University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
Marc Fuchs

Algorithm Theory
Sample Solution Exercise Sheet 6

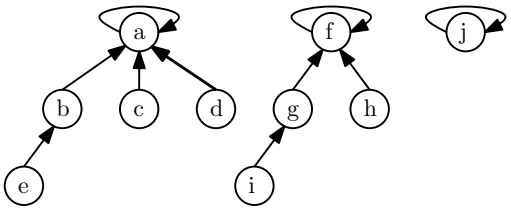
Due: Friday, 28th of November 2025, 10:00 am

Exercise 1: Union-Find *(11 Points)*

In the lecture we have seen two heuristics (i.e., the **union-by-size** and the **union-by-rank** heuristic) to implement the **union-find** data structure. In this exercise we will focus on the **union-by-rank** heuristic only! To solve the following tasks consider the **union-find** data structure implemented by disjoint forest using union-by-rank heuristic and path compression.

(a) Give the pseudocode for `union(x, y)`. *(2 Points)*
Remark: Use $x.parent$ to access the parent of some node x and use $x.rank$ to get its rank. The `find(x)` operation is implemented as stated in the lecture using path compression.

(b) Consider the following state of such a union-find data structure represented as disjoint-set forest where the *current* rank of each node equals the height of the tree rooted at it.



Conduct the following operations *sequentially* on the union-find data structure (the result of the prior operation is the input of the next). Give the state of the data structure after each operation.

(i) `union(`f, j`)` using the *union-by-rank* heuristic *(1 Point)*

(ii) `union(`b, g`)` using the *union-by-rank* heuristic *(1 Point)*

(iii) `find(`i`)` using *path compression* *(1 Point)*

(c) Show that the height of each tree (in the disjoint forest) is at most $O(\log n)$ where n is the number of nodes. *(3 Points)*
Remark: First show that the maximum rank of a tree is at most $O(\log n)$, and then explain why this maximum rank is an upper bound on the height of the tree.

(d) Show that the above's bound is tight, i.e., give an example execution (of `makeSet`'s and `union`'s) that creates a tree of height $\Theta(\log n)$. Proof your statement! *(3 Points)*

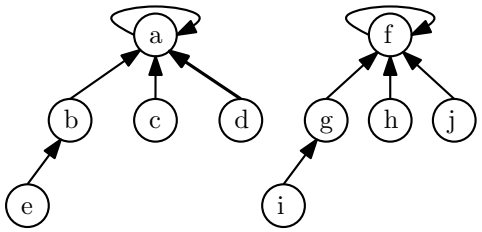
Sample Solution

(a) The pseudocode is given below.

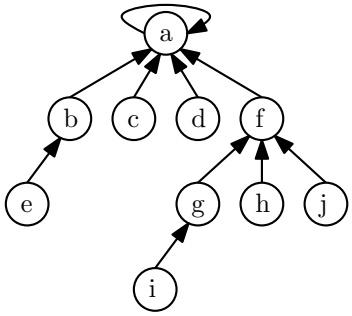
Algorithm 1 `union`(x, y)

```
a := find(x)
b := find(y)
if a.rank > b.rank then
    b.parent := a
    return a                                  ▷ returning the root of the new set (optional)
else
    a.parent = b
    if a.rank = b.rank then
        b.rank = b.rank + 1
    return b                                  ▷ returning the root of the new set (optional)
```

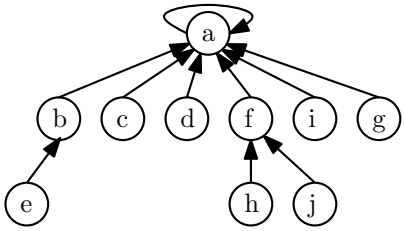
(b) (i) after `union(`f, j`)` :



(ii) after `union(`b, g`)`:



(iii) after `find(`i`)`:



(c) We will first argue why the maximum rank of a tree upper bounds the hight of this tree. As shown in the lectrue, the rank is exactly the height of the tree when we ignore path compression. Consider path compression, we may construct new shortcuts from nodes to the root of the tree, that can lower the height of three, but never increase it. Hence, for any tree, the rank of the root node of this tree is an upper bound of the height of this tree.
We will now show that the maximum rank of a tree is $O(\log n)$. We will show by induction over the rank of the tree that for any tree T with $T.rank = r$, T contains at least $|T| \geq 2^r$ many nodes. We start our induction at $r = 0$. Surely, only trees with exactly one node can have a rank of 0, what fulfills the condition.

Induction Hypothesis: For a fixed r, any tree T_r of rank r contains at least 2^r nodes.

Induction Step: Let T_{r+1} be a tree with rank $r + 1$. Since the rank of a tree increases only then when it is merged with another tree of equal rank, we can say w.l.o.g that T_{r+1} was created by

the union of trees T_r and T'_r, both with rank r. Due to our hypothesis, we know $|T_r| \geq 2^r$ and $|T'_r| \geq 2^r$. Since T_{r+1} contains at least the nodes of T_r and T'_r we have

$$|T_{r+1}| \geq |T_r| + |T'_r| \geq 2^r + 2^r = 2^{r+1}$$

which ends the inductive proof.

We can conclude that for any tree T_r: $n = |T_r| \geq 2^r \Rightarrow height(T_r) \leq r \leq \log_2 n$.

(d) For simplicity, assume we have $n = 2^k$ nodes that we add to our union-find data structure using `makeSet`. We now have n trees of rank 0. Next, we `union` all pairs of trees s.t. we get $n/2$ trees of rank 1. Again, we `union` over all pairs of trees and get $n/4$ trees of rank 2. Continuing like this, we will finally get a single tree with rank k. Note that even with *path compression*, the rank of these trees is equal to its height, since we can always use the roots of each tree for our `union`-method from (a) and therefore the find operation will not rearrange the pointers of the children. Hereby our execution leads to a tree with $height = k = \log_2 n$.

Note that if n is not a power of 2, we can still use the same construction and finally get 2 trees where the larger one contains more than $n/2$ of the nodes. This tree's height is $\log_2 n - 1 = \Theta(\log n)$.

Exercise 2: Coming Home *(9 Points)*

Imagine we are in a city with N houses and N people. Each house is numbered from 1 to N and each person is numbered from 1 to N as well. We say that person i's home is house i. However, due to some unexplainable occurrence, each person wakes up in an arbitrary house (random permutation over the numbers 1 to N) and wants to go back to his own. Furthermore, they can not just walk home outside, they need to use hidden tunnels between the houses. These M tunnels are bidirectional connections that connect two distinct houses. The big question here is *if anyone can get home through the tunnels.*

(a) What (graph based) property needs to be fulfilled that each person located at house p_i can come back home to house i? Imagine that each house represents a node in a graph and two nodes are connected by an edge if there exists a tunnel between the representative houses. *(1 Point)*

To make the question more interesting, assume each tunnel t has a capacity $c_t \in \mathbb{N}$ and persons prefer to go through tunnels with large capacities.

(b) Given a fixed threshold $W > 0$, give an algorithm that decides if every person can get home by only taking tunnels with capacity at least W. Your algorithm should run in time $O((N + M) \cdot \log^* N)$. Argue why this is the case. *(5 Points)*

(c) Not given a fixed W, can you find the largest possible W such that a solution exists (i.e., every person can get home)? What is the runtime? Try to make your algorithm as efficient as possible! *(3 Points)*

Sample Solution

(a) For the person in house p_i to get back to house i there must exist a path (in the described graph) between the nodes i and p_i. Such a path has to exist for all pairs of (i, p_i) for all houses $1 \leq i \leq N$. In other words, for all such i, nodes i and p_i have to be in the same **connected component**.

(b) We use the union-find data structure. We first add all the N houses. Then for every tunnel $t = (a_t, b_t)$ with capacity $c_t \geq W$ we perform `union`(a_t, b_t). Note that when 2 houses, say x and y have the same representative in this data structure, they are in the same connected component, i.e., there is a path of tunnels that connects these two houses. Hence, we can check with `find`(p_i) ==`find`(i) if person i can get home. The runtime thus combines `make-set` the N elements/houses, the $\leq M$ `union`s and finally the N `find` operations to verify that a solution

exits. In the lecture we have seen that m `union-find` on n elements take time $O(m \cdot \log^* n)$, since we have $M + N$ union-find operations, the runtime is as stated in the task.

(c) We just sort all the weights and use binary search on finding the correct capacity and use the (b) subtask to solve the created subproblems. So we first sort all capacities in time $O(M \log M)$ and then do this binary search on this sorted array. The overall runtime thus is: $O(M \log M + \log M \cdot (M + N) \cdot \log^* N) = O(\log^* n \cdot \log M \cdot (M + N))$.