cycle?
  - We loop through every possible road ($\{1 \rightarrow 2, 1 \rightarrow 3, 1 \rightarrow 4, \ldots, 1 \rightarrow n, 2 \rightarrow 3, 2 \rightarrow 4, \ldots\}$), if adding in the current road does not join two separate components together, we skip over the road as it will create a cycle.

Time Complexity

- What is the time complexity of our current algorithm?

Time Complexity

- What is the time complexity of our current algorithm?
- $O(n^2 \log n)$ since the dominating factor comes from the part where we test out to see which edge should be added back in.

Time Complexity

- What is the time complexity of our current algorithm?
- $O(n^2 \log n)$ since the dominating factor comes from the part where we test out to see which edge should be added back in.
- Can we optimise our solution to $O(n \log n)$? (we somehow can optimise the part where we are adding the edges back in...).

Time Complexity

- We actually only need to check the edges $1 \to 2, 1 \to 3, 1 \to 4, \ldots, 1 \to n$ and try to add each one into the graph, and that will be enough to guarantee we will have one connected road network at the end.

Time Complexity

- We actually only need to check the edges $1 \to 2, 1 \to 3, 1 \to 4, \ldots, 1 \to n$ and try to add each one into the graph, and that will be enough to guarantee we will have one connected road network at the end.
- We can prove this by contradiction, assume that this does not form a single connected component, as in, there will be multiple connected components after we try to add all the edges. Then, there must be a connected component with country 1, and then another one without country 1, but in that case, those two would have been added to the same connected component when we iterated through all the edges. Thus, every country must be in the same connected component after we loop through N edges.

Time Complexity

- We actually only need to check the edges $1 \to 2, 1 \to 3, 1 \to 4, \ldots, 1 \to n$ and try to add each one into the graph, and that will be enough to guarantee we will have one connected road network at the end.
- We can prove this by contradiction, assume that this does not form a single connected component, as in, there will be multiple connected components after we try to add all the edges. Then, there must be a connected component with country 1, and then another one without country 1, but in that case, those two would have been added to the same connected component when we iterated through all the edges. Thus, every country must be in the same connected component after we loop through N edges.
- This reduces our time complexity to $O(n \log n)$.

Time Complexity

- We actually only need to check the edges $1 \to 2, 1 \to 3, 1 \to 4, \ldots, 1 \to n$ and try to add each one into the graph, and that will be enough to guarantee we will have one connected road network at the end.
- We can prove this by contradiction, assume that this does not form a single connected component, as in, there will be multiple connected components after we try to add all the edges. Then, there must be a connected component with country 1, and then another one without country 1, but in that case, those two would have been added to the same connected component when we iterated through all the edges. Thus, every country must be in the same connected component after we loop through N edges.
- This reduces our time complexity to $O(n \log n)$.
- https://codeforces.com/problemset/problem/25/D

Arc Attendance :D

Further events

Please join us for:

- COMP3121/3821 Revision Session (Thursday Week 10)