# Ugeopgave 4

Mathias Bramming Philip Meulengracht
Rasmus Borgsmidt

## Task 1

We use a disjoint-set forest, identical to the one described in CLRS section 21.3, e.g. a tree structure using *union by rank* and *path compression*. The `LINK` operation, works by finding the root of each set, and appending the tree with the smallest rank, to the biggest. In case the roots have the same ranks, one is incremented. It uses a helper method, `FIND-SET` to find the root of each tree.

```
1 LINK(i,j)
2         r1 = FIND-SET(i)
3         r2 = FIND-SET(j)
4         if r1.rank > r2.rank
5                 r2.p = r1
6         else    r1.p = r2
7                 if r1.rank == r2.rank
8                         r2.rank = r2.rank+1
```

Since all breweries with links are connected to the same tree, any given brewery $x$ and $y$ *must* have the same root if they are connected, which means that `QUERY` will only have to check if $x$ and $y$ have the same root:

```
1 QUERY(i,j)
2         return FIND-SET(i) == FIND-SET(j)
```

We use the `FIND-SET` operation from CLRS (two-pass method) as a helper function to find the root of a given node:

```
1 FIND-SET(x)
2         if x != x.p
3                 x.p = FIND-SET(x.p)
4         return x.p
```

According to CLRS, *Union by rank* has a tight running-time bound of $O(m \lg n)$, which means that our `LINK` operation has the same running time. The running time of `QUERY` has a best case of $O(1)$ and a worst case running time of $O(n)$: The first time the operation is used, it's possible that it has to traverse the entire tree (up to $n$ size) to find the root. The second pass, however, is in constant time, since all nodes now point directly to the root.

# Task 2

In order to solve this task, we extend the data structure from Task 1. Each vertex gets two new boolean fields, *marked* and *linked*, used to count the number of components in the supplied graph. These fields can obviously be set and cleared in $\Theta(1)$ time.

```
1  DECREMENT(G, U)
2    // Runs in O(m a(n)) due to union by rank and path
         compression, CLRS p. 571
3    comp_count = COUNT-COMPONENTS(G)
4
5    // Runs in O(m a(n)) due to union by rank and path
         compression, CLRS p. 571
6    for each unlink (u, v) in U
7      UNLINK(FIND-SET(u), FIND-SET(v))
8      if not QUERY(u, v)
9        increment comp_count
10     print comp_count
```

`DECREMENT` itself is very simple. It merely determines the number of components in the supplied graph using a helper function, `COUNT-COMPONENTS`, then applies each of the specified unlink operations. After breaking each connection, the `QUERY` function from Task 1 is used to check if the break caused a component split and the component count should be incremented. After that, simply printing the count yields the desired behavior.

The `COUNT-COMPONENTS` function below uses the new fields to count the number of components in $\Theta(m\alpha(n))$ time.

```
1  COUNT-COMPONENTS(G)
2    let R be an empty list of root vertices
3
4    // Runs in O(m a(n)) due to union by rank and path
         compression, CLRS p. 571
5    for each link (u, v) in G.E
6      u.linked = true
7      v.linked = true
8      r = FIND-SET(u)
9      r.marked = true
10     R = [r | R]
11
12   // Runs in O(m) as there can be at most m roots
13   root_count = 0
14   for each vertex r in R
15     if r.marked
16       increment root_count
17       r.marked = false
18
19   // Runs in O(m)
20   linked_count = 0
```

```
21    for each link (u, v) in G.E
22      if u.linked
23        increment linked_count
24        u.linked = false
25      if v.linked
26        increment linked_count
27        v.linked = false
28
29    return G.n - linked_count + root_count
```

The counting is calculated by determining three factors:

1. The total number of breweries $n$, which can be read directly from the supplied graph.

2. The number of breweries $k$ linked to one or more other breweries. There can be at most $2m$ such breweries, where $m$ is the total number of links.

3. The number of components $r$ that the $k$ breweries are divided into.

With this information at hand, the number of components in the graph can be calculated as $n - k + r$ because $n$ is the number of components there would be if there were no links at all, $k$ is the number of singleton components consumed by links, and $r$ is the number of components produced by the linking.

The counting proceeds as follows; we walk the list of $m$ links, for each setting the *linked* field on both breweries. The field is boolean, so it does not matter if a brewery has already been involved in another link, there is no double-counting. We also set the *marked* field on the root vertex, and again because the field is boolean there is no double-counting. The root vertex is also added to a list of roots.

Subsequently, we count the distinct roots by walking the list of root vertices built in the initial loop. For each root vertex, we increment the count if the *marked* field is set and then clear the field. This means that although each root vertex likely appears in the list multiple times, we count each distinct root only once.

Finally, we count the number of linked breweries using the same technique; only this time, we check and clear the *linked* instead of the *marked* field. We don't need to keep a separate list of breweries to check as we can simply walk the list of $m$ links again.

```
1  UNLINK(r1, r2)
2    r1.p = r1
3    r2.p = r2
```

The arguments of the `UNLINK` operation must be root vertices. It simply sets the $p$ field of each vertex to point to itself. One of these operations will be redundant but it is simpler just to set both than to check which one to set.

# Task 3

## Correctness

The correctness of `COUNT-COMPONENTS` is based on the property of any graph $G$ with $n$ vertices and $m$ connections, that its vertices are divided in two sections; $k \leq 2m$ vertices are linked together in $0 \leq r \leq m$ disjoint sets, while $n - k \geq 0$ vertices float around, each forever lost in their own little singleton set. Therefore by counting the number of sets in each section of the graph and adding them together, we obtain the total number of disjoint sets in the graph.

First our algorithm walks through the list of links in the graph, marking each vertex and their common root for later counting. If $n = 0$ then consequently $m = 0$ and the algorithm marks no vertices. The subsequent counting therefore correctly yields 0 components. If $n > 0$ and $m = 0$, all the vertices fall in the singleton section of the graph. Again no vertices get marked and the algorithm returns the correct answer $n$. If both $n > 0$ and $m > 0$, at least one vertex gets marked as root and at least one (possibly the same one) gets marked as linked. Because the extra fields in each vertex are boolean, it makes no difference if they are set more than once. After the loop has completed, it is guaranteed that every vertex in the linked section of the graph has the *linked* field set, and that every root of the linked vertices has the *marked* field set.

The next step is to count the number of roots $r$ in the linked section of the graph. In addition to marking vertices, the first loop of the algorithm also built a list of root vertices. This list has length $m$, and unless every link in the graph links together two otherwise unlinked vertices, there will be duplicates in this list. However, by walking the list and counting the number of vertices with the *marked* field set, provided we clear the field as soon as we count it, we can count the number of *distinct* root vertices in the list. Therefore this loop correctly calculates $r$.

Then the algorithm counts the number of vertices (including roots) that fall in the linked section of the graph. The technique is the same as before. The first loop marked every vertex with a link at least once, so by walking the list of links again, counting the number of vertices with the *linked* field set, provided we clear the field as soon as we count it, we can count the number of *distinct* vertices in the linked section of the graph. Therefore this loop correctly calculates $k$.

The final step of `COUNT-COMPONENTS` simply returns $n - k + r$ which yields the correct result.

Having determined the number of components in the graph, the `DECREMENT` function itself is simple. It loops through the list of supplied unlink operations. If the length of this list is empty, the algorithm correctly does not print any output. Otherwise it prints the number of components in the graph after completing each `UNLINK` operation. Since two linked vertices are known to share a common root, checking if they still share a common root after unlinking them is sufficient to determine if the operation resulted in two split components. And since any link always joins exactly two (possibly entangled) subcomponents, breaking a single link can never produce more than two disjoint parts. Therefore, simply

incrementing the component count by 1 every time the two roots are distinct is correct and the algorithm prints the correct result.

## Running Time

The running time of $\Theta(m\alpha(n))$ is based on the argument in CLRS, page 571. Due to path compression, finding the root of $m$ linked vertices is nearly linear in $m$. It is illustrated well in figure 21.5. Asking for the root of $a$ is expensive but after completing that operation, asking for any root is $\Theta(1)$. Therefore the running time of calling `FIND-SET` on $m$ nodes is nearly linear in $m$.

So looking at `COUNT-COMPONENTS`, the first loop has $m$ iterations and completes in $\Theta(m\alpha(n))$ because the only non-constant-time operation in the loop is `FIND-SET` mentioned above. The second loop has $\leq m$ iterations, performs only constant-time operations and therefore completes in $\Theta(m)$ time. The last loop has $m$ iterations, performs only constant-time operations and therefore completes in $\Theta(m)$ time. The overall running time is therefore:

$$\Theta(m\alpha(n)) + \Theta(m) + \Theta(m) = \Theta(m\alpha(n)).$$

To this, `DECREMENT` itself adds a single loop with $\leq m$ iterations. The call to `UNLINK` is constant in time, the two calls to `FIND-SET` as described above is linear in $m$ over the loop as a whole and the same applies to the two calls to `FIND-SET` hidden in `QUERY`. The overall running time is therefore:

$$\Theta(m\alpha(n)) + \Theta(m\alpha(n)) = \Theta(m\alpha(n)).$$