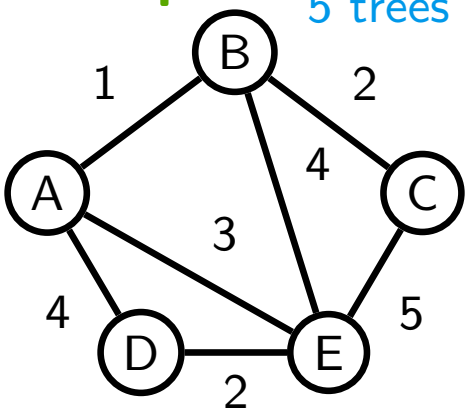
Minimal Spanning Forests and Kruskal's Algorithm

Example: Execution of Kruskal's algorithm

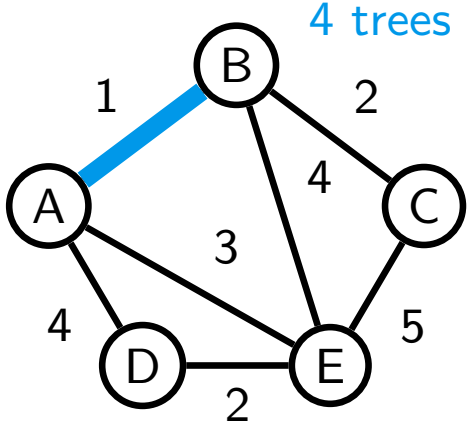
Idea: Starting with a forest where every node in the graph is a separate tree, greedily add edges with minimum weights such that each edge is between different trees.

Example

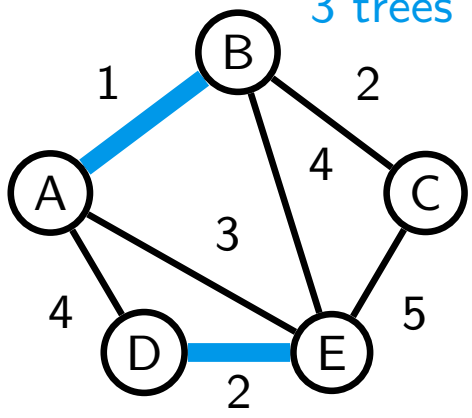
5 trees



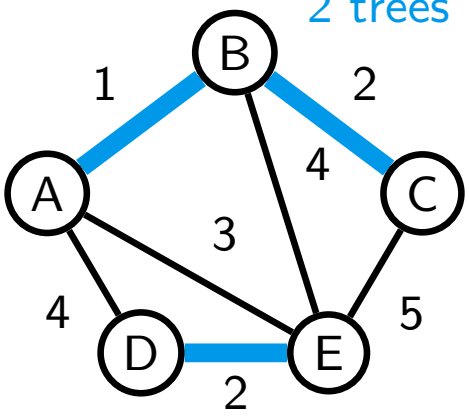
4 trees



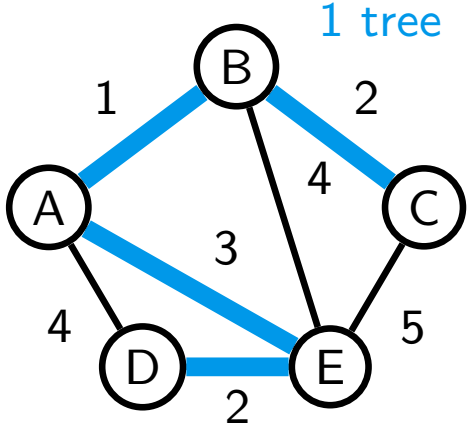
3 trees



2 trees



1 tree



The minimal spanning tree consists of edges: AB, BC, AE, DE.

- A *Forest* is a set of trees
- If the graph is *connected* then this creates a minimal spanning tree. Otherwise it creates a *minimal spanning forest*: a set of minimal spanning trees, one for each connected component of the graph
- The algorithm is greedy edge-based because it always chooses the best (i.e. minimal) edge to add at each step
- It finishes when no more edges can be added
  - Note: unless the graph is connected, this is **NOT** the same as finishing when there is only one tree
- Good performance is dependent on having an efficient way to
  1. Discover if an edge is between nodes in the same tree
  2. If an edge is between 2 different trees, which 2 trees these are

  For this purpose, we can use a **Union-Find** data structure

Union-Find

The *Union-Find* ADT has 3 operations:

- `makeSet(x)`: Make a new set containing the element x
- `find(x)`: Find the set that x is an element of. It returns a particular element of the set that is used to identify the set
- `union(x,y)`: merge the two sets that contain elements x and y. This replaces the two pre-existing sets containing x and y by a single new set which contains the union of the two sets

In Kruskal: each tree is represented by the set of nodes in the tree:

- `makeSet(n)` is called on each node of the graph at the start
- To test if an edge (u,v) is between nodes in the same or different trees, call `find(u)` and `find(v)` and compare their results.
- When an edge (u,v) is added, `merge(find(u),find(v))` is called to merge the two trees joined by the edge

Union-Find Implementation

There are pointer based implementations, but we will use a simple array based implementation, where we assume that the graph nodes are numbered $0, 1, \ldots, n-1$.

Sets will be represented by trees, where a `parent` array, indexed by the node number, identifies the parent of each node in the set, and the root of the tree is the set representative that identifies the set. The root of each tree has itself as its own parent

To find the set that a node is in, we recursively follow the chain of parents up to the root of the tree

To merge two sets, you find the roots of the two trees and set one to have the other as its parent

A good explanation of this structure can be found at: `https://cp-algorithms.com/data_structures/disjoint_set_union.html`. The code in these slides is based on that.

Union-Find Naïve Implementation

```
1  void makeSet(int v) {
2      parent[v] = v;
3  }
4
5  int find(int v) {
6      if (v == parent[v])
7          return v;
8      return find(parent[v]);
9  }
10
11 void union(int a, int b) {
12     a = find(a);
13     b = find(b);
14     if (a != b)
15         parent[b] = a;
16 }
```

Naïve Union-Find Problem 1

Unions can end up making very deep chains, i.e. the trees that represent sets can end up very deep and narrow, or even linear. This makes `find(x)` require a nearly linear search up a long chain to find the root.

This can be fixed by changing `find(x)` so that, on return from the recursive calls to find the root, it sets the parent of all nodes on the path just traversed to be the root, thus reducing the `find(x)` cost for all future calls on any of those nodes

Naïve Union-Find Problem 2

The previous code for `union(u, v)` just makes the second tree be a child of the root of the first tree. If the second tree is deeper than the first, than this makes the resulting tree deeper still, again reducing performance of `find(x)`

Here we just need to choose the deeper tree to be the one that we add the shallower tree to. Actually, it works out that you can alternatively choose to make the smaller tree the child of the deeper one, and in practice you get the same improved performance

In either case, you need an extra array to record either (an approximation to) the rank (level) of the tree rooted at each node, or the size of the subtree rooted at each node.

One subtlety is that it is not necessary to maintain the depth/size of the subtree at *every* node, just that of the *root* nodes.

Improved Union-Find (based on size)

```
1  void makeSet(int v) {
2      parent[v] = v;
3      size[v] = 1;
4  }
5
6  int find(int v) {
7      if (v == parent[v])
8          return v;
9      return parent[v] = find(parent[v]);
10 }
11
12 void union(int a, int b) {
13     a = find(a);
14     b = find(b);
15     if (a != b) {
16         if (size[a] < size[b])
17             swap(a, b);
18         parent[b] = a;
19         size[a] += size[b]
20     }
21 }
```

The improved algorthm that is based on size does not need to do anything extra in the `find(x)` method as even the updates done there to shorten the paths from a node to its root still leaves the node in the same tree and hence does not change the size of the tree.

Improved Union-Find (based on rank)

```
1  void makeSet(int v) {
2      parent[v] = v;
3      rank[v] = 0;
4  }
5
6  int find(int v) {
7      if (v == parent[v])
8          return v;
9      return parent[v] = find(parent[v]);
10 }
11
12 void union(int a, int b) {
13     a = find(a);
14     b = find(b);
15     if (a != b) {
16         if (rank[a] < rank[b])
17             swap(a, b);
18         parent[b] = a;
19         if (rank[a] == rank[b])
20             rank[a]++
21     }
22 }
```

The improved alorithm that is based on rank can not efficiently reduce the rank of the root correctly during a call to $find(x)$ that reduces path lengths because that would require paths of from all leaves to the root and not just the one that has been traversed.

However, it is sufficient to use the heuristic of tracking an upperbound of the depth, where no change to the rank is made during a call to `find(x)`.

Union-Find performance

Without the two optimisations, the trees involved can have height O(n), so that `find(x)` and `union(x, y)` both have O(n) complexity

With both the above optimisations the amortised (over many operations) complexity of each operation will be $\Theta(\alpha(n))$

$\alpha(n)$ is the reverse Ackermann function which is extremely slow growing and will not exceed 4 for any realistic value of n.

Hence the amortised upperbound complexity for all operations is effectively $O(1)$

Kruskal's Algorithm

Now that we have an efficient Union-Find ADT, we can use it in Kruskal's algorithm:

```
1  let result be a new empty list of edges
2  for each node n in G
3    makeSet(n)
4  let E be a list of edges in G sorted by increasing weights
5  for each edge e = (u,v) in E in order
6  {
7    if (find(u) != find(v))
8    {
9      result.add(e)
10     union(find(u), find(v))
11   }
12 }
13 return result
```

Kruskal's Algorithm Complexity

This algorithm requires sorting the graph edges by weight, which is $O(e \log e)$, where e is the number of edges.

Then it does a linear search through the edges, during which it carries out only operations of $O(1)$. This contributes $O(e)$

Hence its final complexity is $O(e \log e + e) = O(e \log e)$

e is at most n^2 where n is the number of nodes in the graph.

So $O(e \log e) = O(e \log n^2) = O(2e \log n) = O(e \log n)$

Kruskal's Algorithm vs Jarník-Prim algorithm

If the graph is sparse, i.e. $e \approx n$, then

- Fib Adj List Jarník-Prim: $O(n \log n + n) = O(n \log n)$
- Adj Matrix Jarník-Prim: $O(n^2)$
- Kruskal: $O(n \log n)$

If the graph is dense, i.e. $e \approx n^2$, then

- Fib Adj List Jarník-Prim: $O(n \log n + n^2) = O(n^2)$
- Adj Matrix Jarník-Prim: $O(n^2)$
- Kruskal: $O(n^2 \log n)$

Finally, note that Kruskal's algorithm computes a spanning forest, if the graph is not connected, or a spanning tree if it is connected, while Jarník-Prim can only calculate a spanning tree on a connected graph.