

Worst-case and Amortised Optimality in Union-Find

(Extended abstract)

Stephen Alstrup*, Amir M. Ben-Amram[†], Theis Rauhe[‡]

Abstract

We study the interplay between worst-case and amortised time bounds for the classic Disjoint Set Union problem (Union-Find). We ask whether it is possible to achieve optimal worst-case and amortised bounds *simultaneously*. Furthermore we would like to allow a tradeoff between the worst-case time for a query and for an update. We answer this question by first providing lower bounds for the possible worst-case time tradeoffs, as well as lower bounds which show where in this tradeoff range optimal amortised time is achievable. We then give an algorithm which tightly matches both lower bounds simultaneously. The lower bounds are provided in the cell-probe model as well as in the algebraic real-number RAM, and the upper bounds hold for a RAM with logarithmic word size and a modest instruction set.

Our lower bounds show that for worst-case query and update time t_q and t_u respectively, one must have $t_q = \Omega(\log n / \log t_u)$, and only for $t_q \geq \alpha(m, n)$ can this tradeoff be achieved simultaneously with the optimal amortised time of $\Theta(\alpha(m, n))$. Our

*DIKU, Comp. Sci. Dept., University of Copenhagen, Denmark. E-mail: stephen@diku.dk. Part of this work was done while visiting BRICS and Lund University.

[†]The Academic College of Tel-Aviv Yaffo, Israel. E-mail: amir-ben@server.mta.ac.il. Part of this work was done while visiting the dept. of Computer Science at the University of Copenhagen.

[‡]BRICS, Comp. Sci. Dept., University of Aarhus, Denmark. Email: theis@brics.dk. Part of this work was done while visiting Fields Institute of Toronto. The work was partially supported by the ESPRIT Long Term Research Programme of the EU, project number 20244 (ALCOM-IT)

algorithm can match this tradeoff for any desired choice of t_u . For faster worst-case query time, the amortised cost becomes larger but is also matched by our lower bounds.

1 Introduction

The efficiency of algorithms for data-structure maintenance is traditionally studied using two different criteria. One is the worst-case time for an operation on the data structure, which is important for ensuring response time in on-line applications. Another is the amortised time, which is the decisive criterion in an off-line situation or when some program includes a whole sequence of data-structure operations. Our purpose in this work is to examine if when insisting on optimal worst-case performance it is still possible to optimise the amortised cost as well. We study this question for a data-structure problem that has received a lot of attention with respect to both worst-case and amortized efficiency, the Union-Find problem.

The Union-Find (UF) problem is to carry out a sequence of the following two types of operations, on a data structure which initially represents n disjoint singleton sets.

Find(x) : Return the name of the set in which element x is contained.

Union(A, B) : Join the sets A and B (destroying sets A and B), and relate a set name to the resulting set.

We refer the reader to Galil and Italiano [11] for a broad survey on the UF problem and several of its variants and their complexity under different models. In the context of RAM algorithms we can assume the element names to be $1, 2, \dots, n$.

Algorithms for the UF problem have been extensively studied under both criteria: single operation worst-case cost (henceforth: worst case) and amortised cost. Currently known algorithms (considered in more detail below) optimise either the worst-case cost or the amortised cost, but not both. In addition, for each of the two criteria, there are studies on the possible tradeoff between the complexity of updates (union operations) and queries (find).

In this work we ask whether it is possible for an algorithm to be optimal in both criteria, amortised *and* worst-case, while allowing optimal tradeoff between update and query costs.

Our answer is almost always: our lower bounds show that for worst-case query and update time t_q and t_u respectively, $t_q = \Omega(\log n / \log t_u)$, and only for $t_q \geq \alpha(m, n)$ this tradeoff can be achieved simultaneously with the optimal amortised time of $\Theta(\alpha(m, n))$. Our algorithm gives the optimal tradeoff for any desired choice of t_u . Before giving fuller details, we give some background.

For lower bounds we use two incompatible computational models. The first is the *cell probe model* with logarithmic word length. In this model, the cost of a computation is the number of memory words accessed, but any computation with these values is free [19, 8]. The second model is the *algebraic real-number RAM* (henceforth: algebraic RAM), where unbounded integers or even real numbers can be manipulated and cost is the number of instructions performed [3]. Every algorithm mentioned in this paper can be implemented on a RAM with logarithmic word size and a modest instruction set, and hence is subject to lower bounds in both models. The algorithms can also be included in the framework of *pointer algorithms* (cf. [4]). The latter is not in the focus of this paper, but is mentioned because the first lower bounds for UF were given in this framework.

The best-known algorithm for the UF problem, using *path compression*, achieves a total running time of $O(n + m\alpha(m, n))$ for n unions and m finds [16, 18]. Here α is a very slow-growing “diagonal inverse” of Ackermann’s function (we remark that its definition is not unique across the literature, although the differences are generally not essential. All results we quote have been adjusted to the definition we give). A different algorithm of similar

amortised time was given by La Poutré [13].

The above running time was proved optimal for certain classes of “pointer algorithms” by Tarjan [17] and La Poutré [12], and later for the cell probe model by Fredman and Saks [7, 8, see also 5], and for the algebraic RAM by Ben-Amram [3]. Among the above proofs all but La Poutré’s assume $m \geq n$. However Tarjan’s proof was extended for all m by Banachowski [2], and the rest extend without any essential change. In this paper m is not restricted.

The path-compression algorithm has *worst-case time* $\Theta(\log n)$. Mehlhorn [14] asked if it was possible to get $o(\log n)$ worst-case time. Blum [6] answered this positively by giving an $O(\log n / \log \log n)$ algorithm, and proved that this is optimal for a class of pointer algorithms; Fredman and Saks [8] gave a matching lower bound for the cell probe model, and Ben-Amram [3] for the algebraic RAM.

We now look more closely at the tradeoff between update time and query time. For every UF algorithm we define four cost measures: t_q , the worst-case time for a query (find); t_u , the worst-case time for an update (union); \bar{t}_q (respectively \bar{t}_u), the amortised cost of a query (resp. update), where we only amortise over operations of the same kind. Finally t_a is the amortized time over all operations (the usual definition).

Blum’s lower bound proof actually relates query cost to update cost. It includes a lemma stating that $t_q = \Omega(\log n / (\log t_u + \log \log n))$. A similar result for the cell probe model follows by extending the Fredman-Saks proof [1]. On the other hand, Blum’s algorithm gives, for arbitrary $k > 1$, the bounds $t_q = O(\log n / \log k)$ and $t_u = O(k + \log n / \log k)$. This was improved by Smid [15] to $t_u = O(k)$ and $t_q = O(\log n / \log k)$. Blum’s lower bound matches Smid’s algorithm provided $t_u = \Omega((\log n)^\epsilon)$, $\epsilon > 0$.

La Poutré [13], using techniques similar to Gabow [9, 10], gave algorithms that for arbitrary i achieve $t_q = O(i)$ and $\bar{t}_u = O(a(i, n))$. Here $a(i, n)$ is the i ’th row inverse of Ackermann function, e.g., $a(2, n) = \log^* n$. La Poutré’s lower bound paper [12] also shows that this tradeoff is optimal for the class of algorithms considered. This means that the total amortised time of $O(\alpha(m, n))$ can not be achieved in that class if we insist on constant query time, because this makes the unions too expensive.

The lower bounds we give show that for both the cell probe model and the algebraic RAM the following hold:

- $t_q = \Omega(\log n / \log t_u)$, matching Smid's upper bound for all values of t_u .
- $\bar{t}_u = \Omega(a(i, n))$, for a certain $i = O(t_q)$. This matches the upper bound by La Poutré.

The algorithm presented in this paper (Section 5) matches both lower bounds simultaneously. Namely, given any parameter k we can obtain the simultaneous bounds $t_u = O(k)$, $t_q = O(\log n / \log k)$ and $t_a = \Theta(a(t_q, n) + \alpha(m, n))$. As a special case we can achieve $t_q = t_u = O(\log n / \log \log n)$ with amortised complexity $O(\alpha(m, n))$. This two-way optimum is achieved by an algorithm which is almost as simple as the popular path-compression algorithm.

We remark that the space used by all the algorithms considered in this paper is linear in n . Finally we define the Ackermann and inverse Ackermann functions.

$$\begin{aligned} A(i, 0) &= 2 && \text{for } i > 1; \\ A(1, j) &= 2^j && \text{for } j \geq 0; \\ A(i, j + 1) &= A(i - 1, A(i, j)) && \text{for } i > 1, j \geq 0. \end{aligned}$$

$$\begin{aligned} a(k, n) &= \min \{ j > 0 \mid A(k, j) > n \}. \\ \alpha(m, n) &= \min \{ k \mid A(k, \lceil \frac{m}{n} \rceil) > \log n \}. \end{aligned}$$

2 The lower bounds

We give our lower bounds in two models. One is the cell probe model with a word size of b bits, denoted CPROBE(b). The other is the *algebraic RAM*: a random access machine where memory cells contain arbitrary real numbers, and can be addressed by such. Thus there are uncountably many possible addresses, but we assume all memory cells to be initialized to zero, so only finitely many can be non-zero in a finite computation. Programs can access memory using direct or indirect addressing, compare numbers and compute with the operations $\{+, -, \times, /\}$. Every operation costs one time unit. Obviously, lower bounds in this model also apply to the more conventional model that only handles integers.

Theorem 1 *Consider the union-find problem on n elements and let $k \geq 2$. Any CPROBE($\log n$) algorithm, that solves this problem with amortized query cost $\bar{t}_q \leq k$, satisfies $\bar{t}_u = \Omega(a(16k + 1, n))$.*

Theorem 2 *Consider instances of the union-find problem on n elements and let $k \geq 2$. Any algebraic RAM algorithm, that solves all these instances with amortized query cost $\bar{t}_q \leq k$, and uses $O(n^k)$ memory cells, satisfies $\bar{t}_u = \Omega(a(16k + 1, n))$.*

Theorem 3 *Consider instances of the union-find problem on n elements and let $4 \leq k \leq n$. If an algebraic RAM algorithm for union-find solves all such instances with update cost $t_u \leq k$, then there are sequences of unions where the cost of a subsequent find is $\Omega(\log n / \log k)$.*

Theorem 4 *Consider instances of the union-find problem on n elements and let $0 < \varepsilon \leq 1$. If a CPROBE(b) algorithm for union-find solves such instances with update cost $t_u \leq k$, where $(\frac{15b}{\log n})^\varepsilon \leq k \leq n$, then there are sequences of unions where the cost of a subsequent find is $\Omega(\frac{\varepsilon}{\varepsilon+1} \log n / \log k)$. Note that for $b = O(\log n)$ the result holds for all k greater than some constant.*

Remark: in Theorem 2, the restriction on space can be omitted if worst-case query cost rather than amortised is considered. With amortised query cost it is necessary because otherwise, during very long sequences of queries we could record every answer and decrease \bar{t}_q to a constant.

All the above theorems can be proved within the framework of Ben-Amram and Galil [5], which is a generalization of the ideas from Fredman and Saks [7, 8]. The results for cell probe can also be proved via a modification of the Fredman-Saks technique, in the style of [1]. In this extended abstract we present the first proof method. Ben-Amram and Galil separate the analysis required to prove a lower bound into three parts. In one part we analyse a characteristic of the problem called *problem variability* (PV). In the second we analyse a characteristic of the computational model called *output variability* (OV). The third part of the proof plugs bounds on PV and OV into the so-called Main Theorem to produce lower bounds.

Ben-Amram and Galil explicitly compute bounds on PV and OV which underly the original proofs

of Fredman and Saks. Their statement of the Main Theorem also clarifies that the lower bound results are in essence tradeoffs between update and query costs. The above four theorems are all proved using this theorem. Specifically:

- For Theorem 1 we use PV and OV from [5].
- For Theorem 2 we use PV from [5] and OV for the algebraic RAM from [3].
- For Theorem 3 we use PV from [5] and OV for the algebraic RAM from [3].
- For Theorem 4 we use PV from [5] and a new bound on OV for the cell probe model.

We consider the last proof to be the most interesting, since it contains a new analysis of the cell probe model. In order to follow this analysis it suffices to have the definition of OV. We give this definition and the new analysis in Section 3. Section 4 recall the Main Theorem and proves Theorem 4. Proofs of the other theorems will be given in the full paper.

3 Output variability for the cell probe model

We start by briefly recalling the definition of Output Variability. Consider a *query program* that receives a *query argument* in the range $1 \dots n$, and using data stored in memory produces an answer in a predetermined range $1 \dots m$ (in our problem $m = n$).

A *memory image* M is a sequence of integers $M(0), M(1), \dots$ which specifies the contents of memory cells. Let $Q(i, M)$ denote the result of a query on i given the memory image M . The vector of results $(Q(1, M), \dots, Q(n, M))$ will be denoted by $\overline{Q}(M)$.

The *output variability* of a model is the function $OV(w, x, q)$ defined as follows. Let M be a memory image that contains a data structure whose cost is w (we have to define our measure of data-structure cost). Let M^x be the set of memory images obtained from M by modifying at most x cells. Let Q be a query program such that for all arguments i and memory images $X \in M^x$ the computation of $Q(i, X)$ costs at most q . More generally, we consider a q -truncated program: this is a program Q obtained from a real query program Q' by forcing

it to halt and output a “dummy” answer (say, zero) whenever its run would originally cost more than q . We define $OV(w, x, q)$ to be the supremum, over all such memory images M and programs Q , of the cardinality of the set of vectors $\{\overline{Q}(X) | X \in M^x\}$.

The parameter w is not used in our bound on OV for the cell probe model. However it is included in the definition for the sake of generality; indeed it is important for the RAM bounds.

Theorem 5 *For the cell probe model CPROBE(b), and $x > 1$, $OV(w, x, q) \leq (2^b n q)^x$.*

Proof: Fix a (truncated) query program Q and a memory image M . We can represent the query program (for a fixed query argument) as a decision tree with 2^b -ary branching that represents the results of memory reads. The height of the tree is at most q since a query’s cost is at least the number of memory reads. Let N be a memory image which was obtained from M by modifying a set X of cells where $|X| \leq x$. Let $a(i, N, k)$ be the address accessed by the query $Q(i, N)$ in the k ’th step, $k \leq q$. We consider a parallel execution of all queries $Q(i, N)$, for $1 \leq i \leq n$. We model the parallel execution as a single decision tree, in the following way: the first node asks which of the cells with addresses in $A_1 = \{a(i, N, 1) | 1 \leq i \leq n\}$ belongs to X . Each result of this question is a set S_1 . Let $j_1 = |S_1|$. Clearly, the value of S_1 and even its cardinality depend on X . Note that since this is the first step, the addresses accessed cannot depend on N , so A_1 is a fixed set of at most n addresses and the number of possibilities for S_1 is $\binom{|A_1|}{|S_1|} \leq \binom{n}{j_1}$. For each set S_1 , the contents of cells not in S_1 is fixed (given by M), so the second node models branching according to the contents of the S_1 -cells. The branching degree is 2^{bj_1} where $j_1 = |S_1|$. These two branching levels together describe the first step of the n queries. The second step is modelled similarly; it accesses a set of at most n addresses A_2 . Their identities are completely determined by the branches taken in the first step, so when considering a specific node it is a fixed set of n addresses. Thus, again, we have at most $\binom{n}{j_2}$ possible sets S_2 , and then a branching degree of 2^{bj_2} . And so on.

What can we say about the sets S_1, S_2, \dots ? Because every node in the decision tree “remembers”

all the branches taken to reach it, it is never necessary to read a memory cell twice. That is, we can assume the sets A_k to be mutually disjoint along each computation path. This implies that the sets S_k are also disjoint. But they are all subsets of X . Hence, *along each computation path*, we have $\sum |S_k| \leq x$.

The output variability $OV(w, x, q)$ is just the number of leaves in the above decision tree. We immediately obtain the bound

$$\begin{aligned} OV(w, x, q) &\leq \sum_{j_1 + \dots + j_q \leq x} \binom{n}{j_1} 2^{bj_1} \binom{n}{j_2} 2^{bj_2} \dots \binom{n}{j_q} 2^{bj_q} \\ &\leq 2^{bx} \sum_{j_1 + \dots + j_q \leq x} \binom{n}{j_1} \binom{n}{j_2} \dots \binom{n}{j_q} \\ &= 2^{bx} \sum_{j \leq x} \binom{nq}{j} \leq 2^{bx} (nq)^x. \end{aligned}$$

4 Proof of Theorem 4

We first recall the other components of the proof method. We omit some details which are not essential for the presentation and can be found in [5].

Let \mathcal{U} be an *update scheme*, a set of sequences of update operations, in which all sequences have the same length, and they are all divided in a fixed manner into subsequences called *rounds*. We focus on a subset of these sequences obtained by fixing the operations preceding round i and continuing in all possible ways (within \mathcal{U}) up to round j . To each such sequence we associate the *correct answer vector* (a_1, \dots, a_n) where a_k is the answer that should be given by a query on k following the given updates. For an arbitrary n -element vector v and a fixed fraction δ consider the set of correct answer vectors whose Hamming distance from v does not exceed δn (the *close vectors*). $PV_{\mathcal{U}, \delta}(i, j)$ is essentially a lower bound on the ratio of the number of correct answer vectors to the number of close vectors. A large $PV_{\mathcal{U}, \delta}(i, j)$ indicates that correct answer vectors are dispersed in the space with Hamming metric, which indicates some sort of hardness in the problem.

In the main theorem we consider an *epoch scheme*. For our purposes this can be simply defined by a set of indices $1 = j_1 < j_2 < \dots \leq j$, where \mathcal{U} contains j rounds. These indices partition the sequences in \mathcal{U} into epochs where epoch e comprises update rounds $j_e, \dots, j_{e+1} - 1$.

The following is a slightly trimmed-down version of the Main Theorem from [5].

Theorem 6 *Consider an update scheme \mathcal{U} , a corresponding epoch scheme \mathcal{E} of q epochs, constants $\delta > 0$ and $c < 1$ and parameters x_e and w such that*

$$OV(w, x_e, q) \leq cPV_{\mathcal{U}, \delta}(j_e, j)$$

for all epochs $1 \leq e \leq q$. Assume that for all update sequences σ from \mathcal{U} , the following conditions hold: (i) throughout the execution of σ the data-structure cost is bounded by w . (ii) at most x_e memory cells are written subsequent to epoch e . Then $\delta(1 - c)q$ is a lower bound on the worst-case cost of a query that follows an update sequence from \mathcal{U} .

The update scheme is taken directly from the worst-case lower bound of [8]. It contains $\frac{1}{2} \log n$ rounds, where the unions of round k pair sets of size 2^{k-1} to sets of size 2^k . Hence round k contains $n[k] = n/2^{k-1}$ unions. For this update scheme it is shown in [5] that

$$PV_{\mathcal{U}, \delta}(i, j) \geq 8^{-n[i]} n^{\frac{n[i]}{4}}.$$

To apply the main theorem, we choose q to be $\frac{1}{2(1+\epsilon-1)} \log n / \log k$. Our *epoch scheme* comprises q epochs of length $\beta = \left\lceil \frac{\frac{1}{2} \log n}{q} \right\rceil$ (the last one may be shorter). Recall that the cost of a union is assumed to be bounded by k ; summing over the epochs subsequent to epoch e we obtain

$$\sum_{i=e\beta+1}^{\frac{1}{2} \log n} \frac{n}{2^i} k < \frac{n[j_e]}{2^\beta} k \leq \frac{n[j_e]}{k^{1/\epsilon}}.$$

so we let $x_e = n[j_e]/k^{1/\epsilon}$, satisfying condition (ii) in the theorem. Condition (i) is irrelevant because our bound on OV does not depend on w . By Theorem 5

$$\begin{aligned} OV(w, x_e, q) &\leq (2^b nq)^{x_e} \leq n^{((b/\log n) + 2)x_e} \\ &\leq n^{3(b/\log n)(n[j_e]/k^{1/\epsilon})}. \end{aligned}$$

Now

$$\begin{aligned} \frac{OV(w, x_e, q)}{PV_{\mathcal{U}, \delta}(j_e, j)} &< n^{3(b/\log n)(n[j_e]/k^{1/\epsilon})} 8^{n[j_e]} n^{-\frac{n[j_e]}{4}} \\ &= \left(8n^{\frac{3(b/\log n) - \frac{1}{4}}{k^{1/\epsilon}}} \right)^{n[j_e]}. \end{aligned}$$

For $k \geq (15b/\log n)^\varepsilon$, this expression tends to zero as $n \rightarrow \infty$, hence for n large enough it becomes less than $c = \frac{1}{2}$. The main theorem now states that the average cost of the next query (hence its worst-case cost) is at least $\delta(1-c)q = q/8 = \Omega(\frac{\varepsilon}{\varepsilon+1} \log n / \log k)$.

5 Upper bounds

Theorem 7 *Let $k > 1$; UF can be solved*

- (a) *with $t_u = O(k)$, $t_a = O(\alpha(m, n))$ and $t_q = O(\log n / \log k + \log k)$.*
- (b) *with $t_u = O(k)$, $t_q = O(\log n / \log k)$ and $t_a = O(\log k + \alpha(m, n))$.*

We remark that this theorem is based on rather simple algorithms. Part (a) with $k = \log n / \log \log n$ gives both amortised-cost and worst-case optimum when union and find are not separated. If we desire to bound the query time by a given i we choose k according to $i = \log n / \log k$, and for $\log k \leq \sqrt{\log n}$ we obtain the optimal results. For a larger k we have to switch to Part (b) which has the right query time but not the amortised, due to costly unions. Therefore a more complicated algorithm is needed in this case which we obtain below by combining Part (b) with techniques of Gabow and La Poutré.

We give the algorithms for parts (a) and (b) at the same time, since the second only differs by using an additional compression.

The algorithm is a slightly modified version of the standard UF algorithm [17]. Sets are represented as rooted trees, where each set element is a tree node and the root node is used to identify the set. Thus, to execute a find we follow pointers from the element specified to the root of its tree. This is augmented with path compression, meaning that all the pointers in this path are redirected to the root, thus achieving amortised complexity $O(\alpha(m, n))$ together with union by size or rank [18]. Here we show that by performing some of the compressions at unions, instead of just at the find operation, we can limit the worst-case complexities as stated in Theorem 7. The additional compression is done by the procedure $1compress(S, r)$ which links all the nodes in S directly to the root r . With each node v we maintain a number called $rank(v)$, and with each root r

we keep in $ranksize(r)$ the number of nodes in the tree with root r , of the same rank as r . Initially a singleton set with node v is assigned the values 0 and 1 respectively for $rank(v)$ and $ranksize(v)$.

We first describe the implementation of union to give Theorem 7(a).

Algorithm 1 *For union(v, w) assume w.l.o.g. that $rank(w) \leq rank(v)$ and if $rank(w) = rank(v)$ then $ranksize(w) \leq ranksize(v)$. The operation makes w a child of v . Let S_1 be the set of nodes in the tree of w before the union, whose rank equals $rank(w)$. Let S_2 be the set of nodes (besides v) in the united tree whose rank equals $rank(v)$.*

- **If $rank(w) < rank(v)$:** *after linking w to v we compress the node set S_1 , $1compress(S_1, v)$.*
- **Otherwise $rank(w) = rank(v)$.** *Update $ranksize(v)$. If $ranksize(v) \geq k$ compress all the nodes of $rank(v)$, $1compress(S_2, v)$, and increment $rank(v)$.*

We assume that $1compress(S, v)$ is implemented in time $O(|S|)$; the technical details are omitted from this abstract. Algorithm 2, to achieve theorem 7(b), is identical to Algorithm 1 except that S_1 is also compressed in the second case (unless S_2 is). Hence, after a union all nodes at $rank(v)$ always point to v . The additional compression performed by Algorithm 2 will be called $2compress$.

First we give some properties for the above algorithm(s).

Lemma 8 *In the above algorithms,*

- (a) *the amortised time of $1compress$ is $O(1)$.*
- (b) *$t_u = O(k)$.*
- (c) *The amortised time of $2compress$ is $O(\log k)$.*
- (d) *The maximum rank is $\log n / \log k$.*
- (e) *The distance from a node of the same rank as the root to the root is at most $\log k$.*

Proof:

- (a) Each node is at most $1compress$ once.
- (b) Since the maximum number of nodes of a given rank is k at most $2k$ nodes may be compressed in a single union operation.
- (c) Each node is only $2compress$ at one rank. And when it is, it belongs to the smallest of two united sets of nodes at this rank. A node can at most $O(\log k)$ times be the in the smallest set, when sets are not allowed to exceed size k .

(d) Let ρ be the maximal rank in a tree. We associate with each node of rank ρ a set of *witness nodes* of cardinality k^ρ from that tree. Different nodes have disjoint sets of witnesses. The existence of these sets immediately yields the desired bound on the maximal rank, since a tree contains at most n nodes.

We construct the witness sets while performing unions (this construction is part of the analysis and is not actually carried out by the algorithm). Initially all the nodes are at rank 0 and are their own witnesses. When a union is performed that does not modify ranks, every node keeps its witness set. When a union increments the rank of the root from i to $i + 1$ there are at least k nodes of rank i in the tree; each has its own set of k^i witnesses. The union of these sets, of size k^{i+1} , becomes the witness set for the root which is now the only node at rank ρ .

(e) is proved by the same argument as (c). ■

Proof: [of Theorem 7] The above lemma establishes the bounds on t_u and \bar{t}_u for both parts of Theorem 7. The bound on t_q depends on whether *2compress* is used or not. If we use *2compress*, a find operations only has to follow a single pointer at every rank, otherwise in the last rank it may have to follow up to $\log k$ pointers (in both algorithms all ranks but the last have been compressed). Thus using *2compress*, we have $t_q = O(\log n / \log k)$, otherwise a find takes additional $O(\log k)$ time. To complete the proof we show that m find operations take $O(n + m\alpha(m, n))$ time. In case of Algorithm 2, this proof is identical to that in [18] regarding union by rank. For Algorithm 1, the last rank in a tree has to be treated separately; if we focus on nodes of the last rank we see that they undergo union by size where again a proof from [18] can be used. ■

We now recall the results from [13]. La Poutré presents a series of algorithms that we denote \mathcal{A}_i , and a combined algorithm \mathcal{A}_{opt} , with the following features.

Fact 9 *In \mathcal{A}_i the total time of all unions is $O(na(i, n))$ while the worst-case time for a union is $O(n)$ and for a find $O(i)$.*

Fact 10 *In \mathcal{A}_{opt} , the total time for n unions and m finds is $O(n + m\alpha(m, n))$, while the f^{th} find has worst case complexity $O(\alpha(f, n))$, and unions have worst-case complexity $O(n)$.*

We remark that these algorithms, too, maintain sets as trees such that the set name is in the root. This makes it easy to combine them with our algorithms above to obtain our final upper bound.

Theorem 11 *Let $k > 1$. UF can be solved with an optimal worst-case tradeoff, $t_u = O(k)$ and $t_q = O(\log n / \log k)$, simultaneously with optimal amortised complexity $t_a = O(\alpha(m, n) + a(t_q, n))$.*

Proof: We use one of the algorithms from [13] to maintain trees that contain up to $\log k$ elements. Roots of such trees are maintained by our Algorithm 2. Theorem 7(b) shows that the union/find operations in this structure already have the desired complexity, except for the additional amortised complexity of $O(\log k)$. However since every element of this structure represents the result of about $\log k$ unions, this added cost amortises to a constant.

It remains to show that the operations on the small trees are efficient enough and here we distinguish two cases. If $t_q \geq \alpha(n, n)$ we use Fact 10. Since the maximal size of the small trees is $\log k$ the worst-case complexity $t_u = O(k)$ for the union operation is not exceeded and since the worst-case query time for the f^{th} find is $\alpha(f, n) \leq \alpha(n, n)$ it does not exceed the desired complexity for a find operation. The contribution of this structure to the amortised complexity is $O(\alpha(m, n))$ which is fine.

For the case $t_q < \alpha(n, n)$, let us assume that the number of finds m is known in advance. Then if $\alpha(m, n) < t_q < \alpha(n, n)$ we use Fact 9 for the small trees, selecting the algorithm $\mathcal{A}_{\alpha(m, n)}$. We obtain the total amortised complexity $O(na(\alpha(m, n), n) + m\alpha(m, n)) = O(m\alpha(m, n))$, hence $t_a = O(\alpha(m, n))$. The worst-case costs are obviously good enough.

If $\alpha(m, n) > t_q$ we maintain the small trees with \mathcal{A}_{t_q} . This increases the amortised complexity with $O(a(t_q, n))$ per union while the worst-case costs are as before.

In the journal version of this paper we will show how to use a rebuilding technique described by Poutré [13] to handle the situations where m is not known in advance. ■

To conclude we remark that in the algorithms we described, it is possible to control the worst-case costs of unions and finds, but not their amortised costs; we only claimed that the combined amortised cost t_a is optimal. A further variant of our method (not included in this abstract) allows both the worst-case costs and the amortised costs to be chosen independently along their respective optimal tradeoff curves, as long as they do not contradict (\bar{t}_u must obviously be bounded by t_u , and \bar{t}_q by t_q).

References

- [1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th Symp. Found. of Comp. Sci. (FOCS)*, pages 534–543, 1998.
- [2] L. Banachowski. A complement to Tarjan’s result about the lower bound on the complexity of the set union problem. *Information Processing Letters*, 11:59–65, 1980.
- [3] A. M. Ben-Amram. On the power of random access machines. PhD thesis, 1995.
- [4] A. M. Ben-Amram. What is a “pointer machine”? *SIGACT News*, 26(2):88–95, 1995. See also <http://www.diku.dk/research-groups/topps/bibliography/1998.html#D-351>.
- [5] A. M. Ben-Amram and Z. Galil. A generalization of a lower-bound technique due to Fredman and Saks. submitted for publication, 1998.
- [6] N. Blum. On the single operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.*, 15:1021–1024, 1986.
- [7] M. L. Fredman. On the cell probe complexity of the set union problem. Technical Report TM-ARH-013-570, Bell Communications Research, 1989.
- [8] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Symp. Theory of Computing (STOC)*, pages 345–354, 1989.
- [9] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Symp. Found. of Comp. Sci. (FOCS)*, pages 90–100, 1985.
- [10] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st Symp. on Discrete Alg. (SODA)*, pages 434–443, 1990.
- [11] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comp. Surveys*, 23(3):319, 1991.
- [12] J. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. *Journal of Computer and Systems Sciences*, 52(1):87–99, 1996. See also STOC’90.
- [13] J. A. La Poutré. New techniques for the union-find problem. In D. Johnson, editor, *Proc. 1st Symp. on Discrete Alg. (SODA)*, pages 54–63, 1990.
- [14] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [15] M. Smid. A data structure for the union-find problem having good single-operation complexity. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 1, 1990.
- [16] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
- [17] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and Systems Sciences*, 18(2):110–127, 1979.
- [18] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.
- [19] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.