Engineering Tree Labeling Schemes: a Case Study on Least Common Ancestors[*]

Saverio Caminiti[1], Irene Finocchi[1], and Rossella Petreschi[1]

Computer Science Department, *Sapienza* University of Rome
Via Salaria, 113 - 00198 Rome, Italy
{caminiti,finocchi,petreschi}@di.uniroma1.it

Abstract. We address the problem of labeling the nodes of a tree such that one can determine the identifier of the least common ancestor of any two nodes by looking only at their labels. This problem has application in routing and in distributed computing in peer-to-peer networks. A labeling scheme using $\Theta(\log^2 n)$-bit labels has been previously presented by Peleg. By engineering this scheme, we obtain a variety of data structures with the same asymptotic performances. We conduct a thorough experimental evaluation of all these data structures. Our results clearly show which variants achieve the best performances in terms of space usage, construction time, and query time.

1 Introduction

Effective representations of large, geographically dispersed communication networks should allow the users to efficiently retrieve information about the network in a distributed and localized way. Labeling schemes provide an answer to this problem by assigning labels to the network nodes in such a way that queries can be computed alone from the labels of the involved nodes, without any extra information source. The primary goal of a labeling scheme is to minimize the maximum label length, while keeping queries fast. Adjacency labeling schemes were first introduced by Breuer and Folkman in [5, 6], and further studied in [12]. The interest in informative labeling schemes, however, was revived only more recently, after Peleg showed the feasibility of the design of efficient labeling schemes capturing distance information [16]. Since then, upper and lower bounds for labeling schemes have been proved on a variety of graph families and for a large variety of queries, including distance [2, 9, 11], tree ancestry [1, 3], flow and connectivity [14]. In spite of a large body of theoretical works, to the best of our knowledge only few experimental investigations of the efficiency of informative labeling schemes have been addressed in the literature [9, 13].

In this paper we focus on labeling schemes for answering least common ancestor queries in trees. Labeling schemes for least common ancestors can be

[*] Work partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project MainStream ("Algorithms for Massive Information Structures and Data Streams").

easily exploited to answer distance queries and are mainly useful in routing messages on tree networks, processing queries in XML search engines and distributed computing in peer-to-peer networks (see, e.g., [3, 4, 13]). In [17], Peleg has proved that for the class of n-node trees there exists a labeling scheme for least common ancestors using $\Theta(\log^2 n)$-bit labels, which is also shown to be asymptotically optimal.

Peleg's labeling scheme hinges upon two main ingredients: a decomposition of the tree into paths, and a suitable encoding of information related to such paths into the node labels. Peleg's data structure uses an *ad hoc* path decomposition as well as an *ad hoc* label structure. In this paper we first discuss different path decomposition approaches and different ways of constructing node labels, with the aim of engineering Peleg's scheme and obtaining a variety of labeling schemes for least common ancestors. We then perform a thorough experimental evaluation of all these variants, also analyzing the effects of structural properties of the input tree (such as balancing and degree) on their performances. The main findings of our experiments can be summarized as follows:

- Among different path decompositions, those that generate the smallest number of paths (with the largest average path length) appear to be preferable in order to minimize the total size of the data structure.
- A variant of Peleg's scheme proposed in [7] achieves the best performances in terms of space usage and construction time.
- Peleg's scheme, used with a minor variant of the path decomposition originally proposed in [17], exhibits the fastest query times.
- All the data structures are very fast in practice. Although node labels have size $O(\log^2 n)$, only a small fraction of the labels is considered when answering random queries: typically, no more than a constant number of words per query is read in all our experiments. However, query times slightly increase with the instance size due to cache effects.
- Variants of the data structures carefully implemented with respect to alignment issues save 20% up to 40% of the space, but increase the query times approximately by a factor 1.3 on our data sets. The space saving reduces as the instance size gets larger.

The remainder of this paper is organized as follows. In Section 2 we describe the data structures being compared, focusing on path decomposition, label structure, and query algorithms. In Section 3 we give implementation details and discuss our experimental framework. The main findings of our experimental study are presented in Section 4.

2 Labeling Schemes for Least Common Ancestors

All the tree labeling schemes that we study in this paper follow the same basic approach: the tree is decomposed into a set of node disjoint paths, that we will call *solid paths*, and information related to the highest node in each path, called *head* of the path, is suitably encoded into the node labels. In the following we will consider different path decomposition approaches, then we will describe two possible ways of designing node labels. Different combinations of these two ingredients yield different labeling schemes: one of them coincides with the tree labeling scheme for least common ancestors originally proposed by Peleg in [17].

Path Decompositions. Let T be a tree with n nodes rooted at a given node r. For any node u, we denote its parent and its level in T by $p(u)$ and $\ell(u)$, respectively. We assume that the root has level 0. We also denote by T_u the subtree of T rooted at u and by $|T_u|$ the number of its nodes. In all the decompositions, for any solid path π, we denote by $\mathit{head}(\pi)$ the node of π with smallest level. We will also say that a solid path π is an *ancestral solid path* of a node u if $\mathit{head}(\pi)$ is an ancestor of u.

Decomposition by Large Child. This decomposition hinges upon the distinction between small and large nodes: a nonroot node v with parent u is called *small* if $|T_v| \le |T_u|/2$, i.e., if its subtree contains at most half the number of nodes contained in its parents' subtree. Otherwise, v is called *large.* It is not difficult to see that any node has at most one large child: we will consider the edge to that large child, if any, as a solid edge. Solid edges induce a decomposition of the tree into solid paths: we remark that the head of any solid path π is always a small node, while all the other nodes in π must be large. Each node can have at most $\lceil \log n \rceil$ small ancestors, and thus at most $\lceil \log n \rceil$ ancestral solid paths (unless otherwise stated, all logarithms will be to the base 2).

Decomposition by Maximum Child. This is a minor variant of the previous decomposition, using a relaxed definition of large nodes: a nonroot node v with parent u is considered a *maximum child* of u if $|T_v| = \max_{w:(u,w)\in T} |T_w|$. If two or more children of u satisfy this condition, ties are broken arbitrarily. The edge to the maximum child is considered as a solid edge. We note that a large node is necessarily a maximum child; however, a maximum child exists even when all the children v of a node u are such that $|T_v| \le |T_u|/2$. All the basic properties of the decomposition by large child remain valid in this variant.

Decomposition by Rank. In this decomposition, an edge (u, v) is solid if and only if $\lceil \log |T_u| \rceil = \lceil \log |T_v| \rceil$. It is not difficult to prove that for any node u there exists at most one child v such that (u, v) is solid (see, e.g., [10, 15]). This implies that solid edges univocally partition the tree into disjoint paths. Some of these paths can consist of a single node: for instance, all the tree leaves are heads of solid paths of length 0. We remark that for all nodes v belonging to a given path π, the size of the subtree rooted at v satisfies the inequality $2^i \le |T_v| < 2^{i+1}$, for some $i \ge 0$: we will say that i is the *rank* of path π. Since the rank of any path can be at most $\lceil \log n \rceil$, it follows that each node u can have at most $\lceil \log n \rceil$ ancestral solid paths.

Label Structure and Query Algorithms. We present two different ways of constructing node labels (the two approaches are extensively described in [17] and [7], respectively). When combined with any of the path decompositions, both schemes yield labels of size $O(\log^2 n)$. We also describe how information maintained in the node labels can be used to infer the least common ancestor of any two nodes.

Peleg's scheme. The first scheme [17] is based on a depth-first numbering of the tree T: as a preprocessing step, each node v is assigned an interval $Int(v) = [DFS(v); DFS(w)]$, where w is the last descendent of v visited by the depth-first tour and $DFS(x)$ denotes the depth-first number of node x. The label of each node v of the tree is defined as $label(v) = < Int(v), list(v) >$; where $list(v)$ contains information related to all the heads $(t_1, t_2, \ldots, t_h)$ of solid paths from the root of T to v: for each head t_i, $list(v)$ contains a quadruple $(t_i, \ell(t_i), p(t_i), succ_v(t_i))$, where $succ_v(t_i)$ is the unique child of t_i on the path to node v. We remark that this is slightly different (and optimized) with respect to the scheme originally proposed in [17].

We now describe the query algorithm: given two nodes u and v, the algorithm infers their least common ancestor $z = lca(u, v)$ using only information contained in $label(u)$ and $label(v)$. By well-known properties of depth-first search, we have that for every two nodes x and y of T, $Int(x) \subseteq Int(y)$ if and only if x is a descendent of y in T: this fact can be easily exploited to check whether the least common ancestor z coincides with any of the two input nodes u and v. If this is not the case, let $(u_1, u_2, \ldots, u_h)$ and $(v_1, v_2, \ldots, v_k)$ be the heads of solid paths from the root of T to u and v, respectively: information about these heads is maintained in the node labels. The algorithm finds the least common ancestor head h, which is identified by the maximum index i such that $u_i = v_i$. If $succ_u(h) \neq succ_v(h)$, then h must be the least common ancestor. Otherwise, the algorithm takes the node of minimum level between u_{i+1} and v_{i+1}, and returns its parent as the least common ancestor. We refer to [17] for a formal proof of correctness. Here, we limit to remark that both depth-first numbering and information about successors appear to be crucial in this algorithm.

CFP's scheme. This scheme [7] avoids the use of depth-first numbers and of successors. The label of each node v of the tree is now defined as $label(v) = < isHead(v), list(v) >$. The Boolean value $isHead(v)$ discriminates whether v is the head of its solid path or not. As in Peleg's scheme, $list(v)$ contains information related to all the heads $(t_1, t_2, \ldots, t_h)$ of solid paths from the root of T to v. In this case, the information for each head is less demanding and $list(v)$ consists just of a sequence of triples: $list(v) = [\,(t_1, \ell(t_1), p(t_1)), \ldots, (t_h, \ell(t_h), p(t_h)), (v, \ell(v), p(v))\,]$; where t_1 always coincides with the root of T. The sentinel triple $(v, \ell(v), p(v))$ is not necessary when v is head of its solid path, since $t_h = v$.

We now describe the query algorithm. Given any two nodes u and v, let $(u_1, u_2, \ldots, u_h)$ and $(v_1, v_2, \ldots, v_k)$ be the heads of solid paths from the root of T to u and v, respectively. Similarly to the previous data structure, the algorithm first identifies the lowest head h which is ancestor of both u and v: let i be such that $h = u_i = v_i$. If neither u nor v coincides with h (in this trivial case it would be $lca(u, v) = h$), the algorithm searches the least common ancestor in the solid path π with head h. At this aim, it identifies two candidates c_u and c_v and returns the highest of them. Notice that node u_{i+1} is either the sentinel of $list(u)$ or the head following u_i in $list(u)$: in the former case the candidate c_u is u itself, while in the latter case the candidate is the parent of u_{i+1}. The candidate c_v is computed similarly and the algorithm returns the highest level node among c_u and c_v. We refer the interested reader to [7] for a formal proof of correctness. We remark that this algorithm compensates for the absence of depth-first intervals and successor information thanks to the use of sentinel triples.

3 Experimental Framework

In this section we describe our experimental framework, discussing implementation details of the data structures being compared, performance indicators we consider, test sets, as well as our experimental setup. All implementations have been realized by the authors in ANSI C. The full package is available over the Internet at the URL: `http://www.dsi.uniroma1.it/~caminiti/lca/`.

Data Structure Implementation Issues. We implemented six different labeling schemes, obtained by combining the three path decompositions (`rank`, `largeChild`, and `maxChild`) and the two label structures (`Peleg` and `CFP`). The labeling scheme originally proposed in [17] corresponds to using Peleg's labels together with the decomposition by large child. It can be proved that all the obtained labeling schemes guarantee maximum label size $\Theta(\log^2 n)$ for trees with n nodes.

Each scheme comes in two variants, depending on alignment issues. In the `word` variant, every piece of information maintained in the node labels is stored at word-aligned addresses: some bytes are therefore used just for padding purposes. The actual sizes of nodes labels may be larger than the size predicted theoretically, but we expect computations on node labels to be fast. In the `bit` variant, everything is 1-bit aligned: this variant guarantees a very compact space usage, but requires operations for bit arithmetics that might have a negative impact on the running times of operations.

Performance Indicators. Main objectives that we considered to evaluate the data structures include space usage, construction time, and query time. Space usage is strictly related to the length of the lists in the node labels, i.e., to the number of entries in such lists: besides the total size of the data structure (measured in MB, unless otherwise stated), we have therefore taken into account also the average and maximum list length. Other structural measures have been used to study the effect of the different path decompositions on the labeling schemes: among them, we considered the number of paths in which the tree is decomposed, the average and maximum length of paths, and the variance of path lengths.

Test Sets. Problem instances consist both of synthesized, randomly generated trees and of real test sets. We used two random tree generators with different characteristics.

Uniformly distributed trees. This generator exploits the existence of a one-to-one correspondence between labeled rooted trees on n nodes and strings of length

$n-1$: it first generates a random codeword of $n-1$ integers in the range $[1, n]$ and then applies a linear-time decoding algorithm [8] to obtain the tree. The approach guarantees that, if each integer is chosen uniformly at random in $[1, n]$, each tree will have the same probability to be generated.

Structured trees. This generator produces structured instances taking into account constraints on the degree and on the tree balancing. It works recursively and takes as input four arguments, named n, d, D, and β: n is the number of nodes of the tree T to be built; d and D are a lower and an upper bound for its degree, respectively; β is the unbalancing factor of T, i.e., a real number in $[0, 1]$ which indicates how much T must be unbalanced (the larger is β, the more unbalanced will be T).

Real test sets. Spanning trees of real networks have been obtained from data provided on the CAIDA (Cooperative Association for Internet Data Analysis) web site. Specifically, we exploited the network of Autonomous Systems monitored by the `skitter` project. We refer the interested reader to `http://www.caida.org/` for detailed information about these datasets.

Experimental Setup. Our experiments have been carried out on a workstation equipped with two Dual Core Opteron processors with 2.2 GHz clock rate, 6 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Debian (Kernel 2.6.8). All programs have been compiled through the GNU `gcc` compiler version 3.3.5 with optimization level `O3`, using the C99 revised standard of the C programming language. Unless stated otherwise, in our experiments we averaged each data point on 1000 different instances. When computing running times of query operations, we averaged the time on (at least) 10^6 random queries.

4 Experimental Results

In this section we summarize our main experimental findings. We performed experiments using a wide variety of parameter settings and instance families, always observing the same relative performances of the data structures. Due to the lack of space, we do not explicitly report results on real data in this extended abstract: all measurements on these data sets completely confirm the results obtained on synthetic instances.

Path Decomposition. Our first aim was to analyze the effects of different path decomposition strategies on the size of node labels. A typical outcome of our experiments on trees generated uniformly at random is exemplified in Table 1. With respect to all measures, `maxChild` appears to be slightly preferable than `largeChild` and considerably better than `rank`. Consider first the structural measures: among the three decompositions, `maxChild` generates the smallest number of solid paths. Paths are therefore longer on the average, and their lengths exhibit a higher variance. On the opposite side, the number of paths generated by `rank` is almost twice as large for the parameter setting of this experiment, and their length is almost twice as small.

	maxChild	largeChild	rank
Number of paths	3678739	4172966	6803270
Average path length (and variance)	2.72(73.7)	2.4(61.2)	1.47(7.9)
Maximum path length	15352	15351	6346
Average list length (and variance) for Peleg	5.72(2.16)	5.89(2.32)	12.40(10.58)
Maximum list length for Peleg	15	15	24
Data structure size for Peleg	1179	1203	2199
Average list length (and variance) for CFP	6.36(2.06)	6.47(2.18)	12.7(10.44)2
Maximum list length for CFP	15	15	24
Data structure size for CFP	1033	1045	1761

Table 1. Comparison of path decompositions. The results of this experiment are averaged over 500 random trees with $n = 10^7$ nodes. Only the `word` variant of the data structures is reported.

Additional experiments were aimed at analyzing the effects of structural properties of the tree on the path decomposition: in all these tests, the relative ranking among the three strategies was always the same observed on uniformly distributed trees. The graphical outcome of two such experiments, obtained by increasing tree unbalancing and maximum degree, is reported in Figure 1. As the tree becomes more and more unbalanced, the advantages of using the `maxChild` decomposition drop: the number of solid paths obtained by `largeChild` and `rank` indeed decreases and, conversely, the average path length increases (see Figure 1a and Figure 1c). To explain this, let u be any node and let v be the child of u that is root of the maximum size subtree: the more T_u is unbalanced, the more $|T_u|$ and $|T_v|$ are close to each other and the edge (u, v) is likely to be solid. This reasoning cannot be applied to the `maxChild` strategy, according to which any internal node has always a solid child: for this reason curves related to `maxChild` exhibit an almost constant trend. Let us now analyze the effect of increasing the degree. Let T_1 and T_2 be two trees generated with the same fixed unbalancing factor β ($\beta = 0.9$ in the right column of Figure 1) and maximum degrees $D_1 < D_2$: for all strategies, we expect the number of solid paths in T_2 to be larger than the number of solid paths in T_1, since a larger degree implies a larger number of heads (not only among the children, but among all the descendants of each node). This intuition has been confirmed by the experiments with increasing maximum degree for all the decompositions, and explains the trend of the curves in Figure 1b and Figure 1d.

Size Comparison. Our next aim is to evaluate the requirements of `Peleg`'s and `CFP`'s schemes with respect to the space usage. Besides the total size of the data structure, we measured also the average number of solid heads in the lists associated to tree nodes (average list length). We performed experiments varying both structural properties of the input tree and the instance size.

At a first sight, it might appear that the average list length should be inversely proportional to the average path length: if paths are shorter on the average, the

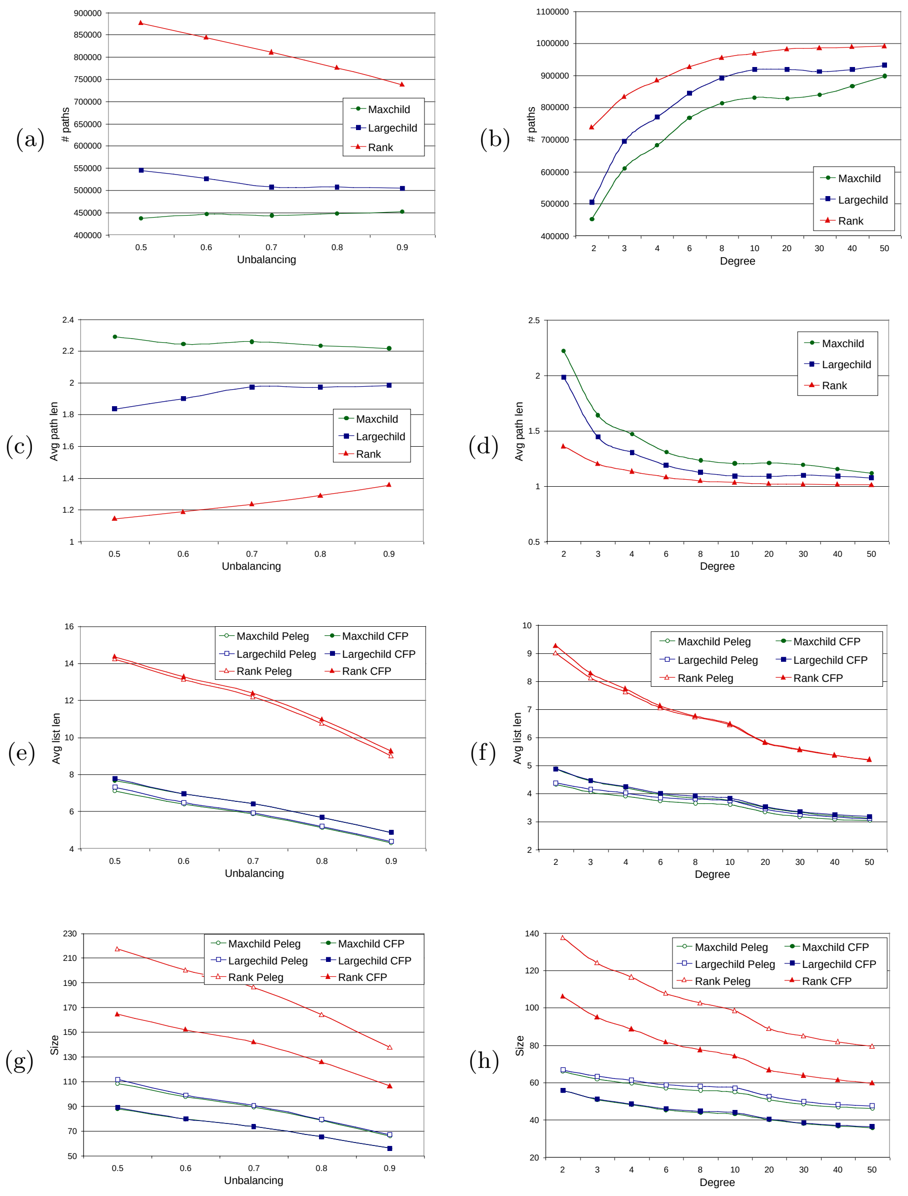


Fig. 1. Experimental results on structured trees with $n = 10^6$ nodes: increasing unbalancing factor β (left column, $d = 2 = D$) and increasing degree (right column, $d = D$, $\beta = 0.9$).

number of paths in any root-to-leaf path is expected to be larger, and so is the number of heads in node labels (both for `Peleg` and `CFP`). While this was confirmed by the experiments on uniformly distributed trees (see Table 1), it is

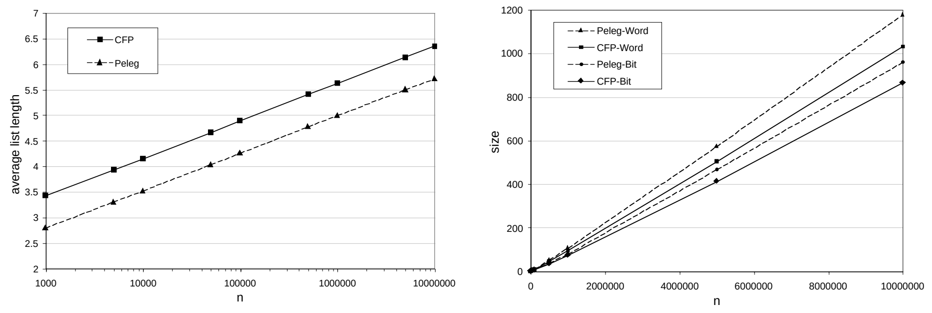


Fig. 2. Size comparison for `Peleg`'s and `CFP`'s schemes on uniformly distributed random trees: average list length and total size, measured in MB.

not necessarily the case on more structured instances: in particular, both the average path length (Figure 1d) and the average list length (Figure 1f) decrease as the maximum degree increases. A more refined analysis suggests that the topology of the tree should also be taken into account, and in particular the average height of tree nodes should be considered: the deeper a node, the larger the number of heads above it can be. As far as our generator works, trees with larger degree have smaller average node height and, according to Figure 1f, the effect of such smaller height appears to dominate on the shorter length of solid paths.

The total size of the data structure is directly proportional to the average list length, and curves related to these two measures exhibit the same trend (see Figure 1g and Figure 1h). However, it is worth observing that the data structure size in the case of `CFP` is considerably smaller than `Peleg`'s size, in spite of a slightly larger average list length. This is also evident from Figure 2, that reports on results obtained using the `maxChild` path decomposition on uniformly distributed random trees with a number of nodes increasing from 10^3 to 10^7 (from this point on we will omit the discussion of `rank` and `largeChild`, since `maxChild` proved to be consistently better in all the tests described so far). The smaller data structure size in the case of `CFP` depends on the fact that the lists are made of triples, instead of quadruples: the smaller list length in `Peleg`'s scheme (due to the absence of sentinel triples) is not sufficient to compensate for the presence of one more information in each element of the lists. We remark that lists are very short in practice for both schemes: they contain on the average 3 up to 6 elements for the data sets considered in this experiment. This value is very close to $\log_{10} n$, showing that the constant factors hidden by the asymptotic notation in the theoretical analysis are very small for the `maxChild` path decomposition. In Figure 2 we also distinguish between the `bit` and `word` versions of the data structures (there is no such difference with respect to the average list length): as expected, for both schemes the `bit` versions can considerably reduce the space usage. We will analyze further these data later in this section.

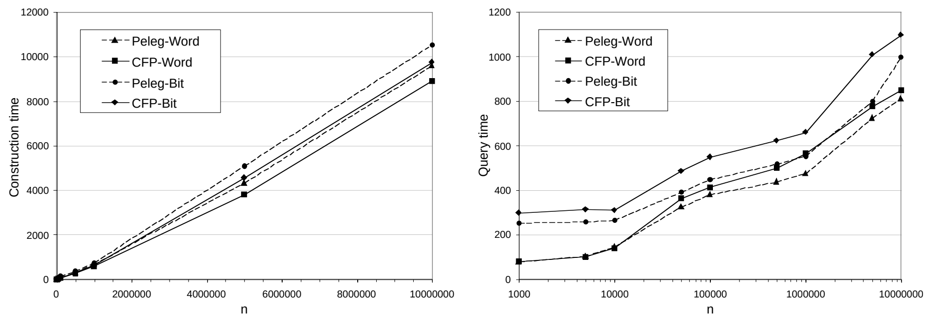
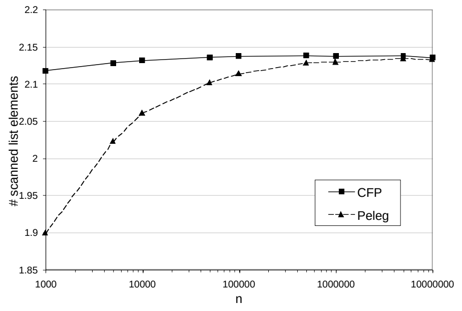


Fig. 3. Running time comparison for `Peleg`'s and `CFP`'s schemes on uniformly distributed random trees: construction time (in milliseconds) and average query time (in milliseconds per 10^6 queries).

Running Times. According to the theoretical analysis, the construction times and the query times for the different labeling schemes are asymptotically the same. A natural question is whether this is the case also in practice. Our experiments confirmed the theoretical prediction only in part, showing that the constant factors hidden by the asymptotic notation can be rather different for `Peleg`'s and `CFP`'s schemes. The charts in Figure 3, for instance, have been obtained on the same data sets used for the test reported in Figure 2: these charts show that `Peleg` is slower than `CFP` when considering initialization time, but faster when considering query times. The `bit` versions of the data structures are always slower than the corresponding `word` versions.

In order to explain the larger construction time of `Peleg`'s scheme, notice that `Peleg` makes use of a depth-first numbering of the tree, that is instead avoided by `CFP`: all the other operations performed by the initialization algorithms (i.e., path decomposition and list construction) are instead very similar. We also recall that Peleg's data structure is larger than `CFP`, and the size of a data structure is clearly a lower bound on its construction time. The larger amount of information maintained by `Peleg` in the list of each node is however efficiently exploited in order to get faster query times: as an example, if one of the two input nodes is ancestor of the other, the query algorithm used by `CFP` needs to scan the beginning of the nodes' lists, while the depth-first intervals directly provide the answer in the case of `Peleg` data structure.

To get a deeper understanding of the query times, we also measured the average number of list elements scanned by the query algorithms during a sequence of operations. This number turns out to be very small both for `Peleg` and for `CFP`, as shown by the left chart reported in Figure 4: on the average, slightly more than 2 elements are considered in each query even on the largest instances. `Peleg` considers less elements than `CFP`, especially for small values of n: on small trees, two nodes taken uniformly at random have indeed a higher probability to be one ancestor of the other, and for all these queries `Peleg` can avoid to scan the list at all, as we observed above. Quite surprisingly, however, for the largest



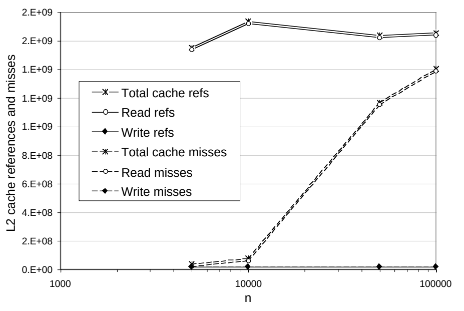
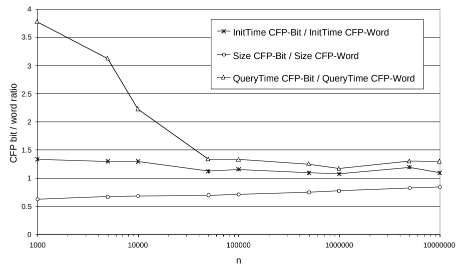


Fig. 4. Average number of list elements scanned by the query algorithms on uniformly distributed random trees (left chart); number of references to L2 cache and number of cache misses incurred by the CFP query algorithm on the same dataset (right chart).



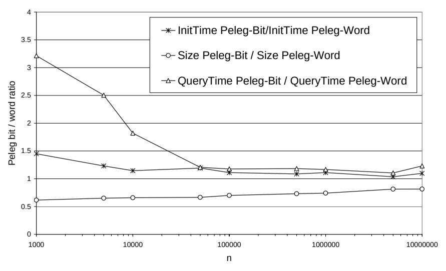


Fig. 5. Space/time saved by the bit/word versions: CFP (left chart) and Peleg (right chart). Tests are made on uniformly distributed random trees.

values of n the number of scanned list elements remains almost constant for both data structures: this seems to be in contrast with the fact that the query times increase (see Figure 3), and suggests that the larger running times may be mainly due to cache effects. To investigate this issue, we used the valgrind profiler to conduct a preliminary experimental analysis of the number of cache misses incurred by the query algorithms: the outcome of one such experiment, related to CFP, is reported in the right chart of Figure 4. The experiment confirms that the total number of cache references does not increase substantially with n (in agreement with the result on the number of scanned list elements), while the number of L2 cache read misses increases sharply, thus justifying the larger query times.

Trading Space for Time. The experimental results discussed up to this point show that the bit versions of the data structures require more space than the corresponding word versions, but have larger construction and query times. In Figure 5 we summarize the space-time tradeoffs, both for Peleg and for CFP. The charts show that, for all measures, the differences between bit and word versions tend to decrease as the instance size increases: this depends on the fact that, as

n increases, the value $\log n$ becomes progressively closer to the word size specific of the architecture, and therefore the number of bits wasted by the `word` versions becomes smaller. The size of the `bit` versions ranges approximately from 60% up to 80% of the size of the `word` versions on our data sets. On the other side, construction and query times of the `bit` versions are approximately 1.3 times higher than the `word` versions for the largest values of n (for small values of n the ratio is even larger).

References

1. S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo, and T. Rauhe. Compact labeling schemes for ancestor queries. *SIAM J. on Computing*, 35(6), 1295–1309, 2006.
2. S. Alstrup, P. Bille, and T. Rauhe. Labeling schemes for small distances in trees. *SIAM J. on Discrete Mathematics*, 19(2), 448–462, 2005.
3. S. Alstrup, C. Gavoille, H. Kaplan and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In Proc. ACM SPAA'02, 258–264, 2002.
4. N. Bonichon, C. Gavoille, and A. Labourel. Short labels by traversal and jumping. In Proc. SIROCCO'06, 143–156, 2006.
5. M.A. Breuer. Coding the vertexes of a graph. *IEEE Transactions on Information Theory*, IT-12, 148–153, 1966.
6. M.A. Breuer and J. Folkman. An unexpected result on coding the vertices of a graph. *J. of Mathematical Analysis and Applications*, 20, 583–600, 1967.
7. S. Caminiti, I. Finocchi, and R. Petreschi. Concurrent data structures for lowest common ancestors. Manuscript available from the authors upon request, 2008.
8. S. Caminiti, I. Finocchi, and R. Petreschi. On coding labeled trees. *Theoretical Computer Science*, 382(2), 97–108, 2007.
9. E. Cohen, E. Halperin, H. Kaplan and U. Zwick. Reachability and Distance Queries via 2-hop Labels. In Proc. ACM-SIAM SODA'02, 937–946, 2002.
10. R. Cole and R. Hariharan. Dynamic LCA Queries on Trees. *SIAM J. on Computing*, 34(4), 894–923, 2005.
11. C. Gavoille, D. Peleg, S. Perennes and R. Raz. Distance labeling in graphs. In Proc. ACM-SIAM SODA'01, 210–219, 2001.
12. S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. In Proc. ACM STOC'88, 334–343, 1988.
13. H. Kaplan, T. Milo and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In Proc. ACM-SIAM SODA'02, 954–963, 2002.
14. M. Katz, N.A. Katz, A. Korman and D. Peleg. Labeling schemes for flow and connectivity. *SIAM J. on Computing*, 34(1), 23–40, 2004.
15. T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In Proc. ACM-SIAM SODA'07, 565–574, 2007.
16. D. Peleg. Proximity-preserving labeling schemes and their applications. In Proc. WG'99, 30-41, 1999.
17. D. Peleg. Informative labeling schemes for graphs. *Theoretical Computer Science*, 340, 577–593, 2005. Preliminary version in Proc. MFCS'00, LNCS 1893, 579–588, 2000.