Recognizing Union-Find Trees Built Up Using Union-By-Rank Strategy is NP-Complete

Kitti Gelle and Szabolcs Iván(✉)

Department of Computer Science, University of Szeged, Szeged, Hungary
{kgelle,szabivan}@inf.u-szeged.hu

Abstract. Disjoint-Set forests, consisting of Union-Find trees, are data structures having a widespread practical application due to their efficiency. Despite them being well-known, no exact structural characterization of these trees is known (such a characterization exists for Union trees which are constructed without using path compression) for the case assuming union-by-rank strategy for merging. In this paper we provide such a characterization by means of a simple PUSH operation and show that the decision problem whether a given tree (along with the rank info of its nodes) is a Union-Find tree is **NP**-complete, complementing our earlier similar result for the union-by-size strategy.

1 Introduction

Disjoint-Set forests, introduced in [10], are fundamental data structures in many practical algorithms where one has to maintain a partition of some set, which supports three operations: *creating* a partition consisting of singletons, *querying* whether two given elements are in the same class of the partition (or equivalently: *finding* a representative of a class, given an element of it) and *merging* two classes. Practical examples include e.g. building a minimum-cost spanning tree of a weighted graph [4], unification algorithms [17] etc.

To support these operations, even a linked list representation suffices but to achieve an almost-constant amortized time cost per operation, Disjoint-Set forests are used in practice. In this data structure, sets are represented as directed trees with the edges directed towards the root; the CREATE operation creates n trees having one node each (here n stands for the number of the elements in the universe), the FIND operation takes a node and returns the root of the tree in which the node is present (thus the SAME-CLASS(x, y) operation is implemented as FIND(x) == FIND(y)), and the MERGE(x, y) operation is implemented by merging the trees containing x and y, i.e. making one of the root nodes to be a child of the other root node (if the two nodes are in different classes).

In order to achieve near-constant efficiency, one has to keep the (average) height of the trees small. There are two "orthogonal" methods to do that: first, during the merge operation it is advisable to attach the "smaller" tree below the

Research was supported by the NKFI grant no. 108448.

G. Pighizzini and C. Câmpeanu (Eds.): DCFS 2017, LNCS 10316, pp. 152–163, 2017.
DOI: 10.1007/978-3-319-60252-3_12

"larger" one. If the "size" of a tree is the number of its nodes, we say the trees are built up according to the *union-by-size* strategy, if it's the depth of a tree, then we talk about the *union-by-rank* strategy. Second, during a FIND operation invoked on some node x of a tree, one can apply the *path compression* method, which reattaches each ancestor of x directly to the root of the tree in which they are present. If one applies both the path compression method and either one of the union-by-size or union-by-rank strategies, then any sequence of m operations on a universe of n elements has worst-case time cost $O(m\alpha(n))$ where α is the inverse of the extremely fast growing (not primitive recursive) Ackermann function for which $\alpha(n) \leq 5$ for each practical value of n (say, below 2^{65535}), hence it has an amortized almost-constant time cost [22]. Since it's proven [9] that *any* data structure maintaining a partition has worst-case time cost $\Omega(m\alpha(n))$, the Disjoint-Set forests equipped with a strategy and path compression offer a theoretically optimal data structure which performs exceptionally well also in practice. For more details see standard textbooks on data structures, e.g. [4].

Due to these facts, it is certainly interesting both from the theoretical as well as the practical point of view to characterize those trees that can arise from a forest of singletons after a number of merge and find operations, which we call Union-Find trees in this paper. One could e.g. test Disjoint-Set implementations since if at any given point of execution a tree of a Disjoint-Set forest is not a valid Union-Find tree, then it is certain that there is a bug in the implementation of the data structure (though we note at this point that this data structure is sometimes regarded as one of the "primitive" data structures, in the sense that it is possible to implement a correct version of them that needs not be certifying [20]). Nevertheless, only the characterization of Union trees is known up till now [2], i.e. which correspond to the case when one uses one of the union-by- strategies but *not* path compression. Since in that case the data structure offers only a theoretic bound of $\Theta(\log n)$ on the amortized time cost, in practice all implementations imbue path compression as well, so for a characterization to be really useful, it has to cover this case as well.

In this paper we show that the recognition problem of Union-Find trees is **NP**-complete when the union-by-rank strategy is used, complementing our earlier results [13] where we proved **NP**-completeness for the union-by-size strategy. The proof method applied here resembles to that one, but the low-level details for the reduction (here we use the PARTITION problem, there we used the more restricted version 3−PARTITION as this is a very canonical strongly **NP**-complete problem) differ greatly. This result also confirms the statement from [2] that the problem "seems to be much harder" than recognizing Union trees. As (up to our knowledge) in most of the actual software libraries having this data structure implemented the union-by-rank strategy is used (apart from the cases when one quickly has to query the size of the sets as well), for software testing purposes the current result is more relevant than the one applying union-by-size strategy.

Related Work. There is an increasing interest in determining the complexity of the recognition problem of various data structures. The problem was considered for suffix trees [16,21], (parametrized) border arrays [8,14,15,19], suffix arrays [1,7,18], KMP tables [6,12], prefix tables [3], cover arrays [5], and directed acyclic word- and subsequence graphs [1].

2 Notation

A *(ranked) tree* is a tuple $t = (V_t, \textsc{root}_t, \textsc{rank}_t, \textsc{parent}_t)$ with V_t being the finite set of its *nodes*, $\textsc{root}_t \in V_t$ its *root*, $\textsc{rank}_t : V_t \to \mathbb{N}_0$ mapping a nonnegative integer to each node, and $\textsc{parent}_t : (V_t - \{\textsc{root}_t\}) \to V_t$ mapping each nonroot node to its parent (so that the graph of $\textsc{parent}_t$ is a directed acyclic graph, with edges being directed towards the root). We require $\textsc{rank}_t(x) < \textsc{rank}_t(\textsc{parent}_t(x))$ for each nonroot node x, i.e. the rank strictly decreases towards the leaves.

For a tree t and a node $x \in V_t$, let $\textsc{children}(t, x)$ stand for the set $\{y \in V_t : \textsc{parent}_t(y) = x\}$ of its children and $\textsc{children}(t)$ stand as a shorthand for $\textsc{children}(t, \textsc{root}_t)$, the set of depth-one nodes of t. Also, let $x \preceq_t y$ denote that x is a (non-strict) *ancestor* of y in t, i.e. $x = \textsc{parent}_t^k(y)$ for some $k \geq 0$. For $x \in V_t$, let $t|_x$ stand for the *subtree* $(V_x = \{y \in V : x \preceq_t y\}, x, \textsc{rank}_t|_{V_x}, \textsc{parent}_t|_{V_x})$ of t rooted at x. As shorthand, let $\textsc{rank}(t)$ stand for $\textsc{rank}_t(\textsc{root}_t)$, the rank of the root of t.

Two operations on trees are that of *merging* and *collapsing*. Given two trees $t = (V_t, \textsc{root}_t, \textsc{rank}_t, \textsc{parent}_t)$ and $s = (V_s, \textsc{root}_s, \textsc{rank}_s, \textsc{parent}_s)$ with V_t and V_s being disjoint and $\textsc{rank}(t) \geq \textsc{rank}(s)$, then their *merge* $\textsc{merge}(t, s)$ (in this order) is the tree $(V_t \cup V_s, \textsc{root}_t, \textsc{rank}, \textsc{parent})$ with $\textsc{parent}(x) = \textsc{parent}_t(x)$ for $x \in V_t$, $\textsc{parent}(\textsc{root}_s) = \textsc{root}_t$ and $\textsc{parent}(y) = \textsc{parent}_s(y)$ for each nonroot node $y \in V_s$ of s, and

$$\textsc{rank}(\textsc{root}_t) = \begin{cases} \textsc{rank}(t) & \text{if } \textsc{rank}(s) < \textsc{rank}(t), \\ \textsc{rank}(t) + 1 & \text{otherwise,} \end{cases}$$

and $\textsc{rank}(x) = \textsc{rank}_t(x)$, $\textsc{rank}_s(x)$ resp. for each $x \in V_t - \{\textsc{root}_r\}$, $x \in V_s$ resp.

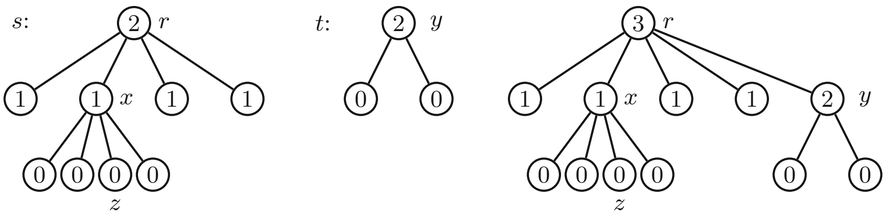
Given a tree $t = (V, \textsc{root}, \textsc{rank}, \textsc{parent})$ and a node $x \in V$, then $\textsc{collapse}(t, x)$ is the tree $(V, \textsc{root}, \textsc{rank}, \textsc{parent}')$ with $\textsc{parent}'(y) = \textsc{root}$ if y is a nonroot ancestor of x in t and $\textsc{parent}'(y) = \textsc{parent}(y)$ otherwise. For examples, see Fig. 1.

Observe that both operations indeed construct a ranked tree (e.g. the rank remains strictly decreasing towards the leaves).

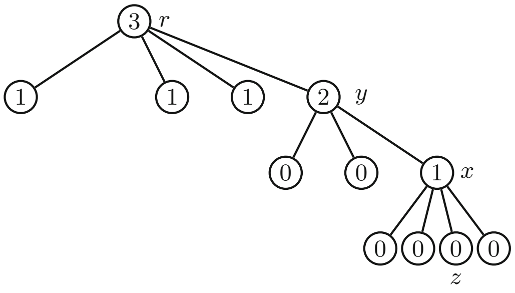
We say that a tree is a *singleton* tree if it has exactly one node, and this node has rank 0.

The class of Union trees is the least class of trees satisfying the following two conditions: every singleton tree is a Union tree, and if t and s are Union trees with $\textsc{rank}(t) \geq \textsc{rank}(s)$, then $\textsc{merge}(t, s)$ is a Union tree as well.

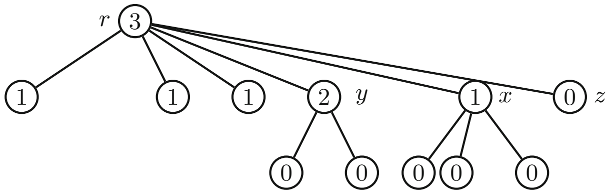
Analogously, the class of Union-Find trees is the least class of trees satisfying the following three conditions: every singleton tree is a Union-Find tree, if t and



(a) Trees s and t (b) $t' = \text{MERGE}(s,t)$



(c) $t'' = \text{PUSH}(t', x, y)$



(d) $t''' = \text{COLLAPSE}(t'', z)$

Fig. 1. Merge, collapse and push.

s are Union-Find trees with $\text{RANK}(t) \geq \text{RANK}(s)$, then $\text{MERGE}(t,s)$ is a Union-Find tree as well, and if t is a Union-Find tree and $x \in V_t$ is a node of t, then $\text{COLLAPSE}(t,x)$ is also a Union-Find tree.

We say that a node x of a tree t *satisfies the Union condition* if

$$\{\text{RANK}_t(y) : y \in \text{CHILDREN}(t,x)\} = \{0, 1, \ldots, \text{RANK}_t(x) - 1\}.$$

Then, the characterization of Union trees from [2] can be formulated in our terms as follows:

Theorem 1. *A tree t is a Union tree if and only if each node of t satisfies the Union condition.*

Note that the rank of a Union tree always coincides by its *height*. (And, any subtree of a Union tree is also a Union tree.) In particular, the leaves are exactly those nodes of rank 0.

3 Structural Characterization of Union-Find Trees

Suppose s and t are trees on the same set V of nodes, with the same root $\textsc{root}$ and the same rank function $\textsc{rank}$. We write $s \preceq t$ if $x \preceq_s y$ implies $x \preceq_t y$ for each $x, y \in V$.

Clearly, $\preceq$ is a partial order on any set of trees (i.e. is a reflexive, transitive and antisymmetric relation). It is also clear that $s \preceq t$ if and only if $\textsc{parent}_s(x) \preceq_t x$ holds for each $x \in V - \{\textsc{root}\}$ which is further equivalent to requiring $\textsc{parent}_s(x) \preceq_t \textsc{parent}_t(x)$ since $\textsc{parent}_s(x)$ cannot be x.

Another notion we define is the (partial) operation $\textsc{push}$ on trees as follows: when t is a tree and $x \neq y \in V_t$ are *siblings* in t, i.e. have the same parent, and $\textsc{rank}_t(x) < \textsc{rank}_t(y)$, then $\textsc{push}(t, x, y)$ is defined as the tree $(V_t, \textsc{root}_t, \textsc{rank}_t, \textsc{parent}')$ with

$$\textsc{parent}'(z) = \begin{cases} y & \text{if } z = x, \\ \textsc{parent}_t(z) & \text{otherwise,} \end{cases}$$

that is, we "push" the node x one level deeper in the tree just below its former sibling y. (See Fig. 1.)

We write $t \vdash t'$ when $t' = \textsc{push}(t, x, y)$ for some x and y, and as usual, $\vdash^*$ denotes the reflexive-transitive closure of $\vdash$.

Proposition 1. *For any pair s and t of trees, the following conditions are equivalent:*

(i) $s \preceq t$,
(ii) *there exists a sequence $t_0 = s, t_1, t_2, \ldots, t_n$ of trees such that for each $i = 1, \ldots, n$ we have $t_i = \textsc{push}(t_{i-1}, x, y)$ for some depth-one node $x \in \textsc{children}(t_{i-1})$, moreover, $\textsc{children}(t_n) = \textsc{children}(t)$ and $t_n|_x \preceq t|_x$ for each $x \in \textsc{children}(t)$,*
(iii) $s \vdash^* t$.

Proof. **(i)⇒(ii).** It is clear that $\preceq$ is equality on singleton trees, thus $\preceq$ implies $\vdash^*$ for trees of rank 0. Assume $s \preceq t$ for the trees $s = (V, \textsc{root}, \textsc{rank}, \textsc{parent})$ and $t = (V, \textsc{root}, \textsc{rank}, \textsc{parent}')$ and let X stand for the set $\textsc{children}(s)$ of the depth-one nodes of s and Y stand for $\textsc{children}(t)$. Clearly, $Y \subseteq X$ since by $s \preceq t$, any node x of s having depth at least two has to satisfy $\textsc{parent}(x) \preceq_t \textsc{parent}'(x)$ and since $\textsc{parent}(x) \neq \textsc{root}$ for such nodes, x has to have depth at least two in t as well. Now there are two cases: either $\textsc{root} = \textsc{parent}(x) = \textsc{parent}'(x)$ for each $x \in X$, or $\textsc{parent}(x) \prec_t \textsc{parent}'(x)$ for some $x \in X$.

If $\textsc{parent}'(x) = \textsc{root}$ for each $x \in X$, then $X = Y$ and we only have to show that $s|_x \preceq t|_x$ for each $x \in X$. For this, let $u, v \in V(s|_x)$ with $u \preceq_{s|_x} v$. Since $s|_x$ is a subtree of s, this holds if and only if $x \preceq_s u \preceq_s v$. From $s \preceq t$ this implies $x \preceq_t u \preceq_t v$, that is, $u \preceq_{t|_x} v$, hence $s|_x \preceq t|_x$.

Now assume $\textsc{parent}(x) \prec_t \textsc{parent}'(x)$ for some $x \in X$. Then $\textsc{parent}'(x) \neq \textsc{root}$, thus there exists some $y \in Y$ with $y \preceq_t \textsc{parent}'(x)$. By $Y \subseteq X$, this y is a member of X as well, and $\textsc{rank}_s(y) = \textsc{rank}_t(y) > \textsc{rank}_t(x) = \textsc{rank}_s(x)$, thus

$s' = \textsc{push}(s,x,y)$ is well-defined. Moreover, $s' \preceq t$ since $\textsc{parent}_{s'}(z) \preceq_t z$ for each $z \in V$: either $z \neq x$ in which case $\textsc{parent}_{s'}(z) = \textsc{parent}(z) \preceq_t z$ by $s \preceq t$, or $z = x$ and then $\textsc{parent}_{s'}(z) = y \preceq_t \textsc{parent}'(x) \preceq_t x = z$ also holds. Thus, there exists a tree $s' = \textsc{push}(s,x,y)$ for some $x \in \textsc{children}(s)$ with $s' \preceq t$; since $\textsc{children}(s') = X - \{x\}$, by repeating this construction we eventually arrive to a tree t_n with $|\textsc{children}(t_n)| = |Y|$, implying $\textsc{children}(t_n) = Y$ by $Y \subseteq \textsc{children}(t_n)$.

(ii)⇒(iii). We apply induction on $\textsc{rank}(s) = \textsc{rank}(t)$. When $\textsc{rank}(s) = 0$, then s is a singleton tree and the condition in ii) ensures that t is a singleton tree as well. Thus, $s = t$ and clearly $s \vdash^* t$.

Now let assume the claim holds for each pair of trees of rank less than $\textsc{rank}(s)$ and let $t_0, \ldots, t_n$ be trees satisfying the condition. Then, by construction, $s \vdash^* t_n$. Since $\textsc{rank}(t_n|_x) < \textsc{rank}(t_n) = \textsc{rank}(s)$ for each node $x \in \textsc{children}(t_n)$, by $t_n|_x \preceq t|_x$ we get applying the induction hypothesis that $t_n|_x \vdash^* t|_x$ for each depth-one node x of t_n, thus $t_n \vdash^* t$, hence $s \vdash^* t$ as well.

(iii) ⇒ (i). For $\vdash^*$ implying $\preceq$ it suffices to show that $\vdash$ implies $\preceq$ since the latter is reflexive and transitive. So let $s = (V, r, \textsc{rank}, \textsc{parent})$ and $x \neq y \in V$ be siblings in s with the common parent z, $\textsc{rank}(x) < \textsc{rank}(y)$ and let $t = \textsc{push}(s,x,y)$. Then, since $\textsc{parent}_s(x) = z = \textsc{parent}_t(y) = \textsc{parent}_t(\textsc{parent}_t(x))$, we get $\textsc{parent}_s(x) \preceq_t x$, and by $\textsc{parent}_s(w) = \textsc{parent}_t(w)$ for each node $w \neq x$, we have $s \preceq t$.

□

The relations $\preceq$ and $\vdash^*$ are introduced due to their intimate relation to Union-Find and Union trees (similarly to the case of the union-by-size strategy [13], but there the $\textsc{push}$ operation itself was slightly different):

Theorem 2. *A tree t is a Union-Find tree if and only if $t \vdash^* s$ for some Union tree s.*

Proof. Let t be a Union-Find tree. We show the claim by structural induction. For singleton trees the claim holds since any singleton tree is a Union tree as well. Suppose $t = \textsc{merge}(t_1, t_2)$. Then by the induction hypothesis, $t_1 \vdash^* s_1$ and $t_2 \vdash^* s_2$ for the Union trees s_1 and s_2. Then, for the tree $s = \textsc{merge}(s_1, s_2)$ we get that $t \vdash^* s$. Finally, assume $t = \textsc{collapse}(t', x)$ for some node x. Let $x = x_1 \succ x_2 \succ \ldots \succ x_k = \textsc{root}_{t'}$ be the ancestral sequence of x in t'. Then, defining $t_0 = t$, $t_i = \textsc{push}(t_{i-1}, x_i, x_{i+1})$ we get that $t \vdash^* t_{k-2} = t'$ and $t' \vdash^* s$ for some Union tree s applying the induction hypothesis, thus $t \vdash^* s$ also holds.

Now assume $t \vdash^* s$ (equivalently, $t \preceq s$) for some Union tree s. We show the claim by induction on the height of t. For singleton trees the claim holds since any singleton tree is a Union-Find tree.

Now assume $t = (V, \textsc{root}, \textsc{rank}, \textsc{parent})$ is a tree and $t \vdash^* s$ for some Union tree s. Then by Proposition 1, there is a set $X = \textsc{children}(s) \subseteq \textsc{children}(t)$ of depth-one nodes of t and a function $f : Y \to X$ with $Y = \{y_1, \ldots, y_\ell\} = \textsc{children}(t) - X$ such that for the sequence $t_0 = t$, $t_i = \textsc{push}(t_{i-1}, y_i, f(y_i))$ we have that $t_\ell|_x \preceq s|_x$ for each $x \in X$. As each $s|_x$ is a Union tree (since so is s), we have by the induction hypothesis that each $t_\ell|_x$ is a Union-Find tree. Now let

$X = \{x_1, \ldots, x_k\}$ be ordered nondecreasingly by rank; then, as s is a Union tree and $X = \textsc{children}(s)$, we get that $\{\textsc{rank}(x_i)\} = \{0, 1, \ldots, \textsc{rank}(\textsc{root}) - 1\}$ by Theorem 1. Hence for the sequence t'_i defined as t'_0 being a singleton tree with root $\textsc{root}$ and for each $i \in \{1, \ldots, k\}$, $t'_i = \textsc{merge}(t'_{i-1}, t_\ell|_{x_i})$, we get that $t_\ell = t'_k$ is a Union-Find tree. Finally, we get t from t_ℓ by applying successively one $\textsc{collapse}$ operaton on each node in Y, thus t is a Union-Find tree as well. □

4 Complexity

In order to show **NP**-completeness of the recognition problem, we first make a useful observation.

Proposition 2. *In any Union-Find tree t there are at least as many rank-0 nodes as nodes of positive rank.*

Proof. We apply induction on the structure of t. The claim holds for singleton trees (having one single node of rank 0). Let $t = \textsc{merge}(t_1, t_2)$ and suppose the claim holds for t_1 and t_2. There are two cases.

- Assume $\textsc{rank}(t_1) = 0$. Then, since $\textsc{rank}(t_1) \geq \textsc{rank}(t_2)$ we have that $\textsc{rank}(t_2)$ is 0 as well, i.e. both t_1 and t_2 are singleton trees (of rank 0). In this case t has one node of rank 1 and one node of rank 0.
- If $\textsc{rank}(t_1) > 0$, then (since $\textsc{root}_{t_1}$ is the only node in $V_t = V_{t_1} \cup V_{t_2}$ whose rank can change at all, in which case it increases) neither the total number of rank-0 nodes nor the total number of nodes with positive rank changes, thus the claim holds.

Let $t = \textsc{collapse}(s, x)$ and assume the claim holds for s. Then, since the $\textsc{collapse}$ operation does not change the rank of any of the nodes, the claim holds for t as well. □

In order to define a reduction from the strongly **NP**-complete problem Partition we introduce several notions on trees:

An *apple* of weight a for an integer $a > 0$ is a tree consisting of a root node of rank 2, a depth-one node of rank 0 and a depth-one nodes of rank 1.

A *basket* of size H for an integer $H > 0$ is a tree consisting of $H+4$ nodes: the root node having rank 3, $H + 1$ depth-one children of rank 0 and one depth-one child of rank 1, which in turn has a child of rank 0.

A *flat tree* is a tree t of the following form: the root of t has rank 4. The immediate subtrees of t are:

- a node of rank 0, having no children;
- a node of rank 1, having a single child of rank 0;
- a node of rank 2, having two children: a single node of rank 0 and a node of rank 1, having a single child of rank 0;
- an arbitrary number of apples,

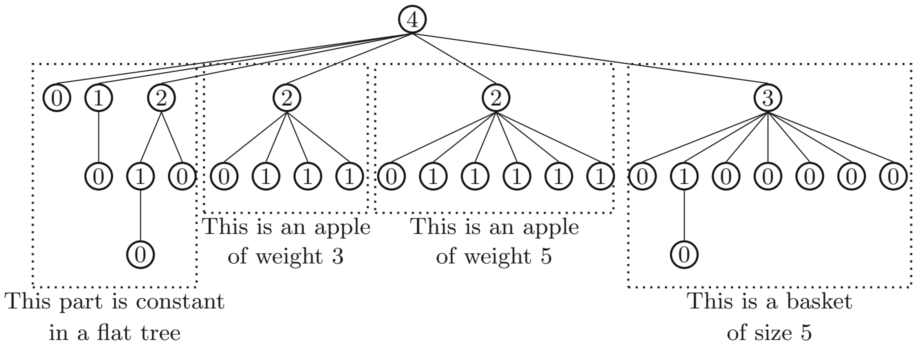


Fig. 2. A flat tree.

– and an arbitrary number of baskets for some fixed size H.

(See Fig. 2.)

At this point we recall that the following problem Partition is **NP**-complete in the strong sense [11]: given a list $a_1, \ldots, a_m$ of positive integers and a value $k > 0$ such that the value $B = \frac{\sum_{i=1}^{m} a_i}{k}$ is an integer, does there exist a partition $\mathcal{B} = \{B_1, \ldots, B_k\}$ of the set $\{1, \ldots, m\}$ satisfying $\sum_{i \in B_j} a_i = B$ for each $1 \leq j \leq k$?

(Here "in the strong sense" means that the problem remains **NP**-complete even if the numbers are encoded in unary.)

Proposition 3. *Assume t is a flat tree having k basket children, each having the size H, and m apple children of weights $a_1, \ldots, a_m$ respectively, satisfying $H \cdot k = \sum_{1 \leq i \leq m} a_i$.*

Then t is a Union-Find tree if and only if the instance $(a_1, \ldots, a_m, k)$ is a positive instance of the Partition *problem.*

Proof. (For an example, the reader is referred to Fig. 3.)

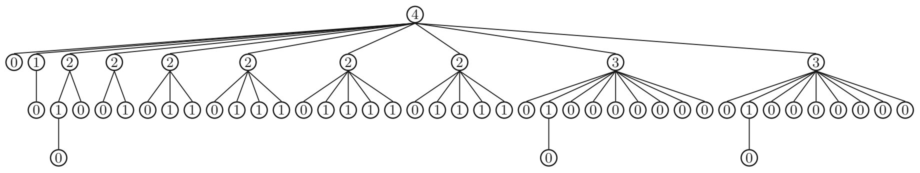


Fig. 3. The initial flat tree t corresponding to the Partition instance $(1, 2, 3, 4, 4, k = 2)$. The size of each basket is $(1 + 2 + 3 + 4 + 4)/k = 7$.

Suppose $\mathcal{I} = (a_1, \ldots, a_m, k)$ is a positive instance of the Partition problem. Let H stand for the target sum $\frac{\sum a_i}{k}$. Let $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$ be a solution of $\mathcal{I}$, i.e., $\sum_{i \in \mathcal{B}_j} a_i = H$ for each $j = 1, \ldots, k$. Let $x_1, \ldots, x_k \in \textsc{children}(t)$ be the nodes corresponding to the baskets of t and let $y_1, \ldots, y_m \in \textsc{children}(t)$ be the nodes corresponding to the apples of t.

We define the following sequence $t_0, t_1, \ldots, t_m$ of trees: $t_0 = t$ and for each $i = 1, \ldots, m$, let $t_i = \textsc{push}(t_{i-1}, y_i, x_j)$ with $1 \leq j \leq k$ being the unique index with $i \in \mathcal{B}_j$. Then, $\textsc{children}(t_m)$ consists of $x_1, \ldots, x_k$ and the three additional nodes having rank 0, 1 and 2. Note that the subtrees rooted at the latter three nodes are Union trees. Thus, if each of the trees $t_m|_{x_j}$ is a Union-Find tree, then so is t.

Consider a subtree $t' = t_m|_{x_j}$. By construction, t' is a tree whose root has rank 3 and has

- $H + 1$ children of rank 0,
- a single child of rank 1, having a child of rank 0,
- and several (say, ℓ) apple children with total weight H.

We give a method to transform t' into a Union tree. First, we push a_i rank-0 nodes to each apple child of weight a_i. After this stage t' has one child of rank 0, one child of rank 1 and ℓ "filled" apple children, having a root of rank 2, thus the root of the transformed t' satisfies the Union condition. We only have to show that each of these "filled" apples is a Union-Find tree.

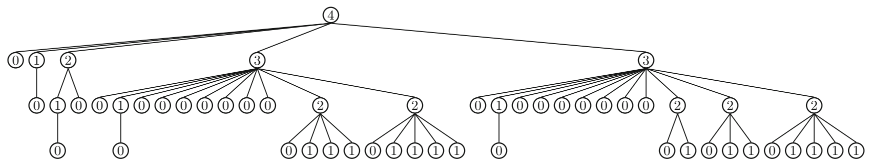
Such a subtree has a root node of rank 2, a_i depth-one nodes of rank 1 and $a_i + 1$ depth-one nodes of rank 0. Then, one can push into each node of rank 1 a node of rank 0 and arrive to a tree with one depth-one node of rank 0, and a_i depth-one nodes of rank 1, each having a single child of rank 0, which is indeed a Union tree, showing the claim by Theorem 2.

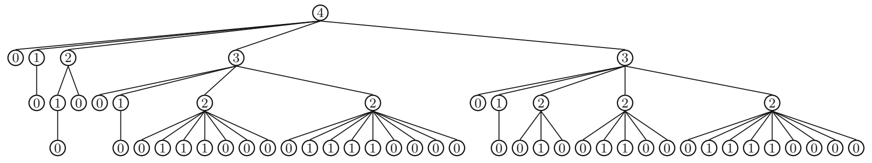
For an illustration of the construction the reader is referred to Fig. 4.

For the other direction, suppose t is a Union-Find tree. By Theorem 2 and Proposition 1, there is a subset $X \subseteq \textsc{children}(t)$ and a mapping $f : Y \to X$ with $Y = \{y_1, \ldots, y_\ell\} = \textsc{children}(t) - X$ such that for the sequence $t_0 = t$, $t_i = \textsc{push}(t_{i-1}, y_i, f(y_i))$ we have that each immediate subtree of t_ℓ is a Union-Find tree and moreover, the root of t_ℓ satisfies the Union condition.

The root of t has rank 4, t_ℓ has to have at least one child having rank 0, 1, 2 and 3 respectively. Since t has exactly one child with rank 0 and rank 1, these nodes has to be in X. This implies that no node gets pushed into the apples at this stage (because the apples have rank 2). Thus, since the apples are *not* Union-Find trees (as they have strictly less rank-0 nodes than positive-rank nodes, cf. Proposition 2), all the apples have to be in Y. Apart from the apples, t has exactly one depth-one node of rank 2 (which happens to be a root of a Union tree), thus this node has to stay in X as well. Moreover, we cannot push the baskets as they have the maximal rank 3, hence they cannot be pushed.

Thus, we have to push all the apples, and we can push apples only into baskets (as exactly the baskets have rank greater than 2). Let $x \in X$ be a basket node, let t' stand for $t_\ell|_x$ and let $\{y'_1, \ldots, y'_j\} \subseteq Y$ be the set of those apples that get pushed into x during the operation. Then, the total number of nodes having rank 0 in t' is $H + 2 + j$ (j of them coming from the apples and the other ones coming from the basket) while the total number of nodes having a positive rank is $2 + j + A$ where A is the total weight of the apples in $\{y'_1, \ldots, y'_j\}$. Applying


(a) Apples of size 3 and 4 are pushed into the first basket, apples of size 1, 2 and 4 are pushed into the second basket.


(b) The apples get filled from the baskets' surplus rank-0 leaves.

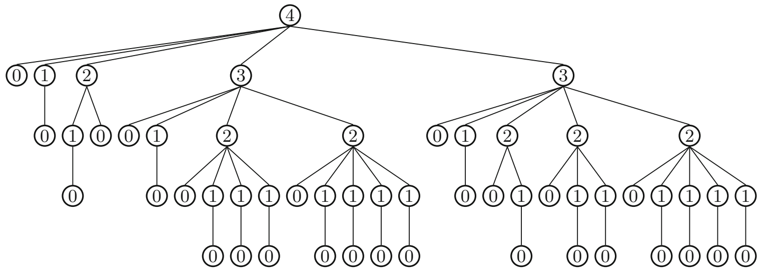

(c) The filling of the apples is pushed a level deeper and we have a Union tree.

Fig. 4. Pushing t of Fig. 3 into a Union tree according to the solution $3+4=1+2+4$ of the PARTITION instance.

Proposition 2 we get that $A \leq H$ for each basket. Since the total weight of all apples is $H \cdot k$ and each apple gets pushed into exactly one basket, we get that $A = H$ actually holds for each basket. Thus, $\mathcal{I}$ is a positive instance of the PARTITION problem. □

Theorem 3. *The recognition problem of Union-Find trees is* **NP**-*complete.*

Proof. By Proposition 3 we get **NP**-hardness. For membership in **NP**, we make use of the characterization given in Theorem 2 and that the possible number of pushes is bounded above by n^2: upon pushing x below y, the depth of x and its descendants increases, while the depth of the other nodes remains the same. Since the depth of any node is at most n, the sum of the depths of all the nodes is at most n^2 in any tree. Hence, it suffices to guess nondeterministically a sequence $t = t_0 \vdash t_1 \vdash \ldots \vdash t_k$ for some $k \leq n^2$ with t_k being a Union tree (which also can be checked in polynomial time). □

5 Conclusion, Future Directions

We have shown that unless $\mathbf{P} = \mathbf{NP}$, there is no efficient algorithm to check whether a given tree is a valid Union-Find tree, assuming union-by-rank strategy, since the problem is **NP**-complete, complementing our earlier results assuming union-by-size strategy. A very natural question is the following: does there exist a merging strategy under which the time complexity remains amortized almost-constant, and at the same time allows an efficient recognition algorithm? Although this data structure is called "primitive" in the sense that it does not really need an automatic run-time certifying system, but we find the question to be also interesting from the mathematical point of view as well. It would be also an interesting question whether the recognition problem of Union-Find trees built up according to the union-by-rank strategy is still **NP**-complete if the nodes of the tree are not tagged with the rank, that is, given a tree without rank info, does there exist a Union-Find tree with the same underlying tree?

References

1. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 208–217. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45138-9_15
2. Cai, L.: The recognition of Union trees. Inf. Process. Lett. **45**(6), 279–283 (1993)
3. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Albers, S., Marion, J.-Y. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science. STACS 2009, vol. 3 of LIPIcs, pp. 289–300. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
4. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education, New York (2001)
5. Crochemore, M., Iliopoulos, C.S., Pissis, S.P., Tischler, G.: Cover array string reconstruction. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 251–259. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13509-5_23
6. Duval, J.-P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays and validation of string matching automata. RAIRO - Theor. Inf. Appl. **43**(2), 281–297 (2009)
7. Duval, J.-P., Lefebvre, A.: Words over an ordered alphabet and suffix permutations. Theor. Inf. Appl. **36**(3), 249–259 (2002)
8. Duval, J.-P., Lecroq, T., Lefebvre, A.: Border array on bounded alphabet. J. Autom. Lang. Comb. **10**(1), 51–60 (2005)
9. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing, STOC 1989, pp. 345–354. ACM, New York (1989)
10. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM **7**(5), 301–303 (1964)
11. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
12. Gawrychowski, P., Jez, A., Jez, Ł.: Validating the Knuth-Morris-Pratt failure function, fast and online. Theory Comput. Syst. **54**(2), 337–372 (2014)
13. Gelle, K., Iván, S.: Recognizing Union-Find trees is NP-complete. CoRR, abs/1510.07462 (2015)
14. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 422–433. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00982-2_36

15. Tomohiro, I., Inenaga, S., Bannai, H., Takeda, M.: Verifying and enumerating parameterized border arrays. Theoret. Comput. Sci. **412**(50), 6959–6981 (2011)
16. Tomohiro, I., Inenaga, S., Bannai, H., Takeda, M.: Inferring strings from suffix trees and links on a binary alphabet. Discr. Appl. Math. **163**(Part 3), 316–325 (2014)
17. Knight, K.: Unification: a multidisciplinary survey. ACM Comput. Surv. **21**(1), 93–124 (1989)
18. Kucherov, G., Tóthmérész, L., Vialette, S.: On the combinatorics of suffix arrays. Inf. Process. Lett. **113**(22–24), 915–920 (2013)
19. Weilin, L., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a border array in linear time. J. Comb. Math. Comb. Comput. **42**, 223–236 (2000)
20. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
21. Starikovskaya, T., Vildhøj, H.W.: A suffix tree or not a suffix tree? J. Discr. Algorithms **32**, 14–23 (2015)
22. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)