Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure

Md. Mostofa Ali Patwary[1], Jean Blair[2], and Fredrik Manne[1]

[1] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
Mostofa.Patwary@ii.uib.no, fredrikm@ii.uib.no
[2] Department of EE and CS, United States Military Academy,
West Point, NY 10996, USA
Jean.Blair@usma.edu

Abstract. The disjoint-set data structure is used to maintain a collection of non-overlapping sets of elements from a finite universe. Algorithms that operate on this data structure are often referred to as Union-Find algorithms. They are used in numerous practical applications and are also available in several software libraries. This paper presents an extensive experimental study comparing the time required to execute 55 variations of Union-Find algorithms. The study includes all the classical algorithms, several recently suggested enhancements, and also different combinations and optimizations of these. Our results clearly show that a somewhat forgotten simple algorithm developed by Rem in 1976 is the fastest, in spite of the fact that its worst-case time complexity is inferior to that of the commonly accepted "best" algorithms.

Keywords: Union-Find, Disjoint Set, Experimental Algorithms.

1 Introduction

Let U be a set of n distinct elements and let S_i denote a subset of U. Two sets S_1 and S_2 are disjoint if $S_1 \cap S_2 = \emptyset$. A disjoint-set data structure maintains a dynamic collection $\{S_1, S_2, \ldots, S_k\}$ of disjoint sets that together cover the universe U. Each set is identified by a representative x, which is usually some member of the set. The two main operations are to Find which set a given element belongs to by locating its representative element and to replace two existing sets with their Union. In addition, there is a Makeset operation which adds a new element to U as a singleton set.

The underlying data structure of each set is typically a rooted tree represented by a parent function $p(x) \in S_i$ for each $x \in U$; the element in the root of a tree satisfies $p(x) = x$ and is the representative of the set. Then Makeset(x) is achieved by setting $p(x) \leftarrow x$ and the output of Find(x) is the root of the tree containing x. This is found by following x's *find-path*, which is the path of parent pointers from x up to the root of x's tree. A set of algorithms that operate on this data structure is often referred to as a Union-Find algorithm.

This disjoint-set data structure is frequently used in practice, including in application areas such as image decomposition, clustering, sparse matrix

P. Festa (Ed.): SEA 2010, LNCS 6049, pp. 411–423, 2010.

computations, and graph algorithms. It is also a standard subject taught in most algorithms courses.

Early theoretical work established algorithms with worst-case time complexity $\Theta(n + m \cdot \alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [2,3,15,16,17,18], where α is the very slowly growing inverse of Ackermann's function. These theoretically best classical algorithms include a standard UNION method using either LINK-BY-RANK or LINK-BY-SIZE and a FIND operation incorporating one of three standard compression techniques: PATH-COMPRESSION, PATH-SPLITTING, or PATH-HALVING. Other early algorithms either use a different compression technique like COLLAPSING or interleave the two FIND operations embedded in a UNION operation, as was the case with Rem's algorithm and a variant of it designed by Tarjan and van Leeuwen. The worst case time complexity of these variations are not optimal [18].

The current work presents an extensive experimental study comparing the time required to execute a sequence of UNION operations, each with two embedded FIND operations. Altogether, we consider 55 variations; 29 of these had been well studied in the theoretical literature by 1984. We call these the classical algorithms. The remaining 26 variations implement a number of improvements. Our results clearly show that a somewhat forgotten simple algorithm developed by Rem in 1976 [6] is the fastest, in spite of the fact that its worst-case complexity is inferior to that of the commonly accepted "best" algorithms.

Related experimental studies have compared only a few UNION-FIND algorithms, usually in the context of a specific software package. In particular, [10] and [8] compared only two and six UNION-FIND algorithms, respectively, in the context of sparse matrix factorization. The works in [19] and [20] compared eight and three UNION-FIND algorithms, respectively, in the setting of image processing. More recently, [13] compared a classic algorithm with a variation described here in the IPC subsection of Section 2.2. The most extensive previous experimental study was Hynes' masters thesis [9] where he compared the performance of 18 UNION-FIND algorithms used to find the connected components of a set of Erdös-Rényi style random graphs.

2 Union-Find Algorithms

This section outlines, to the best of our knowledge, the primary UNION-FIND algorithms. We also include a number of suggested enhancements designed to speed up implementations of these algorithms. Our presentation is from the viewpoint of its use in finding connected components of a graph $G(V, E)$ as shown in Algorithm 1. In this case, the UNION-FIND algorithm computes a minimal subset $S \subseteq E$ such that S is a spanning forest of G.

Algorithm 1. Use of UNION-FIND

```
1: S ← ∅
2: for each x ∈ V do
3:     MAKESET(x)
4: for each (x, y) ∈ E do
5:     if FIND(x) ≠ FIND(y) then
6:         UNION(x, y)
7:         S ← S ∪ {(x, y)}
```

2.1 Classical Algorithms

Here we discuss the classical UNION techniques and then present techniques for compressing trees during a FIND operation. Finally, we describe classical algorithms that interleave the FIND operations embedded in a UNION along with a compression technique that can only be used with this type of algorithm.

Union techniques. The UNION(x, y) operation merges the two sets containing x and y, typically by finding the roots of their respective trees and then linking them together by setting the parent pointer of one root to point to the other.

Clearly storing the results of the two FIND operations on line 5 and then using these as input to the UNION operation in line 6 will speed up Algorithm 1. This replacement for lines 5–7 in Algorithm 1 is known as QUICK-UNION (QUICK) in [7]. Throughout the remainder of this paper we use QUICK.

Three classic variations of the UNION algorithm center around the method used to LINK the two roots. Let r_x and r_y be the roots of the two trees that are to be merged. Then UNION with NAIVE-LINK (NL) arbitrarily chooses one of r_x and r_y and sets it to point to the other. This can result in a tree of height $O(n)$.

In UNION with LINK-BY-SIZE (LS) we set the root of the tree containing the fewest nodes to point to the root of the other tree, arbitrarily breaking ties. To implement LS efficiently we maintain the size of the tree in the root. For the UNION with LINK-BY-RANK (LR) operation we associate a rank value, initially set to 0, with each node. If two sets are to be merged and the roots have equal rank, then the rank of the root of the combined tree is increased by one. In all other LR operations the root with the lowest rank is set to point to the root with higher rank and all ranks remain unchanged. Note that when using LR the parent of a node x will always have higher rank than x. This is known as the *increasing rank property.* The union algorithm presented in most textbooks uses the QUICK and LR enhancements to implement lines 5–7 of Algorithm 1.

Both LS and LR ensure that the find-path of an n vertex graph will never be longer than $\log n$. The alleged advantage of LR over LS is that a rank value requires less storage than a size value, since the rank of a root in a set containing n vertices will never be larger than $\log n$ [3]. Also, sizes must be updated with every UNION operation whereas ranks need only be updated when the two roots have equal rank. On the other hand LR requires a test before each LINK operation.

Compression techniques. Altogether we describe six classical compression techniques used to compact the tree, thereby speeding up subsequent FIND operations. The term NF will represent a FIND operation with no compression.

Using PATH-COMPRESSION (PC) the find-path is traversed a second time after the root is found, setting all parent pointers to point to the root. Two alternatives to PC are PATH-SPLITTING (PS) and PATH-HALVING (PH). With PS the parent pointer of every node on the find-path is set to point to its grandparent. This has the effect of partitioning the find-path nodes into two disjoint paths, both hanging off the root. In PH this process of pointing to a grandparent is only applied to every other node on the find-path. The advantage of PS and PH over

PC is that they can be performed without traversing the find-path a second time. On the other hand, PC compresses the tree more than either of the other two.

Note that when using ranks PC, PS, and PH all maintain the increasing rank property. Furthermore, any one of the three combined with either LR or LS has the same asymptotic time bound of $O(m \cdot \alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [3,16,18].

Other compression techniques include REVERSAL-OF-TYPE-k. With this the first node x on the find-path and the k last nodes are set to point to the root while the remaining nodes are set to point to x. In a REVERSAL-OF-TYPE-0 (R0) every node on the find-path from x, including the root, is set to point to x and x becomes the new root, thus changing the representative element of the set. Both R0 and REVERSAL-OF-TYPE-1 (R1) can be implemented efficiently, but for any values of $k > 1$ implementation is more elaborate and might require a second pass over the find-path [18]. We limit $k \leq 1$. Using either R0 or R1 with any of NL, LR, or LS gives an asymptotic running time of $O(n + m \log n)$ [18].

In COLLAPSING (CO) every node of a tree will point directly to the root so that all find-paths are no more than two nodes long. When merging two trees in a UNION operation, nodes of one of the trees are changed to point to the root of the other tree. To implement this efficiently the nodes are stored in a linked list using a sibling pointer in addition to the parent pointer. The asymptotic running time of CO with either LS or LR is $O(m + n \log n)$; CO with NL is $O(m + n^2)$ [18].

It is possible to combine any of the three different UNION methods with any of the seven compression techniques (including NF), thus giving rise to a total of 21 different algorithm combinations. We denote each of these algorithms by combining the abbreviation of its UNION method with the abbreviation of its compression technique (e.g., LRPC). The asymptotic running times of these classical algorithms are summarized in the tables on page 280 of [18].

Classical interleaved algorithms. INTERLEAVED (INT) algorithms differ from the UNION-FIND algorithms mentioned so far in that the two FIND operations in line 5 of Algorithm 1 are performed as one interleaved operation. The main idea is to move two pointers r_x and r_y alternatively along their respective find-paths such that if x and y are in the same component then $p(r_x) = p(r_y)$ when they reach their lowest common ancestor and processing can stop. Also, if x and y are in different components, then in certain cases the two components are linked together as soon as one of the pointers reaches a root. Thus, one root can be linked into a non-root node of the other tree. The main advantage of the INT algorithms is that they can avoid traversing portions of find-paths.

The first INT algorithm is Rem's algorithm (REM) [6]. Here it is assumed that each node has a unique identifier (*id*) that can be ordered (typically in the range 1 through n). In our presentation we let this identifier be the index of the node.

The constructed trees always have the property that a lower numbered node either points to a higher numbered node or to itself (if it is a root). Instead of performing FIND(x) and FIND(y) separately, these are executed simultaneously by first setting $r_x \leftarrow x$ and $r_y \leftarrow y$. Then whichever of r_x and r_y has the smaller parent value is moved one step upward in its tree. In this way it follows that if x

and y are in the same component then at some stage of the algorithm we will have $p(r_x) = p(r_y) =$ the lowest common proper ancestor of x and y. The algorithm tests for this condition in each iteration and can, in this case immediately stop.

As originally presented, Rem integrates the UNION operation with a compression technique known as SPLICING (SP). In the case when r_x is to be moved to $p(r_x)$ it works as follows: just before this operation, r_x is stored in a temporary variable z and then, just before moving r_x up to its parent $p(z)$, $p(r_x)$ is set to $p(r_y)$, making the subtree rooted at r_x a sibling of r_y. This neither compromises the increasing parent property (because $p(r_x) < p(r_y)$) nor invalidates the set structures (because the two sets will have been merged when the operation ends.) The effect of SP is that each new parent has a higher value than the value of the old parent, thus compressing the tree. The full algorithm is given as Algorithm 2. The running time of Rem with SP (RemSP) is $O(m \log_{(2+m/n)} n)$ [18].

Tarjan and van Leeuwen present a variant of Rem that uses ranks rather than identifiers. This algorithm is slightly more complicated than Rem, as it also checks if two roots of equal rank are being merged and if so updates the rank values appropriately. Details are given on page 279 of [18]. We label this algorithm as TvL. The running time of TvL with SP (TvLSP) is $O(m \cdot \alpha(m, n))$.

Algorithm 2. RemSP(x, y)

```
 1: r_x ← x, r_y ← y
 2: while p(r_x) ≠ p(r_y) do
 3:    if p(r_x) < p(r_y) then
 4:       if r_x = p(r_x) then
 5:          p(r_x) ← p(r_y), break
 6:       z ← r_x, p(r_x) ← p(r_y), r_x ← p(z)
 7:    else
 8:       if r_y = p(r_y) then
 9:          p(r_y) ← p(r_x), break
10:       z ← r_y, p(r_y) ← p(r_x), r_y ← p(z)
```

Note that SP can easily be replaced in either Rem or TvL with either PC or PS. However, it does not make sense to use PH with either because PH might move one of r_x and r_y past the other without discovering that they are in fact in the same tree. Also, since R0 and R1 would move a lower numbered (or ranked) node above higher numbered (ranked) nodes, thus breaking the increasing (rank or *id*) property, we will not combine an INT algorithm with either R0 or R1.

2.2 Implementation Enhancements

We now consider three different ways in which the classical algorithms can be made to run faster by: *i)* making the algorithm terminate faster, *ii)* rewriting so that the most likely case is checked first, and *iii)* reducing memory requirements.

Immediate parent check (IPC). This is a recent enhancement that checks before beginning QUICK if x and y have the same parent. If they do, QUICK is not executed, otherwise execution continues. This idea is motivated by the fact that trees often have height one, and hence it is likely that two nodes in the same tree will have the same parent. The method was introduced by Osipov et al. [13] and used together with LR and PC in an algorithm to compute minimum weight spanning trees. IPC can be combined with any classical algorithm except Rem, which already implements the IPC test.

Better interleaved algorithms. The TvL algorithm, as presented in [18], can be combined with the IPC enhancement. However, the TvL algorithm will already, in each iteration of the main loop, check for $p(r_x) = p(r_y)$ and break the current loop iteration if this is the case. Still, three comparisons are needed before this condition is discovered. We therefore move this test to the top of the main loop so that the loop is only executed while $p(r_x) \neq p(r_y)$. (This is similar to the IPC test.) In addition, it is possible to handle the remaining cases when $rank(p(r_x)) = rank(p(r_y))$ together with the case when $rank(p(r_x)) < rank(p(r_y))$. For details see [14]. This will, in most cases, either reduce or at least not increase the number of comparisons; the only exception is when $rank(p(r_x)) < rank(p(r_y))$, which requires either one or two more comparisons. We call this new enhanced implementation eTvL.

A different variation of the TvL algorithm, called the ZIGZAG (ZZ) algorithm was used in [11] for designing a parallel UNION-FIND algorithm where each tree could span across several processors on a distributed memory computer. The main difference between the ZZ algorithm and eTvL is that the ZZ algorithm compares the ranks of r_x and r_y rather than the ranks of $p(r_x)$ and $p(r_y)$. Due to this it does not make sense to combine the ZZ algorithm with SP.

Memory smart algorithms. We now look at ways to reduce the amount of memory used by each algorithm. In the algorithms described so far each node has a parent pointer and, for some algorithms, either a size or rank value. In addition, for the CO algorithm each node has a sibling pointer. It follows that we will have between one and three fields in the record for each node. (Recall that we use the corresponding node's index into the array of records as its "name").

It is well known, although as far as we know undocumented, that when the parent pointer values are integers one can eliminate one of the fields for most UNION-FIND implementations. The idea capitalizes on the fact that usually only the root of a tree needs to have a rank or size value. Moreover, for the root the parent pointer is only used to signal that the current node is in fact a root. Thus it is possible to save one field by coding the size or rank of the root into its parent pointer, while still maintaining the "root property." This can be achieved by setting the parent pointer of any root equal to its negated rank (or size) value.

This MEMORY-SMART (MS) enhancement of combining the rank/size field with the parent pointer can be incorporated into any of the classical algorithms except those using an INT algorithm (because they require maintaining the rank at every node, not just the root) or PH (because PH changes parent pointers to the grandparent value, which, if negative, will mess up the structure of the tree.) MS can also be combined with the IPC enhancement. Because Rem does not use either size or rank, we will also classify it as an MS algorithm.

3 Experiments and Results

For the experiments we used a Dell PC with an Intel Core 2 Duo 2.4 GHz processor and 4MB of shared level 2 cache, and running Fedora 10. All algorithms were implemented in C++ and compiled with GCC using the -O3 flag.

We used three test sets. The first consists of nine real world graphs (rw) of varying sizes drawn from different application areas such as linear programming, medical science, structural engineering, civil engineering, and the automotive industry [4]. The second includes five synthetic small world graphs (sw) and the third contains six synthetic Erdös-Rényi style random graphs (er). For each synthetic graph (sw or er), we generated five different random graphs with the same number of vertices and with the same edge probability using the GTGraph package [1]. Statistics reported here about these graphs are an average for the five different random graphs of that type and size. For structural properties of the test sets as well as additional figures see [14].

To compute the run-time of an algorithm for a given graph, we execute the algorithm five times using each of five different random orderings of the edges, taking the average time as the result. For test sets sw and er this is also done for each graph. Hence we compute the average run-time of each graph in rw by taking the average of 25 total runs and for sw and er by taking the average of 125 total runs. Algorithms stop if and when they find that the entire graph is a single connected component. The time for all runs of reasonable algorithms (not including the extremely slow algorithms NL with no compression and NL with CO) ranged from 0.0084 seconds to 28.7544 seconds.

We now present the results of experiments from 55 different algorithms. We first consider the classical algorithms and then using the enhancements presented in Section 2.2. Finally, we compare and discuss the 10 overall fastest algorithms. For each type of algorithm we give a table in which each cell represents an algorithm that combines the row's union method with the column's compression technique. The combinations for crossed out cells are either not possible or non-sensical. The rows with gray background are repeated from an earlier table.

Throughout we will say that an algorithm X *dominates* another algorithm Y if X performs at least as well as Y (in terms of run-time) on every input graph. For illustrative purposes we will pick four specific dominating algorithms numbered according to the order in which they are first applied. These will be marked in the tables with their number inside a colored circle, as in ❶. If algorithm X dominates algorithm Y, an abbreviation for X with its number as a subscript will appear in Y's cell of the table, as in LRPC_1. Algorithms that are not dominated by any other algorithm are marked as undominated.

Classical Algorithms (Table 1). The two algorithms LRPC and LRPH are generally accepted as best, and so we begin by examining these. Although they dominate a large majority, RemSP dominates still more. This gives us the first three dominating algorithms. LRPC_1 dominates 14 algorithms. Three additional algorithms are dominated by LRPH_2, including LRPC; hence, LRPH also dominates all algorithms dominated by LRPC. Figure 1(a) shows the relative performance of the ten remaining algorithms dominated by RemSP_3. Note that RemSP dominates LRPH. Because LRPC dominates both LSPC and TVLPC, our experiments show that LR is a better UNION method to combine with PC. Only two algorithms are undominated. In the following we do not report results for algorithms using NF,

Table 1. Relative performance of the classical UNION-FIND algorithms

	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC$_1$	LRPH$_2$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	LRPC$_1$	✗
LR	LRPC$_1$	❶ LRPH$_2$	❷ RemSP$_3$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	✗
LS	LRPC$_1$	LRPC$_1$	RemSP$_3$	RemSP$_3$	RemSP$_3$	LRPC$_1$	LRPC$_1$	✗
Rem	LRPC$_1$	RemSP$_3$	✗	undom.	✗	✗	✗	❸ undom.
TVL	LRPC$_1$	LRPC$_1$	✗	RemSP$_3$	✗	✗	✗	LRPH$_2$

R0, or R1 as these compression techniques consistently require several orders of magnitude more time than the others.

IPC Algorithms (Table 2). The algorithms for which using IPC always performed better than the corresponding non-IPC version are marked in the table with an uparrow (↑). Note that Rem is, by definition, an IPC algorithm, and hence, there are no changes between its "IPC" version and its "non-IPC" version; they are the same algorithm. In each of the other five cases, using IPC is sometimes better than not, and vice versa. No IPC version is consistently worse than its non-IPC version.

One algorithm, IPC-LRPS$_4$, dominates three others. Also, RemSP dominates that algorithm and others. Figure 1(b) shows the performance of the remaining six dominated algorithms relative to Rem–SP. Among the IPC-LR algorithms, PS dominates the other two compression techniques. Note also that unlike the results of the previous subsection where LRPC dominated LSPC, neither IPC-LRPC nor IPC-LSPC dominates the other. In general IPC-LSPC performed better than IPC-LRPC.

Table 2. IPC relative performance

	PC	PH	PS	SP
IPC-LR	IPC-LRPS$_4$↑	IPC-LRPS$_4$	❹ RemSP$_3$	✗
IPC-LS	RemSP$_3$↑	RemSP$_3$	RemSP$_3$	✗
Rem	RemSP$_3$	✗	undom.	❸ undom.
IPC-TVL	IPC-LRPS$_4$↑	✗	RemSP$_3$	RemSP$_3$↑

Interleaved Algorithms (Table 3). RemSP dominates the five new INT algorithms. Figure 1(c) shows the performance of each of these relative to RemSP. The performance of the INT algorithms was impacted more by the compression technique than by the union method. Also, PC is considerably worse (by approximately 50%) than either PS or SP.

Memory-smart Algorithms (Table 4). RemSP dominates six of the MS algorithms. Figure 1(d) shows the relative performance of these relative to RemSP.

Note that MS-IPC-LSPS and MS-IPC-LSPC come close to RemSP, and, in fact, these are the only two dominated algorithms that are among the 10 fastest algorithms using the metric described next.

The Fastest Algorithms. We now compare all algorithms using a different metric than the "dominates" technique. We begin by calculating, for each input graph and each algorithm, its run-time relative to the best algorithm for that graph (GLOBAL-MIN). For each algorithm and each type of input (rw, sw, er) we then compute the average relative time over all input of that type. The average of these three averages is then used to rank order the algorithms.

The results for the top ten ranked algorithms are given in Table 5. Each cell contains both the algorithm's rank for the given type of graph and its relative timing reported as a percent. The last row in the table is included to show how far the last ranked algorithm is from the algorithms that are not in the top 10.

Table 3. INT relative performance

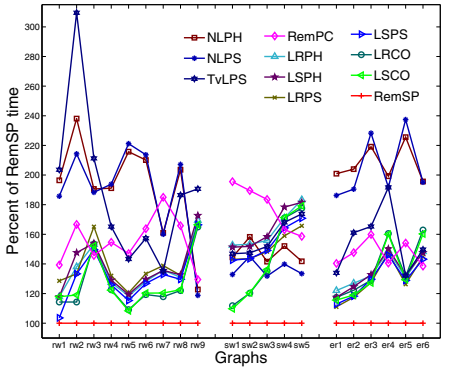
	PC	PS	SP
Rem	RemSP_3	undom.	❸ undom.
TVL	LRPC_1	RemSP_3	LRPH_2
IPC-TVL	IPC-LRPS_4	RemSP_3	RemSP_3
ETVL	RemSP_3	RemSP_3	RemSP_3
ZZ	RemSP_3	RemSP_3	⨉

When considering rw, er, and "all graphs," the next best algorithm is ranked 11 and is 8.68%, 10.68%, and 8.12% worse than the slowest algorithm reported in the table, respectively. For the sw graphs, however, the fastest algorithm not in the table is faster than four of the algorithms in the table and only 2.66% slower than the algorithm ranked 6 (MS-LRCO). For sw graphs five algorithms not in the table were faster than the slowest of those reported in the table.

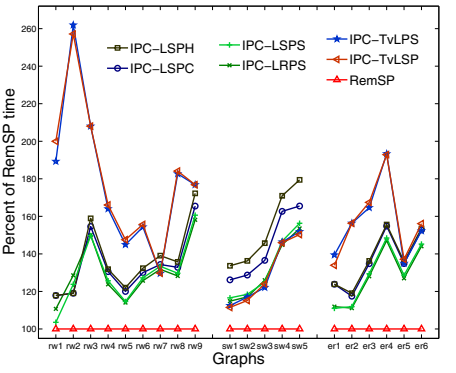
The relative performance of the five algorithms with ranks 1-5 on the overall average is plotted in Figure 1(e). The figure clearly shows RemSP outperformed all other algorithms. Notice that LS and LR with all other variations on the algorithm remaining constant tend to have similar performance trends. Furthermore, neither LS nor LR consistently outperformed the other. Of the top 10 algorithms all use the MS enhancement and out of these all but MS-IPC-LRPC and MS-IPC-LSPC are one-pass algorithms. However, out of the top five algorithms two use PC. We note that PS has the advantage over PH, perhaps because it is easier to combine with MS.

Table 4. MS relative performance

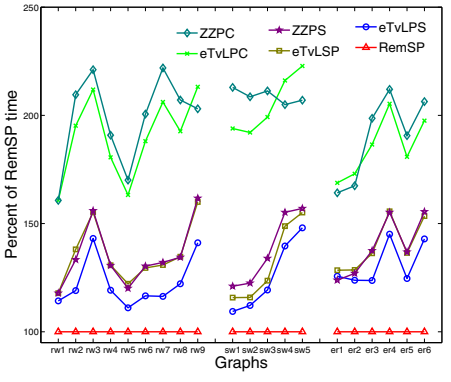
	PC	PS	CO
MS-NL	RemSP_3	RemSP_3	⨉
MS-LR	$\text{RemSP}_3\uparrow$	undom.↑	undom.↑
MS-LS	$\text{RemSP}_3\uparrow$	undom.↑	undom.↑
MS-IPC-LR	undom.↑	undom.	⨉
MS-IPC-LS	RemSP_3	RemSP_3	⨉
Rem	RemSP_3	undom.	⨉



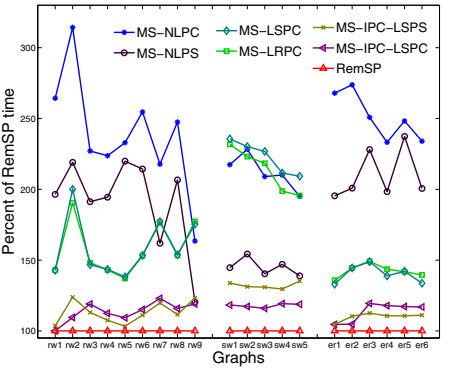
(a) Classical dominated by RemSP



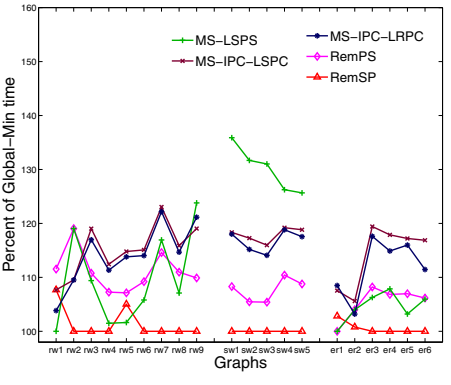
(b) IPC dominated by RemSP



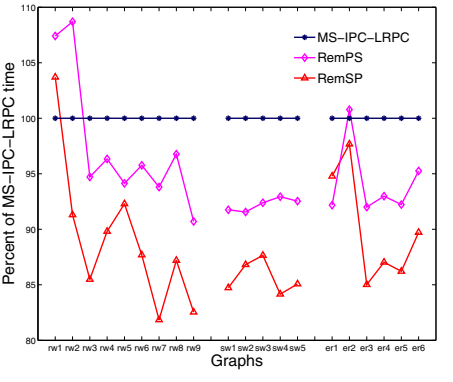
(c) Int dominated by RemSP



(d) Memory-Smart dominated by RemSP



(e) The five fastest algorithms



(f) Improvement over best enhanced classical algorithm (MS-IPC-LRPC)

Fig. 1. Relative performance of Union-Find algorithms

Table 5. Rank order(% of GLOBAL-MIN) of the fastest algorithms based on graph type

	All graphs	Real-World	Small-World	Erdős-Rényi
RemSP	1 (100.67)	1 (101.41)	1 (100.00)	1 (100.61)
RemPS	2 (108.07)	5 (111.16)	2 (107.67)	4 (105.36)
MS-IPC-LRPC	3 (114.28)	6 (114.17)	3 (116.73)	7 (111.94)
MS-LSPS	4 (114.71)	2 (109.47)	10 (130.1)	2 (104.55)
MS-IPC-LSPC	5 (115.73)	8 (115.17)	4 (117.92)	8 (114.09)
MS-LRPS	6 (115.90)	4 (111.02)	13 (131.65)	3 (105.05)
MS-IPC-LRPS	7 (116.90)	3 (111.00)	11 (131.22)	5 (108.48)
MS-LSCO	8 (118.29)	9 (115.21)	6 (123.05)	9 (116.61)
MS-LRCO	9 (118.91)	10 (115.47)	5 (123.04)	10 (118.22)
MS-IPC-LSPS	10 (119.08)	7 (114.47)	15 (132.15)	6 (110.62)
Fastest not listed	11 (127.20)	11 (124.15)	7 (125.71)	11 (128.90)

Unlike the sparse matrix studies in [8,10,20] we found that using either LR or LS does pay off. Hynes [9] and later Wassenberg et al. [19] found that CO was the best choice. Our results also show that CO used with either LR or LS is one of the faster algorithms, but only if used with the MS enhancement. However, on average it is outperformed by several classical algorithms (such as LSPC and LRPC) if these are enhanced with IPC and MS.

The clear winner in our study was RemSP. This was the fastest algorithm for all types of graphs. On average it was 13.52% faster (with a range of −3.57% to 22.18%) compared to the best non-Rem algorithm and 7.34% faster (with a range of −2.75% to 19.05%) than the second best Rem algorithm. We believe that this is due to several factors: it has low memory overhead; INT algorithms perform less operations than other classical algorithms; it incorporates the IPC enhancement at every step of traversal, not only for the two initial nodes; and even when integrated with SP the algorithm is relatively simple with few conditional statements.

4 Concluding Remarks

This paper reports the findings of 1600 experiments on each of 53 different UNION-FIND algorithms: 27 classical variations that were studied from a theoretical perspective up through the 1980s, nine IPC variations, five more INT variations, and 12 additional MS variations. We also ran two experiments using the very slow algorithms NLNF and NLCO. In order to validate the results, we reran the 84,800 experiments on the same machine. While the absolute times varied somewhat, the same set of algorithms had the top five ranks as did the set of algorithms with the top 10 ranks, and RemSP remained the clear top performer.

We have also constructed spanning forests using both DFS and BFS, finding that use of the UNION-FIND algorithms are substantially more efficient.

The most significant result is that RemSP substantially outperforms LRPC even though RemSP is theoretically inferior to LRPC. This is even more surprising because LRPC is both simple and elegant, is well studied in the literature, is often implemented for real-world applications, and typically is taught as best in standard algorithms courses. In spite of this RemSP improved over LRPC by an average of 52.88%, with a range from 38.53% to 66.05%. Furthermore, it improved over LRPH (which others have argued uses the best one-pass compression technique) by an average of 28.60%, with a range from 15.15% to 45.44%.

Even when incorporating the MS and IPC enhancements, RemSP still improved over these other two classical algorithms on all inputs except one (rw1), where MS-IPC-LRPC improved over RemSP by only 3.57%. On average, RemSP improves over MS-IPC-LRPC by 11.91%, with a range from -3.70% to 18.15%. The savings incurred over the MS-IPC-LRPC are illustrated in Figure 1(f) where the times for the top two ranked algorithms are plotted relative to the time for MS-IPC-LRPC.

To verify that our results hold regardless of the cache size, we ran experiments using twice, three times, and four times the memory for each node (simulating a smaller cache). The relative times for the algorithms under each of these scenarios were not significantly different than with the experiments reported here.

We believe that our results should have implications for developers of software libraries like [5] and [12], which currently only implement LRPC and LRPH in the first case and LSPC in the second. Initial profiling experiments show that RemSP gives both fewer cache misses and fewer parent jumps than the classical algorithms. We postpone discussion of these results until the full version of the paper.

These UNION-FIND experiments were conducted under the guise of finding the connected components of a graph. As such, the sequences of operations tested were all UNION operations as defined by the edges in graphs without multiedges. It would be interesting to study the performances of these algorithms for arbitrary sequences of intermixed MAKESET, UNION, and FIND operations.

References

1. Bader, D.A., Madduri, K.: GTGraph: A synthetic graph generator suite (2006), `http://www.cc.gatech.edu/~kamesh/GTgraph`
2. Banachowski, L.: A complement to Tarjan's result about the lower bound on the complexity of the set union problem. Inf. Proc. Letters 11, 59–65 (1980)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
4. Davis, T.A.: University of Florida sparse matrix collection. Submitted to ACM Transactions on Mathematical Software
5. Dawes, B., Abrahams, D.: The Boost C++ libraries (2009), `www.boost.org`
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
7. Galil, Z., Italiano, G.F.: Data structures and algorithms for disjoint set union problems. ACM Comput. Surv. 23(3), 319–344 (1991)

8. Gilbert, J.R., Ng, E.G., Peyton, B.W.: An efficient algorithm to compute row and column counts for sparse Cholesky factorization. SIAM J. Matrix Anal. Appl. 15(4), 1075–1091 (1994)
9. Hynes, R.: A new class of set union algorithms. Master's thesis, Department of Computer Science, University of Toronto, Canada (1998)
10. Liu, J.W.H.: The role of elimination trees in sparse factorization. SIAM J. Matrix Anal. Appl. 11, 134–172 (1990)
11. Manne, F., Patwary, M.M.A.: A scalable parallel union-find algorithm for distributed memory computers. To appear in proc. PPAM 2009. LNCS. Springer, Heidelberg (2009)
12. Melhorn, K., Näher, S.: LEDA, A Platform for Combinatorial Geometric Computing. Cambridge University Press, Cambridge (1999)
13. Osipov, V., Sanders, P., Singler, J.: The filter-Kruskal minimum spanning tree algorithm. In: ALENEX, pp. 52–61 (2009)
14. Patwary, M.M.A., Blair, J., Manne, F.: Efficient Union-Find implementations. Tech. Report 393, Dept. Computer Science, University of Bergen (2010), `http://www.uib.no/ii/forskning/reports-in-informatics/reports-in-informatics-2010-2019`
15. Poutré, J.A.L.: Lower bounds for the union-find and the split-find problem on pointer machines. J. Comput. Syst. Sci. 52, 87–99 (1996)
16. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM 22, 215–225 (1975)
17. Tarjan, R.E.: A class of algorithms which require nonlinear time to maintain disjoint sets. J. Comput. Syst. Sci. 18, 110–127 (1979)
18. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM 31, 245–281 (1984)
19. Wassenberg, J., Bulatov, D., Middelmann, W., Sanders, P.: Determination of maximally stable extremal regions in large images. In: Proc. of Signal Processing, Pattern Recognition, and Applications (SPPRA). Acta Press (2008)
20. Wu, K., Otoo, E., Suzuki, K.: Optimizing two-pass connected-component labeling algorithms. Pattern Anal. Appl. 12, 117–135 (2009)