ISSN 0874-338X Technical Report TR2015/02

Recovering a labeling algorithm for the disjoint-set problem to improve Kruskal's efficiency on dense graphs

Ana Maria de Almeida[1,3] Rui Salgueiro[2,3]

[1]ISCTE-IUL
[2]DMUC
[3]Centre for Informatics and Systems of the University of Coimbra, Portugal

Abstract. This work recovers a fast labeling algorithm for the disjoint-set problem and shows that, specially for non-sparse graphs, this data structure is more efficient than the Disjoint-Set Union–and–Find method for implementing Kruskal's algorithm (that returns a minimum spanning tree of a given graph). The latter data structure was introduced by Knuth in his classical trilogy on algorithms and is based on a special tree representation and path compression algorithms. Tarjan and van Leeuwen proved it to have an almost linear asymptotic complexity. We present bounds for the asymptotic behavior of the labeling algorithm and discuss the particularities that account for the difference in performance of either data structures implementations for Kruskal's algorithm. Several computational experiments using different versions of the canonical algorithm and the labeling one are presented and provide evidence supporting the theoretical conclusions.

Key words: Data Structures, Disjoint-set Problem, Spanning tree, Kruskal's algorithm, Performance and Efficiency.

1 Introduction

The disjoint-set union problem is a classical reference found in several textbooks. Disjoint-set data structures are widely used e.g they are used for efficient Kruskal's algorithm implementations to find a minimum spanning tree of a graph ([1], [2], [3], and [4]). The disjoint-set algorithm carries out a sequence of intermixed operations that ends with the union set of all the initial singleton sets. The data structure used for the representation of sets and operators must efficiently perform unions of sets and determine the membership of an element in a set.

A widely spread data structure for the set union problem represents the sets as trees and stores the name of the set at the root of its representative tree. A `Union` operation between two sets is performed by linking the root of the smaller tree (set) as a descendent of the root of the larger one. To determine the membership of a node (element) a `Find` operation is implemented by following the path from the given node to the root of the tree containing it. During this

traversal all the nodes in the path have their parent pointers redirected to the root - *path compression* [6]. In [5], besides describing one-pass `Find` algorithms that are more efficient than mere path compression, the authors also acknowledge that there may exist applications for which a different data structure could be more appropriated. Nevertheless, this union–and–find approach is considered to be very efficient and thus has been used throughout the last three decades in a vast majority of applications without further questioning its efficiency ([7], [8], [9], and [10]). Hereafter, we will term this data structure as *DSpath*.

This work recovers a labeling algorithm and a tree representation using simple linked lists described in [7] and [8]. This data structure will be referred to as *LA*. By using arrays to represent the tree structure we simplify the `Find` and `Union` operations and shown that this implementation may outperform any of the DSpath versions for particular applications.

Next section introduces some definitions needed thereafter. Section 3 presents brief descriptions of the variants of the DSpath algorithm and the respective asymptotic complexity. Section 4 describes the labeling algorithm data structure LA and establishes its algorithmic asymptotic complexity. In the next section an average analysis of the number of memory accesses is made showing why the LA, under certain general restrictions, will tend to perform better when used to implement Kruskal's algorithm. Finally, Section 7 presents the computational experiments supporting the previous considerations. In the last section some conclusions are drawn.

2 Preliminaries

Let $G = (V, E)$ be a connected weighted undirected graph, where $V = \{1, \ldots, n\}$ is the set of nodes, $E = \{e = \{i, j\} : i, j \in V\}$ the set of k edges, and positive costs are associated with each one of the edges, denoted by c_e (or c_{ij} if the end nodes must be highlighted). A graph is said to be *connected* if there is a path between any pair of different nodes. Notice that when the graph is complete (all the nodes are interconnected) we have $k = n(n-1)/2$ edges.

A connected acyclic subgraph of G, $T = (V_T, E_T)$, with $V_T \in V$ and $E_T \subset E$ is called a *tree* of G and if $V_T = V$, T is called a *spanning tree* for G. If we need to use an identification of the tree we denotie one particular node as the *root* of the tree, that then becomes a *rooted tree*.

Since a rooted tree is a hierarchical representation, levels of hierarchy can be defined by beginning with the root and, in an analogy with ancestor trees, all the descendent nodes are termed *children* (or *siblings*). The direct ancestor of a node is called its *parent*.

According to [5,10], let α denote the functional inverse of Ackermann's A function: $\alpha(m, n) = \min\{i \geq 1 \ : A(i, \lfloor m/n \rfloor) > \log n\}$. While Ackermann's function has an explosive growth, α grows very slowly. Tarjan and van Leeweun conclude that, for all practical purposes, $\alpha(m, n)$ can be considered as a constant no larger than four [5] .

Throughout this paper the base 2 logarithms are always used.

3 Data Structures and Time bounds for DSpath

The most common representation for the disjoint-set data structure uses rooted trees whose nodes are the elements of the set. Each node points to its parent and the root points to itself. Initially, each element is a singleton set (one node rooted tree) and therefore all the initial sets are disjoint. To join two sets both an `Union` and a `Find` operation must be defined. The *Union* operation combines two disjoint sets, rooted at A and B respectively, into one new set, either rooted at A or B whichever is most convenient. We will always consider that the smaller tree is connected to the larger one thus retaining this latter root. This is known as *union–by–size*. The *Find* operation for a given node determines to which tree it belongs. The search is made by following the parent pointers until a node is repeated - the root. Since the latter operation dominates the entire cost of the combination of both operations, three different versions were proposed to shorten traversal paths in search of more efficient implementations: *compression*, *splitting* and *halving*. After following the path from a given node to the root of the tree that contains it, the compression rule is the redirection of the parent of each node encountered along the path to the root. Therefore, path compression requires two passes along the same path. In order to to use only one pass van Leeuwen and van der Weide (cited by [5]) propose other two variants for compression of the path- *halving* and *splitting*. When searching for the root, make each node point to its grandparent (the parent node of its parent), with the exception of the root and its direct child – path *splitting*. For as long as there are enough ancestors a path is split into two paths approximately half as long as the original one. If every other node is made to point to its grandparent with the same exceptions we get path *halving* since it requires only about half of the pointer updates for each `Find` than those required by splitting. Moreover, it keeps the nodes on the path together while halving the length of the path with each search. Thus, the more the traversals, the better the compression of the path to the root.

As for the time complexity of disjoint-set algorithms, it it is obvious that the `Find` operation is costlier than the `Union` one, dominating the total time bounds. In fact, after a `Find` operation the union–by–size of two trees is quite straightforward. For n initial singletons there are, at most, $n-1$ `Union` operations. If k is the number of edges on the tree then $k \leq n-1$. Following the assumption in [9] we shall assume that $k \geq n/2$ w.l.o.g. since there are at most $2k$ elements in sets containing more than one element.

The exact number of `Find` operations depends on the disjoint-sets, i.e, on the edges of the tree. A search in a singleton set only requires time $\mathcal{O}(1)$. Denote the total number of `Find` operations by m and hereafter assume that $m \geq n$ (the usual case). Tarjan showed an upper bound in terms of the inverse Ackermann's A function. In [5] he proves that a DSpath algorithm performing $n-1$ `Union` and m `Find` operations requires time $\Theta(m\alpha(m,n))$, assuming that $m \geq n$ and where $\alpha(m,n)$ is the quantity of accesses made by each `Find` operation to the data structure whatever the type of path compression used: compression, splitting or halving. Tarjan and Gabow also show that, for the most general problem, no algorithm can overcome this bound ([12]).

4 Labeling Algorithm Data Structure, LA, and its Time complexity

The next data structure recovers a version found in [7] and [8] although using a slightly different representation for implementing the sets. A set is then represented by a linear linked list and implemented by an array of registers with two important information fields: the tree label (the root) and a pointer to the next node in the list. The number of nodes in the tree is also recorded.

Using this representation, the `Union` operation is just the relabeling of each node in one of the trees (the smallest one). This is accomplished by one linear traversal on the smallest tree. The `Find` operation is a single array access and no compression is needed, being therefore always $\mathcal{O}(1)$.

The algorithm begins by initializing the n singleton trees (n array elements). It then proceeds by joining two trees thus reducing the number of trees by one at each iteration. This means that it performs as many `Union` operations as the DSpath algorithm, $n-1$. Testing if two different trees are connected needs two array accesses and one comparison. Therefore, the total number of `Find` calls needed for the `Union` operations, m, will use $2m$ accesses to the array list. Furthermore, no search for a path to the root is needed.

Consider $rl(s)$, the total number of labels needed to relabel one of two trees into a tree of size s. The best scenario corresponds to repeatedly adding a size-one tree to another tree. Thus n nodes would require $rl(n) = n-1$ relabeling operations. The worst case occurs when repeatedly joining trees having approximately the same size. The total number of relabels can be calculated by using the following recurtion where we assume w.l.o.g. that n is a power of 2:

$$rl(n) = \begin{cases} 0 & , n = 1 \\ 2\,rl(n/2) + n/2 & , n > 1 \end{cases}$$

Applying Theorem 2.1 from [7] we get $rl(n) = \mathcal{O}(n \log_2 n)$. In fact, by applying the Master Theorem in [3] we have that, in the worst case, LA `Union` operation uses $\Omega(n \log n)$ relabels. For each `Union`, two assignments are needed to maintain the linked list, adding up to a total of $2(n-1)$ assignments which shows this section main result:

Theorem 1 *The total number of operations for an LA union–find algorithm on n singleton sets is $am + bn \log_2 n + c(n-1)$, where a, b and c are constants and m is the number of calls to the Find function.*

In conclusion, LA algorithm has an upper time bound of $\mathcal{O}(\max\{m, n \log_2 n\})$, which is in accordance with [7].

5 A lower bound for the number of memory accesses for DS-path algorithms

For the DSpath versions, most `Find` operations need multiple accesses to memory. In effect, a `Find` for a given node x uses as many accesses and comparisons

to find the root as the distance from x to the root itself plus one (if x is the root of a tree there is only one access to check that it is indeed a root; if x is the child of a root, there are two: one to find the root plus the previous case, and so on). In order to evaluate a lower bound on the number of memory accesses needed for the total m `Find` operations of a DSpath algorithm assume that:

1. the `Find` operations are uniformly distributed between nodes representing roots and the other nodes (non-root nodes) of sets. If there are i roots among the n nodes the probability of a node being a root is i/n.
2. the `Find` operations are uniformly distributed among the `Union` operations, that is, approximately $m/(n-1)$ `Find` calls are made between two consecutive achieved unions[1].

Under this assumptions we present the following result:

Theorem 2 *The lower bound on the total number of accesses of the DSpath union–find algorithm on n singleton sets and m `Find` operations is $m(\frac{3}{2} - \frac{1}{n})$.*

Proof: In the first step of the algorithm there are n sets, thus n roots, and so the probability of hitting a root is 1. After the first union there are $n-1$ sets (or $n-1$ roots) and the probability of hitting a root is $(n-1)/n$. Repeating this reasoning[2] at the last iteration of the algorithm there are only 2 sets (or roots) and the probability of hitting a root is $2/n$.

Under Assumption 2 the average probability is the (non-weighted) average of the above probabilities. Therefore, the average probability of hitting a root is:

$$P(root) = \frac{\sum_{i=2}^{n} i/n}{n-1} = \frac{(n+2)(n-1)}{2n(n-1)} = (n+2)/2n = 1/2 + 1/n$$

The probability of not hitting a root is then $P(\sim root) = 1/2 - 1/n$.

Whenever the algorithm hits a root it needs one access and more than one when it hits a non-root node. For the evaluation of the total number of `Find` accesses, na, we need to add the quantity of accesses when hitting a root, $na_{root} = m * P(root) = m * (1/2 + 1/n)$, and when not hitting a root, $na_{\sim root} \geq m * P(\sim root) * 2 = m * (1/2 - 1/n) * 2$. Therefore $na = na_{root} + na_{\sim root} \geq 3/2m - m/n = m(3/2 - 1/n)$.

Note that, since the `Find` operation for the LA algorithm uses exactly m accesses (one per `Find`), the determined average number of accesses for DSpath is superior to the one for LA by a factor of approximately 1.5.

[1] In the context of Kruskal's algorithm this assumption is, in fact, optimistic. The first edge is always accepted, so only 2 `Find` operations are made before the first union. On the other hand, in the last iteration of this algorithm, it is very likely that many edges connect nodes already in the same set.

[2] Note that the `Union` operations are always intermixed with `Find` operations.

6 An application: Kruskal's algorithm

Kruskal's algorithm is usually implemented using DSpath algorithms ([3]). This is a greedy algorithm for finding the minimal spanning tree (MST) of a given graph $G = (V, E)$ (or the minimal spanning forest if the graph is not connected). It begins with a forest of n size one rooted trees and, after creating a list of all the edges ordered by increasing edge weight, chooses the next edge to join two disjoint trees constructing a larger tree, until it achieves a tree that spans the graph G. A disjoint-set algorithm is, therefore, most appropriate to implement Kruskal's algorithm.

In order to create a tree with n nodes we need to test l edges for the union of disjoint trees. That is, the number of times the algorithm must use `Find` operations to make sure that it is joining disjoint sets will add up to $l \leq k = |E|$ (see Section 2). In fact, once an edge is rejected it will no longer be a candidate for a union due to the initial ordering of the edges. The exact value of l depends on the weights of the edges of the graph. If all the edges connected to one particular node have higher weight than all the remaining edges then the algorithm must test all the other edges first. In the worst case we would have $l = n(n-1)/2 - (n-1)+1 = n(n-1)/2-n = n(n-3)/2$. Since testing each edge needs two `Find` operations (one for each extreme node) we have $m = 2 * l = 2 * n(n-3)/2 = n(n-3)$.

Conversely, in the best case the $n-1$ edges that constitute a spanning tree have the lowest possible weights and all the remaining (rejected or not chosen) edges have higher weights. Therefore, $l = n-1$, and $m = 2(n-1)$.

In [7] the authors stated that when $m = \mathcal{O}(n)$, due to the derived asymptotic complexity, the DSpath versions are more adequate than the LA type of data structure. However, this is not the case in the particular context of Kruskal's algorithm regardless of the number of edges involved and their weights as it can be observed in the following section.

7 Computational results and Numerical evidence

All the algorithms considered were implemented using C programming language, and the tests run on a computer with an Intel Core2 Q9550 at 2.83GHz CPU and 4GB RAM.

Next we compare execution times and number of relabels for Kruskal's algorithm using a LA data structure as well as the several implementations of DSpath versions for compression of the path but always using union-by-size. The representation of the trees uses a linear linked list implemented by an array of registers where each element has three integer fields: one for the tree label (which is the root for LA or the parent node for the DSpath versions); one for the next node in the list; one that indicates the size (number of nodes) of the tree which is only used if the node is the root of a tree, otherwise it is ignored.

The 50 randomly generated complete graphs[3] for the tests have a number of nodes ranging from 1000 to a maximum of 250 000.

7.1 Implementations and characteristics of Disjoint-Set Data Structures

The `Find` operation was implemented using all the variants previously refereed for DSpath and the LA version. We also implemented a non-recursive version based on the classical recursive one but avoiding redundant assignments through the use of an auxiliary vector `x` that keeps track of the sequence of the nodes in the path. The DSpath versions were based on the algorithms described in [5] with the necessary alterations to use the vector tree representation previously mentioned. The LA algorithm is the one described in Section 4.

The figures in Sections (7.2) and (**??**) present the average results for 50 instances solved by Kruskal's algorithm using each one of the variants next mentioned:

- Kruskal-Ls: Labeling algorithm using a linear linked list (Section 4)
- Kruskal-NRn: DSpath version avoiding unnecessary assignments
- Kruskal-NRd: DSpath double-pass version ([5])
- Kruskal-NRh: DSpath path halving ([5])
- Kruskal-NRs: DSpath path splitting ([5])

Although the different versions of DSpath have all the same asymptotic strict classification, $\Theta(m\alpha(m,n))$, and LA is $\mathcal{O}(\max\{m, n\log n\})$, the numerical tests have shown that the implementations above have quite variant and interesting average performances.

7.2 Numerical experiments

The first noticeable feature is that using LA as data structure for Kruskal turns it into a faster algorithm than that obtained using DSpath versions. In practice the LA data structure made Kruskal's algorithm faster than the any of the DSpath ones, independently of the number of nodes and edges. For complete graphs LA is significantly faster as it can be observed in Figure 1. Also interesting is the fact that NRn (implemented based on the recursive version but avoiding unnecessary assignments) is substantially faster than the path compression DSpath version, NRd. The DSpath variants with path splitting, NRs, and halving, NRd, are faster then the latter, with the splitting version being slightly better. Nevertheless, the labeling version, Ls, is the fastest of all.

The *degree* of a node is the number of edges it is connected to and the *density* is the number of edges divided by the number of nodes, that is, the average degree of the nodes is $2 * density$. For graphs with about half as many

[3] The graphs were created by generating n nodes in a two-dimensional space and the edges represent the line segments between each pair of points. The associated edge weight is the euclidian distance, or the measure of the line segment, rounded down.

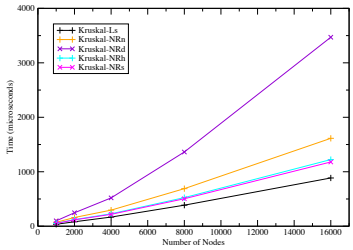


Fig. 1. Average time (μs) for running Kruskal's algorithm on complete graphs.

edges as the complete graphs (density $n/4$), like in Figure 2.a), the relative differences in time are similar but, since there are substantially less edges to be tested, the times are approximately half of the ones in Figure 1. For very sparsely connected graphs like the ones used for the tests in Figure 2.b) density value of 20 was used. The difference in average times between the three best versions - NRs, NRh, and Ls - is not so significant. Nevertheless, the LA version is still faster than the best DSpath versions. This difference is due to the number of

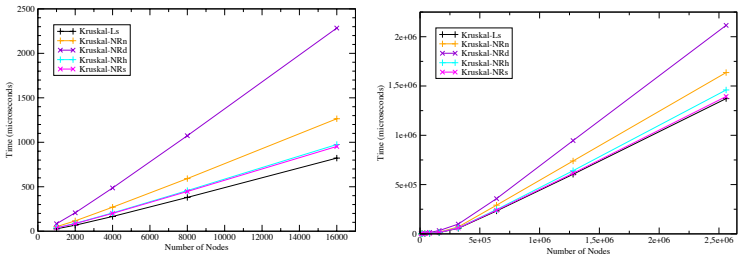
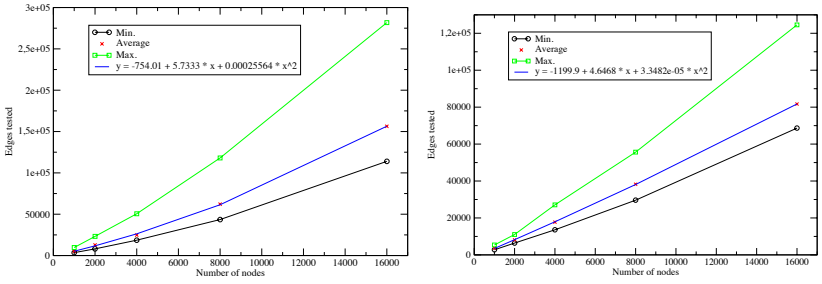


Fig. 2. Average times (μs) for Kruskal on randomly generated graphs with density : (a) $n/4$ (b) 20.

accesses made to the tree (sets) representation. Kruskal's algorithm activates the `Find` function each time an edge must be tested. The relabeling of the smallest tree to be joined also implies accessing the array fields and thus must likewise be measured. The number of accesses to the structure might be as important as the number of `Find` operations, the operation considered the most onerous on the CPU time. In order to better understand the influence of the values of the number of accesses to the structure we studied the number of structure accesses (assignments, checks, and relabels) made by the fastest three versions in the previous subsection: Kruskal-Ls, Kruskal-NRs, and Kruskal-NRh. Note

that the number of tests on edges made by any version of the Kruskal algorithm is independent of the implementation and depends only on the edge weight[4]. Figure 3 refer to measurements made for the randomly generated instances. The plots present the minimum, the average, and the maximum number of accesses. Figure 3 shows that the variability of these numbers of tests on edges, that is, of



(a) (b)

Fig. 3. Number of accesses for: complete graph instances (a); instances with density $n/4$ (b).

`Find` operations, is not of consequence. The line that presents a regression on the average numbers tends to a quadratic polynomial but the quadratic coefficient is very small specially so for the sparser graphs. Nevertheless, since the number m of `Find` operations is the double of the number of tested edges, we get $m = \mathcal{O}(n^2)$.

The worst possible case for the number of accesses are the complete graphs instances. Let us start by the analysis of the results obtained for a particular example: the 50 randomly generated instances with 8000 nodes. Since the average number of tested edges 62158.2 we chose an instance having exactly 62124 tested edges for the analysis.

Version	Accesses	Attributions
Kruskal-Ls	124248	33786
Kruskal-NRd	246442	6906
Kruskal-NRh	263864	7686
Kruskal-NRs	255491	6999

The number of relabels for the LA version was 17788 however the number of assignments includes the ones needed to merge the lists, which are 2 for each of the $(n-1)$ `Union` operations. Thus, for this 8000 nodes instance, we have $2 * 7999 = 15998$, which adds up to a total of 33786 assignments.

For the same instances, all the versions perform equal number of `Find` operations, due to the previously explained fact that they all test the same number of edges. Thus the smallest figures that appear in the previous table are the ones corresponding to the version of the Kruskal algorithm using path compression, where the number of accesses to the vector was 246442. This means that, on average, there are approximately 1.98 accesses per `Find`, significantly more than

[4] And eventually on the sorting of the edges with equal values.

1.5 which was the expected approximate value for the lower bound on the number of accesses made by any DSpath version (see Section 5). This version also achieves the lowest number of assignments, that is, the number of changes in the tree structure (namely relabels and change of parents): 6906. This is due to the fact that if a node is a root or the child of a root there is no need to change the tree, thus this figure also counts the number of times the node was a further descendent from the root. Such a small figure implies that the 1.98 accesses made by the `Find`s cannot be due to the fact that some of these operations needed more than 2 accesses. The probable cause must then be that the `Find` operations are not equiprobable but follow some other law. Remember that the `Find` oper-

	Path compression (NRd)			Path halving (NRh)			Path splitting (NRs)		
n	min	avg.	max	min	avg.	max	min	avg.	max
1000	1.97064	1.98921	2.00406	2.15561	2.27639	2.35016	2.07606	2.13351	2.17267
2000	1.97421	1.99472	1.99472	2.12667	2.21789	2.31932	2.05928	2.10021	2.14817
4000	1.97225	1.98164	1.99988	2.09345	2.17852	2.22159	2.04254	2.0816	2.1009
8000	1.96895	1.98344	1.98344	2.07476	2.1289	2.16363	2.03403	2.05839	2.07302
16000	1.97748	1.9855	1.99389	2.05739	2.09769	2.12798	2.02559	2.04412	2.05747

Table 1. Number of accesses per `Find` for Kruskal with the most efficient DSpath variants.

ation for LA only uses one access to the structure and thus the total number of accesses in LA is exactly twice the number of tested edges. The same does not apply to the path algorithms in either of the 3 different variants. Table 1 shows the number of accesses per `Find` obtained by using Kruskal's algorithm with NRd, NRs, and NRh versions. As it was expected after the particular analysis made for the 8000 nodes instances, all the values are very close to 2 accesses per `Find`, though path compression is below 2 (smallest average difference of about 0.016), path splitting is above 2 (smallest average difference of about 0.04), and path halving is the worst of the three, although for the bigger instances, the average values begin to come closer to the value of 2 (smallest distance of about 0.09). This means that, on average, the number of accesses to the structure per `Find` using a DSpath algorithm is very close to 2, and thus the double of the value achieved by LA. Finally, notice that, with the exception of NRd, for path halving, path splitting, and LA, the increase on the number of nodes leads to a decrease on the number of accesses.

Table 2 presents the minimum, the maximum and the average number of assignments made by the each of the DSpath best versions. It is an obvious fact that path compression wins again, although the average figures are closely followed by path splitting. Notoriously, the phenomenon of the increase on the number of nodes leading to a decrease on the number of assignments is now evident for the three variants. At first sight this might seem strange but it might be because the bigger the number of nodes, the bigger the probability of producing real large subtrees rapidly, implying that the most joins after that

	Path compression (NRd)			Path halving (NRh)			Path splitting (NRs)		
n	min	avg.	max	min	avg.	max	min	avg.	max
1000	1067	1228.04	1381	1171	1352.54	1513	1130	1308.18	1481
2000	2070	2354.76	2640	2313	2667.86	2941	2152	2455.66	2755
4000	3387	3815.88	4677	3775	4272.48	5328	3460	3904.88	4782
8000	6289	7042.74	7952	6944	7871.24	8963	6363	7131.62	8038
16000	12009	13389.8	14959	13239	14927.5	16520	12071	13479.2	15062

Table 2. Number of overall assignments for Kruskal with the most efficient DSpath variants.

would relabel less nodes, ie., smaller trees. On the contrary, for smaller number of nodes there could be many symmetrical sized trees for a great quantity of the iterations thus comparatively relabeling more nodes. The number of extra accesses and assignments is due to the fact that the compression of the path is not achieved as fast as with path compression. However, we have seen that splitting and halving were faster than path compression, which seems to consolidate the theoretical (and intuitive) concept that the time saved by avoiding the second pass in the `Find` operation is more than enough to compensate the time wasted by the extra accesses and assignments.

	Relabels			Total assignments			$\log n$
n	min	avg.	max	min	avg.	max	(approx.)
1000	2730.26	2996	2.508	4506	4728.26	4994	10
2000	4748	5193.78	5681	8746	9191.78	9679	11
4000	8974	9679.98	10781	16972	17678.0	18779	12
8000	16632	18255.8	20173	32630	34253.8	36171	13
16000	32512	35220.9	38058	64510	67218.9	70056	14

Table 3. Number of relabels and total assignments of nodes for Kruskal_Ls.

In Table 3 observe that the numerical results indicate that the number of relabels done by the `Union` operation and classified as being $\mathcal{O}(n \log n)$ (see Section 4) is rather smaller and very close to the best case performance $rl(n) = \mathcal{O}(n)$.

Although the LA presents a bigger number of assignments than DSpath variants, the fact that all the instances have a number of accesses about half of the ones made the latter seems to compensate, turning LA algorithm into the fastest one. The difference in absolute numbers is relevant. In the previous example, Ls made 33786 assignments while the DSpath versions made about 26700 less. However the difference on the total number of accesse between the DSpath versions and Ls is superior 120000, almost 5 times more.

8 Conclusions

This work recovers a particular modeling for the disjoint-set union problem for the implementation of efficient Kruskal's algorithm versions [1,2] – the labeling data structure.

In [5], several variants of disjoint-set data structures using union and find operations are presented. The best variants (using path compression with double passing, path halving and path splitting) are proved to be asymptotically almost linear on the number m of find operations and also asymptotical optimal.

Nevertheless, we provided evidence that when Kruskal's algorithm is implemented using the labeling data structure, LA, with a quite simple vector representation, despite its asymptotic complexity being log-linear, in practice, turns out to be faster and this is even more evident when the density of the graphs increases. The reason being the fact that this data structure simplifies immensely the `Find` operation.

Finally, most of the applications for an MST algorithm seldom require only one solution: there are applications that need to perform these algorithms thousands of times. Therefore the evidence of the differences in μs for the several data structures might turn out to be significant in temporal performance for this applications.

References

1. Kruskal, J. B.: On the shortest spanning subtree of a graph and the travelling salesman problem., Proc. Am. Math. Soc., vol.7, n.1, 48–50 (1956)
2. Gondran, M. and Minoux, M.: Graphs and algorithms. John Wiley & Sons, Chichester, UK (1984)
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press (2001)
4. Goodrich, M. T. and Tamassia, R.: Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, (2002)
5. Tarjan, R. E. and van Leeuwen, J.: Worst-case Analysis of Set Union Algorithms. J. ACM, vol. 31, n. 2, 245–281(1984)
6. Knuth, D. E.: Art of Computer Programming, Volume 1: Fundamental Algorithms (2nd Edition). Addison-Wesley Professional (1973)
7. Aho, A. V. and Hopcroft, J. E.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Boston, MA, USA (1974)
8. Aho, A. V., Hopcroft, J. E., and Ullman, J.: Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Boston, MA, USA (1983)
9. Tarjan, R. E.: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM, vol. 22, n. 2, 215–225 (1975)
10. Tarjan, R. E.: Data structures and network algorithms. SIAM, Philadelphia, PA, USA (1983)
11. Galil, Z. and Italiano, G. F.: Data structures and algorithms for disjoint set union problems. ACM Comput. Surv., vol. 23, n. 3, 319–344 (1991)
12. Fredman, M. and Saks,, M.: The cell probe complexity of dynamic data structures. STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of Computing, 345–354 (1989)