Word encoding tree connectivity works

Stephen Alstrup* Jens Peter Secher† Mikkel Thorup‡

Abstract—**Experimentally, we demonstrate the practical relevance of the theoretical trick of encoding trees in words.**

1 Introduction

In 1983, Gabow and Tarjan [3] showed that union-find can be supported in linear time, if the union operations, but not the order they come in, is known in advance. A different way of casting this problem is that of answering on-line connectivity queries for a subforest F of a given tree T while edges from T are added to F in some arbitrary order, ending with $F = T$.

Gabow and Tarjan [3] introduced the idea of encoding small "atomic" subtrees of T size $\Theta(\log n)$ in a single word. This is based on the standard assumption that the word length ω is at least $\log n$; for otherwise, we cannot even address the nodes of the tree. Having encoded an atomic tree t, in constant time, we can add an edge from t to F, and answer connectivity queries for $F \cap t$. The techniques from [3] were fairly involved, and unlikely to be of practical relevance, particularly because the alternative solution is the very efficient standard union-find algorithm with running time $O(\alpha(m, n)n)$ where n is the tree size and m the number of queries [4]. Still it is interesting to know if the tree encoding idea itself is just a theoretical trick, or whether it can actually be used to improve practical performance.

More recently, Alstrup, Secher, and Spork [1] introduced a simpler kind of tree encoding, and applied it to the reverse problem of the above; namely, that of answering m connectivity queries to a forest F over n nodes as the edges are deleted. Again, they achieve a linear time algorithm, but this time, the best previous algorithm is that of Even and Shiloach [2] with running time $O(m + n \log n)$. Like the standard union-find algorithm, Even and Shiloach's algorithm is very simple, likely with very small constants, but log is a sufficiently fast growing function, that there is a hope of beating it for n not too large.

*E-mail:stephen@itu.dk. The IT University in Copenhagen. Glentevej 67, 2400 Copenhagen NV, Denmark

†E-mail:jpsecher@diku.dk. Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen, Denmark

‡E-mail:mthorup@research.att.com AT&T Labs—Research

For this paper, we implemented and compared the ASS-algorithm of Alstrup et al. [1] and the ES-algorithm of Even and Shiloach [2]. Quite interestingly, the ASS-algorithm started winning already around 1000 nodes. This emphasizes the importance of being very careful dismissing creative usage of word operations as being just a theoretical game, without any practical relevance. The operations are there in standard programming languages such as C, ready for use both in theory and in practice.

Below, we first review the simple ES-algorithm, then we sketch the ASS-algorithm, and finally we present our experimental results.

2 The ES-algorithm

In the ES-algorithm, each connected component has a unique number which is assigned to all nodes within it. Thus nodes are connected exactly if they are assigned the same number, which can be determined in constant time. Initially, we assign 0 to all nodes. When an edge is removed, splitting some tree, the resulting two subtrees are traversed in parallel until the smaller tree S is identified. A fresh number is chosen and assigned to nodes in S. The total time of an edge deletion is proportional to the number of reassignments, and every time a node is assigned a new number, the component it is in is at least halved, so this happens at most $\log n$ times per node. Thus, the total time of the ES-algorithm is $O(m + n \log n)$.

3 The ASS-algorithm

In the ASS-algorithm, as a first preprocessing, we partition the initial tree into $O(n/\log n)$ "atomic" subtrees of size $\log n$. Each subtree has at most two boundary nodes in which it intersects other subtrees. Connectivity between the $O(n/\log n)$ boundary nodes are maintained by the ES-algorithm in a tree induced by the boundary nodes. Each subtree is represented in a single word such that deletion of an edges and connectivity among nodes in a subtree is supported in worst case constant time. Here we just sketch the idea. Let the edges be ordered arbitrarily. A subset of the edges can be represented in a single machine word by letting the ith bit be 1 if edge i belongs to the set and 0 otherwise. Let the tree be

Algorithm	$m\backslash n$	10^3	$5*10^3$	10^4	$5*10^4$	10^5
ASS	n	0.1	0.9	1.8	10.413	21.763
$O(n+m)$	$n\log n$	0.738	5.150	11.375	77.263	166.850
	$n\sqrt{n}$	2.250	28.113	80.550	1004.050	2876.250
ES	n	0.1	1.0	2.062	17.187	40.525
$O(n\log n+m)$	$n\log n$	0.75	5.225	11.562	83.175	187.188
	$n\sqrt{n}$	2.287	28.137	80.312	998.350	2863.762

Table 1: *Run time in seconds.*

rooted in an arbitrary node r. To each node v in the tree we associate the set of edges on the path from v to r, denoted *RootPath*(v). Furthermore let *CurrentEdges* be the set of edges not deleted from the tree. In [1] it is shown how to initialize these sets in linear time using linear space. Deleting an edge now correspond to clearing a bit in *CurrentEdges*. If the edge deletion disconnect the subtrees boundary nodes the induced tree is updated to. Connectivity between two nodes v and w in the same subtree can now be computed as

$$0 = (\mathit{RootPath}(v) \oplus \mathit{RootPath}(w)) \wedge \neg\mathit{CurrentEdges}.$$

where '$\oplus$", '$\wedge$', and '$\neg$' denote the bitwise operations 'exclusive-or', 'and', and 'not'. If the two nodes belong to different trees we check reachability among the boundary nodes they can reached. As described in [1], the above algorithm can be implemented in $O(m+n)$ time.

4 Experimental results

Asymptotically, the ES-algorithm spends more time per deletion than the ASS-algorithm, but it is simpler, and we expect it to have smaller constants. Further, they both spend constant time on connectivity queries, but here the ES-algorithm should be strictly faster, as it only needs to compare two component numbers. The natural questions are (1) if we let deletions dominate, does the ASS-algorithm start dominating the ES-algorithm for some reasonably small n? and (2) how much slower do the connectivity queries become with the ASS-algorithm?

To answer these questions, we ran the ASS-algorithm, including preprocessing, against the ES-algorithm on randomly generated trees of varying sizes $n = 1000, 5000, 10000, 50000, 100000$, and for each tree we tried varying numbers $m = n, n\log n, n\sqrt{n}$ of randomly generated connectivity queries. The results are presented in Table 1. In the run time of the algorithms the preprocessing time are included. For the largest values of $n = 10000, 50000, 100000$ the following expressions cover the complexity approximately : ASS $(1.27n + 0.93m) \times 10^{-4}$ seconds, and ES $(0.19n\log n + 0.89m) \times 10^{-4}$ seconds. Thus ASS takes over already around 1000 nodes if the deletions are dominating, and the extra price ASS pays for the queries is so small that the ASS algorithm is always at least competitive. The reason that the ES- and ASS-query times are so close to each other relates to the fact that they are both so fast that the overhead of one procedure call for our interface is a dominating factor.

References

[1] S. Alstrup, J.P. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *IPL*, 64(4):161–164, 1997.

[2] S. Even and Y. Shiloach. An on-line edge-deletion problem. *JACM*, 28:1–4, 1981.

[3] H.N. Gabow and R.E. Tarjan. A linear–time algorithm for a special case of disjoint set union. *JCSS*, 30(2):209–221, 1985. See also STOC'83.

[4] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.