# 15. Components and spanning trees

The graph algorithms we have seen so far compute their results by visiting all nodes and edges of the graph. Such algorithms are slow, however, when the graph keeps changing frequently.

In this chapter, we will learn about the union-find data structure that can maintain the information about the components of a graph as edges are added to the graph. For example, we can efficiently find if two nodes are in the same component, or how many components the graph has.

We will also see Kruskal's algorithm that uses the union-find data structure for computing a minimum spanning tree. A *spanning tree* of a graph is a subset of the edges that connects all nodes.

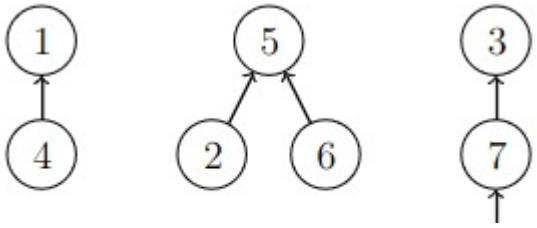
## Union-find data structure

A *union-find data structure* maintains a collections of elements divided into subsets. Initially each element is alone in its own set, and then the sets can be repeatedly merged into bigger sets. The data structure supports two efficient operations:

- `find`: Find which set contains a given element
- `union`: Merge two sets into a single set

A union-find data structure is implemented so that one element in each set represents the whole set. All other elements in a set contain a reference to the representative element either directly, or indirectly through other elements in the set. By following the references, we can find the representative element of the set containing a given element.

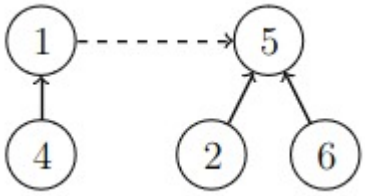
As an example, consider a union-find structure for the elements $1, 2, \ldots, 8$. In the following figure, the sets are $\{1, 4\}$, $\{2, 5, 6\}$ and $\{3, 7, 8\}$:




Here the representatives of the sets are $1$, $5$ and $3$. For any other element, the chain of references leads to one of the representatives. For example, the path from the element $2$ to the representative of its set is $2 \to 5$, and the path from the element $8$ to the representative of its set is $8 \to 7 \to 3$.

Two elements belong to the same set if they have the same representative. For example, the elements $2$ and $6$ are in the same set, because the representive of both is $5$. Conversely, the elements $2$ and $3$ belong to different sets, because the representative of the element $2$ is $5$ and the representative of the element $3$ is $3$.

When two sets are merged, the reference from one of their representatives is set to point to the other representative, which then becomes the representative of the new set. For example, the following figure shows how the sets $\{1, 4\}$ and $\{2, 5, 6\}$ can be merged:




Here the representative $1$ of the set $\{1, 4\}$ is set to reference the element $5$, which is the representative of the set $\{2, 5, 6\}$. Afterwards, there is a new set $\{1, 2, 4, 5, 6\}$ with the representative $5$, and the old sets $\{1, 4\}$ and $\{2, 5, 6\}$ exist no more. For example, the path from the element $4$ to its representative is now $4 \to 1 \to 5$.

The efficiency of the union-find data structure depends on how fast the representative of a given element can be found. The shorter the reference paths are, the faster the representatives can be found. The paths can be kept short by implementing the merging in a specific way.

When two sets are merged, there are two choices for the representative of the new set. For efficiency, the better choice is the representative of the *bigger* of the old sets. This ensures that the length of every path is $O(\log n)$, and thus the representative of any element can be found efficiently.

The union-find data structure can be implemented as follows:

```
class UnionFind:
```

```
    def __init__(self, nodes):
        self.link = {node: None for node in nodes}
        self.size = {node: 1 for node in nodes}

    def find(self, x):
        while self.link[x]:
            x = self.link[x]
        return x

    def union(self, a, b):
        a = self.find(a)
        b = self.find(b)
        if a == b: return

        if self.size[a] > self.size[b]:
            a, b = b, a
        self.link[a] = b
        self.size[b] += self.size[a]
```

The dictionary `link` stores the references from the elements. The reference of a representative element has the value `None`. The dictionary `size` stores the size of the set for each representative element.

The method `find` follows the chain of references until it reaches a representative element. The method `union` merges the sets containing the elements `a` and `b`. First, the method finds the representatives of the sets. If the elements are already in the same set, the method does nothing. Otherwise, the method sets the reference from the representative of the smaller set to the representative of the bigger set, and updates the size for the representative of the new set.

The following code demonstrates the use of the class:

```
u = UnionFind([1, 2, 3, 4, 5, 6, 7, 8])

u.union(1, 4)
u.union(2, 5)
u.union(5, 6)
u.union(3, 7)
u.union(7, 8)

print(u.find(1)) # 4
print(u.find(2)) # 5
print(u.find(3)) # 7
```

```
print(u.find(4)) # 4
print(u.find(5)) # 5
print(u.find(6)) # 5
print(u.find(7)) # 7
print(u.find(8)) # 7
```

Here the representative of the set $\{1, 4\}$ is $4$, the representative of the set $\{2, 5, 6\}$ is $5$ and the representative of the set $\{3, 7, 8\}$ is $7$.

## Example: New roads

**Task**

Bitland has $n$ cities but initially no roads. Your task is to design a class `NewRoads` with the following methods:

- `add_road`: adds a road between two cities
- `has route`: checks if there is two cities are connected by roads

Both methods should be efficient.

The task can be solved efficiently using the union-find data structure as follows:

```
class NewRoads:
    def __init__(self, n):
        self.uf = UnionFind(range(1, n + 1))

    def add_road(self, a, b):
        self.uf.union(a, b)

    def has_route(self, a, b):
        return self.uf.find(a) == self.uf.find(b)
```

The idea is that the sets of the union-find data structure correspond to the components of the road network. Initially, each element is in its own set of size one, which means that each node is in its own component of size one.

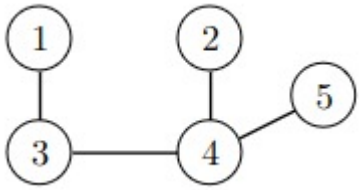
The method `add_road` calls the method `union` that merges the components of the graph. The method `has_route` calls the method `find` for each element. The elements belong to the same component if the method `find` returns the same

representative for both elements.

In this solution, the time complexity of both the method `add_road` and the method `has_route` is $O(\log n)$.

# Trees in graphs

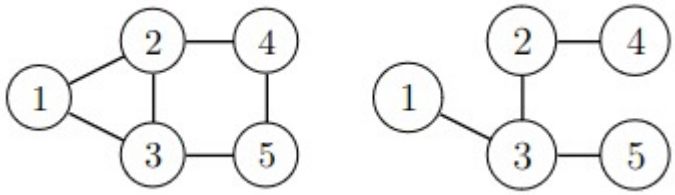
An undirected graph is a *tree* if the graph is connected and acyclic. For example, the following graph is a tree:




Unlike the trees we have seen previously on this course, this kind of a tree has no root, and the nodes have no children or parents. However, the tree has leaves: The leaves are the nodes with exactly one connecting edge. For example, the leaves of the above tree are $1$, $2$ and $5$.

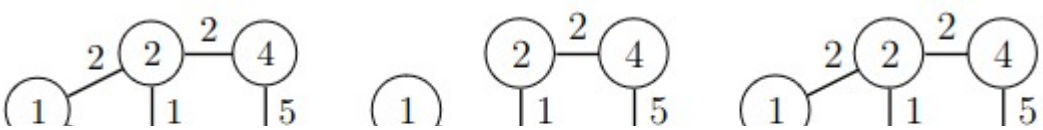
When a graph of $n$ nodes is a tree, it has exactly $n - 1$ edges. If any edge is removed from a tree, it is no longer connected. If a new edge is added to a tree, it is no longer acyclic.

A *spanning tree* of a graph is a tree that contains all the nodes of the graph and some subset of the edges. The following figure shows a graph on the left, and one of the spanning trees of the graph on the right:




Typically a graph has multiple different spanning trees, because there are multiple ways of choosing the edges so that the result is a tree.

When the graph is weighted, the weight of a spanning tree is computed as the sum of the weights of its edged. As an example, the following figure shows a weighted graph and two of its spanning trees:




The weight of the first spanning tree is $4 + 1 + 2 + 5 = 12$, and the weight of the second spanning tree is $2 + 1 + 2 + 5 = 10$.

In this case, the second spanning tree happens to be a *minimum spanning tree* of the graph. i.e., a spanning tree with the smallest possible weight. There can be multiple minimum spanning trees.

# Kruskal's algorithm

Kruskal's algorithm computes a minimum spanning tree of a graph with the help of a union-find data structure. The algorithm goes through the edges in order of weight, and selects an edge to be included in the spanning tree if adding it does not create a cycle in the tree.

For the above example graph, the algorithm goes through the edges as follows:

| Edge | Weight | Included in the tree? |
|---|---|---|
| $2 - 3$ | $1$ | Yes |
| $1 - 2$ | $2$ | Yes |
| $2 - 4$ | $2$ | Yes |
| $1 - 3$ | $4$ | No |
| $4 - 5$ | $5$ | Yes |
| $3 - 5$ | $7$ | No |

The algorithm first adds the edges $2 - 3$, $1 - 2$ and $2 - 4$ to the tree. The edge $1 - 3$ is excluded, because adding it would create a cycle. Then the edge $4 - 5$ is still added to the tree, after which the tree is complete.

If two edges have the same weight, Kruskal's algorithm can process them in either order.

The union-find data structure is useful for Kruskal's algorithm, because it can be used for efficiently checking if an edge should be included or excluded. If the nodes connected by the edge are in different components, adding the edge does not create a cycle.

The following code implements Kruskal's algorithm:

```
class Kruskal:
    def __init__(self, nodes):
        self.nodes = nodes
        self.edges = []

    def add_edge(self, node_a, node_b, weight):
        self.edges.append((node_a, node_b, weight))

    def construct(self):
        self.edges.sort(key=lambda x: x[2])

        uf = UnionFind(self.nodes)
        edges_count = 0
        tree_weight = 0

        for edge in self.edges:
            node_a, node_b, weight = edge
            if uf.find(node_a) != uf.find(node_b):
                uf.union(node_a, node_b)
                edges_count += 1
                tree_weight += weight

        if edges_count != len(self.nodes) - 1:
            return None
        return tree_weight
```

The method `construct` computes a minimum spanning tree and returns its the weight. The method first sorts the edge list by weight. Then the method iterates through the edges and selects the edges for the tree using a union-find data structure. When an edge is selected, its weight is added to the weight of the tree.

If a graph is not connected, it has no spanning tree. To detect this, the method counts the edges included in the tree. If the count at the end is less than $n-1$, the selected edges do not form a tree, which means that the graph is not connected. In this case, the method returns `None`.

The method can be used as follows:

```
k = Kruskal([1, 2, 3, 4, 5])

k.add_edge(1, 2, 2)
```

```
k.add_edge(1, 3, 4)
k.add_edge(2, 3, 1)
k.add_edge(2, 4, 2)
k.add_edge(3, 5, 7)
k.add_edge(4, 5, 5)

print(k.construct()) # 10
```

### Why the algorithm works correctly?

Kruskal's algorithm constructs the spanning tree greedily based on the weight order of edges. Why does this always produce a minimum spanning tree?

Consider a situation, where the edge $a - b$ is the next edge to be processed, and the nodes $a$ and $b$ are in different components. If the edge $a - b$ is not selected, then there must be another edge selected later that joins two components containing the nodes $a$ and $b$.

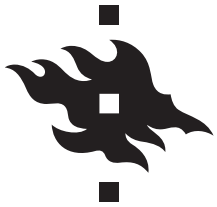
If we replace that later edge with the edge $a - b$, the spanning tree remains a spanning tree. Since the weight of that later edge is the same or greater than the weight of the edge $a - b$, the replacement does not increase the weight of the spanning tree. Thus selecting the edge $a - b$ is safe.

## Minimizing vs. maximizing

Minimizing and maximizing something related to graphs can be very different problems. For example, the shortest path between two nodes can be found using the algorithms in Chapter 14, but how can we find the *longest* path that visits each node at most once?

One approach that we could try is to negate the edge weights and then use one the shortest path algorithms. However, this may not work, because the negation can create negative cycles, which none of the algorithms can handle. In fact, no efficient algorithm is known for the longest path problem.

With spanning trees, however, this approach works: To find the *maximum* spanning tree, we can negate the edge weights and then find the minimum spanning tree. With Kruskal's algorithm, the same effect can be achieved by processing the edges in the reverse order of weight.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI