Information Processing Letters 131 (2018) 7–14

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Recognizing Union-Find trees is **NP**-complete ☆

Kitti Gelle, Szabolcs Iván *

University of Szeged, Hungary

ARTICLE INFO

Article history:
Received 26 October 2015
Received in revised form 11 September 2017
Accepted 12 November 2017
Available online 14 November 2017
Communicated by A. Tarlecki

Keywords:
Union-Find trees
Complexity
NP-completeness
Data structures
Union-by-size

ABSTRACT

Disjoint-Set forests, consisting of Union-Find trees, are data structures having a widespread practical application due to their efficiency. Despite them being fundamental, no exact structural characterization of these trees is known (such a characterization exists for Union trees which are constructed without using path compression). In this paper we provide such a characterization by means of a simple PUSH operation and we show that the problem of deciding whether a given tree is a Union-Find tree is **NP**-complete.

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 support 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.

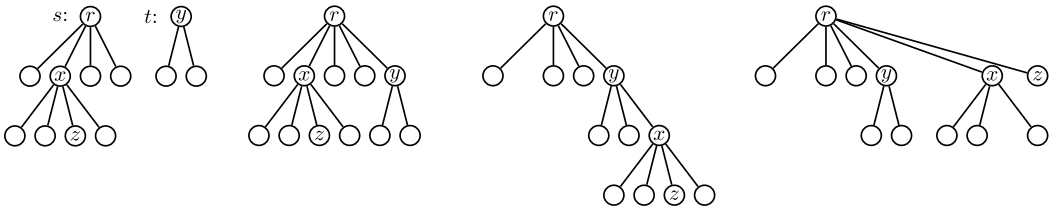
To support these operations, even a linked list representation suffices but to achieve an almost-constant amortized time cost per operation, Disjoint-Set forests are used in practice. In this data structure, sets are represented as directed trees with the edges directed towards the root; the CREATE operation 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 "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 the size of the tree is its depth, 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. Doing so, one 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 ex-

☆ Kitti Gelle was supported by the ÚNKP-17-3-I-SZTE-18 New National Excellence Program of the Ministry of Human Capacities. Szabolcs Iván was supported by NKFI grant number K108448.

* Corresponding author.
E-mail address: szabivan@inf.u-szeged.hu (S. Iván).

https://doi.org/10.1016/j.ipl.2017.11.003
0020-0190/



(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.

tremely fast growing (not primitive recursive) Ackermann function. As $\alpha(n) \leq 5$ for each practical value of n (say, below 2^{65535}), hence it can be seen an amortized almost-constant time cost [22]. Since it's proven [9] that *any* data structure 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 points 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, due to their simple implementation, Union-Find trees are sometimes regarded as being "primitive" 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-size strategy is used (and leave open the case of the union-by-rank strategy). This confirms the statement from [2] that the problem "seems to be much harder" than recognizing Union trees (which in turn can be done in low-degree polynomial time).

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 [15,19,8,14,15], suffix arrays [1,7,18], KMP tables [6,12], prefix tables [3], cover arrays [5], and directed acyclic word and subsequence graphs [1]. Union-Find trees are fundamental data structures used in the context of dynamic graph algorithms [23].

2. Notation

A *tree* is a tuple $t = (V_t, \text{ROOT}_t, \text{PARENT}_t)$ with V_t being the finite set of its *nodes*, $\text{ROOT}_t \in V_t$ its *root* and $\text{PARENT}_t : (V_t - \{\text{ROOT}_t\}) \to V_t$ mapping each non-root node to its *parent* (so that the graph of PARENT_t is a directed acyclic graph, with edges being directed towards the root).

For a tree t and a node $x \in V_t$, let $\text{CHILDREN}(t, x)$ stand for the set $\{y \in V_t : \text{PARENT}_t(y) = x\}$ of its *children* and $\text{CHILDREN}(t)$ stand as a shorthand for $\text{CHILDREN}(t, \text{ROOT}_t)$, the set of depth-one nodes of t. Two nodes are *siblings* in t if they have the same parent. Also, let $x \preceq_t y$ denote that x is an *ancestor* of y in t, i.e. $x = \text{PARENT}_t^k(y)$ for some $k \geq 0$. Let $\text{SIZE}(t, x) = |\{y \in V_t : x \preceq_t y\}|$ stand for the number of *descendants* of x (including x itself). Let $\text{SIZE}(t)$ stand for $\text{SIZE}(t, \text{ROOT}_t)$, the number of nodes in the tree t. For $x \in V_t$, let $t|_x$ stand for the *subtree* $(V_x = \{y \in V_t : x \preceq_t y\}, x, \text{PARENT}_t|_{V_x})$ of t rooted at x. When $x, y \in V_t$, we say that x is *lighter* than y (or y is *heavier* than x) in t if $\text{SIZE}(t, x) < \text{SIZE}(t, y)$.

Two operations on trees are that of *merging* and *collapsing*. Given two trees $t = (V_t, \text{ROOT}_t, \text{PARENT}_t)$ and $s = (V_s, \text{ROOT}_s, \text{PARENT}_s)$ with V_t and V_s being disjoint, their merge $\text{MERGE}(t, s)$ (in this order) is the tree $(V_t \cup V_s, \text{ROOT}_t, \text{PARENT})$ with $\text{PARENT}(x) = \text{PARENT}_t(x)$ for $x \in V_t$, $\text{PARENT}(\text{ROOT}_s) = \text{ROOT}_t$ and $\text{PARENT}(y) = \text{PARENT}_s(y)$ for each non-root node $y \in V_s$ of s. Given a tree $t = (V, \text{ROOT}, \text{PARENT})$ and a node $x \in V$, the tree $\text{COLLAPSE}(t, x)$ is the tree $(V, \text{ROOT}, \text{PARENT}')$ with $\text{PARENT}'(y) = \text{ROOT}$ if y is a non-root ancestor of x in t, and $\text{PARENT}'(y) = \text{PARENT}(y)$ otherwise. For examples, see Fig. 1.

The class of *Union trees* is the least class of trees satisfying the following two conditions: every *singleton tree* (having exactly one node) is a Union tree, and if t and s are Union trees with $\text{SIZE}(t) \geq \text{SIZE}(s)$, then $\text{MERGE}(t, s)$ 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 s are Union-Find trees with $\text{SIZE}(t) \geq \text{SIZE}(s)$, 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'll frequently sum the size of "small enough" children of nodes, so we introduce one more shorthand: for a tree t, a node x of t, and a threshold $W \geq 0$, let $\text{SUMSIZE}(t, x, W)$ stand for $\sum\{\text{SIZE}(t, y) : y \in \text{CHILDREN}(t, x), \text{SIZE}(t, y) \leq W\}$. We say that a node x of a tree t *satisfies the Union condition* if for each child y of x we have $\text{SUMSIZE}(t, x, W) \geq W$ where $W = \text{SIZE}(t, y) - 1$. Oth-