Algorithmica (1997) 18: 263–270

Algorithmica

A Simpler Minimum Spanning Tree Verification Algorithm

V. King[1]

Abstract. The problem considered here is that of determining whether a given spanning tree is a minimal spanning tree. In 1984 Komlós presented an algorithm which required only a linear number of comparisons, but nonlinear overhead to determine which comparisons to make. We simplify his algorithm and give a linear-time procedure for its implementation in the unit cost RAM model. The procedure uses table lookup of a few simple functions, which we precompute in time linear in the size of the tree.

Key Words. Minimum spanning tree, Verification.

1. Introduction. The problem of determining whether a given spanning tree in a graph is a minimal spanning tree has been studied by Tarjan [6], Komlós [4], and most recently by Dixon *et al*. [1]. Tarjan's 1979 algorithm uses path compression and gives an almost linear running time. Komlós's algorithm was the first to use a linear number of comparisons, but no linear-time method of deciding which comparisons to make has been known. Indeed, a linear implementation of this algorithm was not thought possible, see [4] and [1]. The only known linear-time algorithm for this problem [1] combines the techniques of both [6] and [4], using the Komlós algorithm to process small subproblems via preprocessing and table lookup.

These verification methods and the method presented here use the fact that a spanning tree is a minimum spanning tree iff the weight of each nontree edge $\{u, v\}$ is at least the weight of the heaviest edge in the path in the tree between u and v. These methods find the heaviest edge in each such path for each nontree edge $\{u, v\}$ in the graph, and then compare the weight of $\{u, v\}$ to it.

The "tree-path" problem of finding the heaviest edges in the paths between specified pairs of nodes ("query paths") arises in the recent randomized minimum spanning tree algorithm of Karger *et al*. [3]. That algorithm is the first to compute the minimum spanning tree in linear expected time, where the only operations allowed on edge weights are binary comparisons. The solution to the tree-path problem is the most complicated part of these randomized algorithms, which are otherwise fairly simple.

The Komlós algorithm is simplified by use of the following observation: If T is a spanning tree, then there is a simple $O(n)$ algorithm to construct a full branching tree B with no more than $2n$ edges and the following property:

Let $T(x, y)$ denote the set of edges in the path in T from node x to node y, and let $B(x, y)$ denotes the set of edges in the path in B from leaf x to leaf y.

[1] Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada V8W 3P6. val@csr.uvic.ca. This work was funded by an NSERC grant.

Received July 13, 1995; revised January 29, 1996. Communicated by Ming Y. Kao.

The weight of the heaviest edge in $T(x, y)$ is the weight of the heaviest edge in $B(x, y)$.

Therefore it suffices to use the version of the Komlós algorithm for full branching trees only, which is much simpler than his algorithm for general trees.

The second part of this paper is to show that this portion of Komlós's algorithm has a linear-time implementation using table lookup of a few simple functions. These tables can be constructed in time linear in the size of the tree. As in Dixon *et al.*'s algorithm, the model of computation is a unit cost RAM with word size $\Theta(\log n)$. The only operations used on edge weights are binary comparisons.

In contrast, Dixon *et al.*'s algorithm separates the tree into a large subtree and many "microtrees" of size $O(\lg \lg n)$. Path compression is used on the large subtree. The comparison decision tree needed to implement Komlós's strategy for each possible configuration of microtree and possible set of query paths in the microtree is precomputed and stored in a table. Each microtree, together with its query paths in the input spanning tree, is encoded and then the table is used to look up the appropriate comparisons to make.

In the next section the construction of B is described, and the property of B is proved. In Section 3 we restate Komlós's algorithm for determining the maximum weighted edge in each of m paths of a full branching tree and describe its implementation.

2. Boruvka Tree Property.

Let T be a spanning tree with n nodes. Tree B is the tree of the components that are formed when the Boruvka algorithm for finding a minimum spanning tree is applied to T.

The Boruvka algorithm, as applied to a tree $T = (V, E)$ is as follows (see [7]): Initially there are n blue trees consisting of the nodes of V and no edges.

Repeat until there is one blue tree, i.e., T: For each blue tree, select a minimum weight edge incident to it. Color all selected edges blue.

Each repetition of these instructions is referred to as a phase. We construct tree B with nodeset W and edgeset F, by adding nodes and edges to B after each phase of the algorithm, so that there is a 1–1 correspondence between the nodes of B and the blue trees created during all the phases of the algorithm.

For each node $v \in V$ of T, we create a leaf $f(v)$ of B. Let A be the set of blue trees which are joined into one blue tree t in a phase i. Then we add a new node $f(t)$ to W and add $\{\{f(a), f(t)\} | \text{for all } a \in A\}$ to F. Each edge $\{f(a), f(t)\}$ is labeled with the weight of the edge selected by a in phase i.

Note that B is a full branching tree, i.e., it is rooted and all leaves are on the same level and each internal node has at least two children.

Since T is a tree, B can be constructed in $O(n)$ time. This may be seen as follows: The cost of executing each phase is proportional to the number of uncolored edges in the tree during that phase. The number of uncolored edges is one less than the number of blue trees, since T is a tree. Finally, the number of blue trees drops by a factor of at least two after each phase.

For any tree T, let $T(x, y)$ denote the set of edges in the path in T from node x to node y.

We prove the following theorem:

THEOREM 1. *Let T be any spanning tree and let B be the tree constructed as described above. For any pair of nodes x and y in T, the weight of the heaviest edge in $T(x, y)$ equals the weight of the heaviest edge in $B(f(x), f(y))$.*

PROOF. We denote the weight of an edge e by $w(e)$. First we show that for every edge $e \in B(f(x), f(y))$, there is an edge $e' \in T(x, y)$ such that $w(e') \geq w(e)$.

Let $e = \{a, b\}$ and let a be the endpoint of e which is farther from the root. Then $a = f(t)$ for some blue tree t which contains either x or y, but not both, and $w(e)$ is the weight of the edge selected by t.

Let e' be the edge in $T(x, y)$ with exactly one endpoint in t. Since t had the option of selecting e', $w(e') \geq w(e)$, which concludes the first part of the proof.

It remains to show the following:

CLAIM 0.1. *Let e be a heaviest edge in $T(x, y)$. Then there is an edge of the same weight in $B(f(x), f(y))$.*

We assume for simplicity that there is a unique heaviest edge. The proof can be easily extended to the general case.

If e is selected by a blue tree which contains x or y, then an edge in $B(f(x), f(y))$ is labeled with $w(e)$. Assume that, on the contrary, e is selected by a blue tree which does not contain x or y. This blue tree contained one endpoint of e and thus one intermediate node on the path from x to y. Therefore it is incident to at least two edges on the path. Then e is the heavier of the two, and is not selected, giving a contradiction . □

3. Komlós's Algorithm for a Full Branching Tree. For a full branching tree of weighted edges with n nodes, and m query paths between pairs of leaves, Komlós has shown a simple algorithm to compute the heaviest edge on the path between each pair with $O(n \log((m + n)/n))$ comparisons. He breaks up each path into two half-paths extending from the leaf up to the lowest common ancestor of the pair and finds the heaviest edge in each half-path, as follows:

Let $A(v)$ be the set of the paths which contain v restricted to the interval $[root, v]$.

Starting with the root, descend level by level and at each node v encountered, the heaviest edge in each path in the set $A(v)$ is determined, as follows.

Let p be the parent of v. Assume we know the heaviest edge in each path in the set $A(p)$. Note that the ordering of the weights of these heaviest edges can be determined by the length of their respective paths, since for any two paths s and t in $A(p)$, path s includes path t or vice versa. Let $A(v|p)$ be the set of the restrictions of each of the paths in $A(v)$ to the interval $[p, root]$. Since $A(v|p) \in A(p)$, the ordering of the weights of the heaviest edges in $A(v|p)$ is known. To determine the heaviest edge in each path in $A(v)$, we need only compare $w(\{v, p\})$ to each of these weights. This can be done by using binary search. Komlós shows that $\sum_{v \in T} \lg|A(v)| = O(n \log((m+n)/n))$, which gives the upper bound on the number of comparisons needed to find the heaviest edge in each half-path. Then the heaviest edge in each query path is determined with one additional comparison per path.

4. Implementation of Komlós's Algorithm. The implementation of Komlós's algorithm requires the use of a few simple functions on words of size $O(\log n)$, such as a shift by a specified number of bits, the bit-wise OR of two words, $\lfloor \log n \rfloor$, the multiplication of two words, and a few more functions which are less conventional and will be described below. All these functions can be precomputed in $O(n)$ time and stored in a table where they can be accessed in unit time. First, we present a description of the data structures we use, followed by a high-level description of the algorithm, and then its implementation details.

4.1. *Data Structures*. Let *wordsize* be the size of a word, which we assume to be $\lceil \lg n \rceil$ bits.

Node Labels and Edge Tags. Following a modification of the scheme of Schieber and Vishkin [5], we label the nodes with a $\lceil \lg n \rceil$ bit label and the edges with an $O(\log \log n)$ bit tag so that:

Label Property. Given the tag of any edge e and the label of any node on the path from e to any leaf, e can be located in constant time.

The labels are constructed as follows: Label the leaves 0, 1, 2, ..., as encountered in a depth-first traversal of the tree. Label each internal node by the label of the leaf in its subtree which has the longest all 0's suffix.

For each edge e, let v be its endpoint which is farther from the root and let $distance(v)$ be v's distance from the root and $i(v)$ be the index of the rightmost 1 in v's label. Then the *tag* of e is a string of $tagsize = O(\lg \lg n)$ bits given by $\langle distance(v), i(v) \rangle$.

We sketch the argument (see [5]) that the Label Property holds. It is not hard to see that the label of an ancestor of a node w is given by a prefix of the label of w possibly followed by a 1 and then all 0's. Also, nodes with the same label are connected by a path up the tree. Hence the label of w and the position of the rightmost 1 in an ancestor's label determine the ancestor's label, while its distance from the root uniquely determines the ancestor's identity, among those nodes with the same label. Once the lower endpoint v of an edge e is found, then e is the unique edge from v to its parent.

LCA. For each node v, $LCA(v)$ is a vector of length *wordsize* whose ith bit is 1 iff there is a path in $A(v)$ whose upper endpoint is at distance i from the root. That is, there is a query path with exactly one endpoint contained in the subtree rooted at v, such that the lowest common ancestor of its two endpoints is at distance i from the root. *LCA* is stored in a single word.

BigLists and smallLists. For any node v, the ith longest path in $A(v)$ will be denoted by $A_i(v)$. The weight of an edge e is denoted $w(e)$. Recall that the set of paths in $A(v)$ restricted to $[a, root]$ is denoted $A(v|a)$. Call a node v *big* if $|A(v)| > wordsize/tagsize$; otherwise v is *small*.

For each big node v, we keep an ordered list whose ith element is the tag of the heaviest edge in $A_i(v)$, for $i = 1, \ldots, |A(v)|$. This list is a referred to as $bigList(v)$.

We may similarly define $bigList(v|a)$ for the set of paths $A(v|a)$. $BigList(v)$ is stored in $\lceil |A(v)|/(wordsize/tagsize)\rceil = O(\log\log n)$ words.

For each small v, let a be the nearest big ancestor of v. For each such v, we keep an ordered list, $smallList(v)$, whose ith element is either the tag of the heaviest edge e in $A_i(v)$, or if e is in the interval $[a, root]$, then the j such that $A_i(v|a) = A_j(a)$. That is, j is a pointer to the entry of $bigList(a)$ which contains the tag for e. Once a tag appears in a *smallList*, all the later entries in the list are tags. For each small v, we keep a pointer to the first tag in its *smallList*. *SmallList* is stored in a single word.

4.2. *The Algorithm.* The goal is to generate $bigList(v)$ or $smallList(v)$ in time proportional to $\log|A(v)|$, so that time spent implementing Komlós's algorithm at each node does not exceed the worst case number of comparisons needed at each node. We show that if v is big, then the implementation time is $O(\log\log n)$, and if v is small, it is $O(1)$.

Initially, $A(root) = \emptyset$. We proceed down the tree, from the parent p to each of the children v. Depending on $|A(v)|$, we generate either $bigList(v|p)$ or $smallList(v|p)$. We then compare $w(\{v, p\})$ to the weights of these edges, by performing binary search on the list, and insert the tag of $\{v, p\}$ in the appropriate places to form $bigList(v)$ or $smallList(v)$. We continue until the leaves are reached.

Let v be any node, p is its parent, and a its nearest big ancestor. To compute $A(v|p)$: There are two cases if v is small:

- If p is small, we create $smallList(v|p)$ from $smallList(p)$ in $O(1)$ time.
- If p is big, we create $smallList(v|p)$ from $LCA(v)$ and $LCA(p)$ in $O(1)$ time.

If v is big:

- If v has a big ancestor, we create $bigList(v|a)$ from $bigList(a)$, $LCA(v)$, and $LCA(a)$ in $O(\lg\lg n)$ time.
  — If $p \neq a$, then we create $bigList(v|p)$ from $bigList(v|a)$ and $smallList(p)$ in time $O(\lg\lg n)$.
- If v does not have a big ancestor, then $bigList(v|p) \leftarrow smallList(p)$.

To insert a tag in its appropriate places in the list:

- Let $e = \{v, p\}$, and let i be the rank of $w(e)$ compared with the heaviest edges of $A(v|p)$. Then we insert the tag for e in positions i through $|A(v)|$, into our list data structure for v, in time $O(1)$ if v is small, or $O(\log\log n)$ if v is big.

4.3. *Implementation Details.* The computation of the *LCA*s is straightforward. First, we compute all lowest common ancestors for each pair of endpoints of the m query paths using an algorithm that runs in time $O(n+m)$, see [5] or [7]. We form the vector $LCA(l)$ for each leaf l using this information, and then form the vector $LCA(v)$ for a node at distance i from the root by ORing together the *LCA*s of its children and setting the jth bits to 0 for all $j \geq i$.

To implement the remaining operations, we need to preprocess a few functions so that we may do table lookup of these functions. We define a *subword* to be *tagsize* bits and $swnum = \lfloor wordsize/tagsize\rfloor$, i.e., *swnum* is the maximum number of subwords stored in a word. Each input and each output described below are stored in single words. The symbol $\cdot$ denotes "concatenated with."

select$_r$ takes as input $I \cdot J$, where I and J are two strings r bits. It outputs a list of bits of J which have been "selected" by I, i.e., let $\langle k_1, k_2, \ldots \rangle$ be the ordered list of indices of those bits of I whose value is 1. Then the list is $\langle j_{k_1}, j_{k_2}, \ldots \rangle$ where j_{k_i} is the value of the k_ith bit of J.

selectS$_r$ takes as input $I \cdot J$, where I is a string of no more than r bits, no more than *swnum* of which are 1, and J is a list of no more than *swnum* subwords. It outputs a list of the subwords of J which have been "selected" by I, i.e., let $\langle k_1, k_2, \ldots \rangle$ be the ordered list of indices of those bits of I whose value is 1. Then the list is $\langle j_{k_1}, j_{k_2}, \ldots \rangle$ where j_{k_i} is the k_ith subword of J.

weight$_r$ takes as input a string of length r and outputs the number of bits set to 1.

index$_r$ takes an r bit vector with no more than h 1's and outputs a list of subwords containing the indices of the 1's in the vector.

*subword*1 is a constant such that for $i = 1, \ldots, swnum$, the $(i * tagsize)$th bit is 1 and the remaining bits are 0 (i.e., each subword is set to 1).

For each of these functions, it is not hard to see that the preprocessing takes $O(n)$ time, when the size of the input is no greater than $\lg n + c$ for c a constant. A table for all inputs of length r can be built by first building a table for inputs of size $r/2$, looking up the result for the two halves, and, in constant time, putting the results together to form the entry.

For example, for $index_r$, if a table is built for $index_{r/2}$, then the table for input strings of size r can be easily constructed, in a constant number of operations per entry, as follows: Let I be the first half of the input and let J be its second half. Add $weight_{r/2}(I)$ to each subword of $index_{r/2}(J)$ by adding $weight_{r/2}(I) * subword1$ to it. Let L be the string formed. Then concatenate the first $weight_{r/2}(I)$ subwords of $index_{r/2}(I)$ with the first $weight_{r/2}(J)$ subwords of L.

Recall that the wordsize is $\lceil \lg n \rceil$. We cannot afford to build a table for $select_{wordsize}$ and $selectS_{wordsize}$ which takes inputs of 2*wordsize* bits, since the table would be too large. However, as explained above, we can compute these functions as needed in constant time using table lookups of those functions on input size *wordsize*/2 as described above.

We can now perform the operations needed for the data structures. (We omit the subscripts of the functions below, since they can be easily inferred from the size of their inputs.) We illustrate these operations with examples where $wordsize = 8$; $tagsize = 3$.

1. *Determine* $|A(v)|$:
   - $|A(v)| = weight(LCA(v))$.

   Example: if $LCA(v) = (01101110)$, then $|A(v)| = 5$.
2. *Create smallList*$(v|p)$ *from smallList*(p):
   - $L \leftarrow select((LCA(p), LCA(v))$.
   - $smallList(v|p) \leftarrow selectS(L, smallList(p))$.

   Example: Let $LCA(v) = (01001000)$, $LCA(p) = (11000000)$. Let $smallList(p)$ be (t_1, t_2). Then $L = select(1100000, 01001000) = (01)$; and $smallList(v|p) = selectS((01), (t_1, t_2)) = (t_2)$.

3. *Create smallList*$(v|p)$ *from LCA*(v) *and LCA*(p)
   - $smallList(v|p) \leftarrow index(select(LCA(p), LCA(v)))$.

   Example: Let $LCA(p) = (01101110)$ and $LCA(v) = (01001000)$. Then $smallList(v|p) = index(select((01101110), (01001000))) = index(10100) = (1, 3)$. (If $bigList(p) = (t_1, t_2, t_3, t_4, t_5)$, then the first and second entries of $smallList(v|p)$ are pointers to t_1 and t_3, respectively.)
4. *Insert tag t into positions i to j of smallList*$(v|p)$ *to form smallList*(v):
   - Concatenate the first $i-1$ subwords of $smallList(v|p)$ with the i through j subwords of $t * subword1$.

   Example: Let $smallList(v|p) = (1, 3)$ as in the example above. Then t is the tag of $\{v, p\}$. To put t into positions 1 to $j = |A(v)| = 2$, we compute $t * subword1 = t * 00100100 = (t, t)$ followed some extra 0 bits, which are discarded to get $smallList(v|p) = (t, t)$.
5. *Create bigList*$(v|a)$ *from bigList*(a), *LCA*(v), *and LCA*(a):
   - Let $L = select(LCA(a), LCA(v))$.
   - Partition L into strings L_i of *swnum* consecutive bits, and store each L_i in a word. (The last string may have fewer bits.)
   - Partition $bigList(a)$ into words, each containing *swnum* subwords. (The last may have fewer subwords.) Let $b_i(a)$ represent the ith word of $bigList(a)$.
   - For each string L_i, do $selectS(L_i, b_i(a))$.
   - Concatenate the outputs to form $bigList(v|a)$.

   Example: Let $LCA(a) = (01101110)$, $LCA(v) = (00100101)$, and let $bigList(a)$ be $(t_1, t_2, t_3, t_4, t_5)$. Then $L = (01010)$; $L_1 = (01)$, $L_2 = (01)$, and $L_3 = (0)$; $b_1 = (t_1, t_2)$, $b_2 = (t_3, t_4)$, $b_3 = (t_5)$. Then $(t_2) = selectS((01), (t_1, t_2))$; $(t_4) = selectS((01)(t_3, t_4))$; and $() = selectS(t_5)$. Thus $bigList(v|a) = (t_2, t_4)$.
6. *Create bigList*$(v|p)$ *from bigList*$(v|a)$ *and smallList*(p) *where p is the parent of v and* $p \neq a$:
   - Let f be the first subword of $smallList(p)$ which contains a tag, rather than a pointer. Replace all subwords in positions f or higher with $smallList(p)$.

   (Note that $A(v|p) = A(p)$ since this case only arises when $|A(v)| > |A(p)|$, so $smallList(v|p) = smallList(p)$.)

   Example: Using a and v from the previous example, we have $bigList(v|a) = (t_2, t_4)$. Suppose $smallList(p) = (2, t')$. (Here, 2 is pointer to the second item in $bigList(a)$ and t' is the tag of some edge below a in the tree.) Then $bigList(v|p) = (t_2, t')$.
7. *Insert the tag of* $\{v, p\}$ *into the appropriate positions of bigList*$(v|p)$ *to form bigList*(v):
   - Similar to item (2) above but must be done for each word in the list.

4.4. *Analysis*. When v is small, the cost of the overhead for performing the insertions by binary search is a constant. When v is big, $|A(v)|/(wordsize/tagsize) = \Omega(\log n/\log\log n)$, the cost of the overhead is $O(\lg\lg n)$. Hence the implementation cost is $O(\lg|A(v)|)$, which is proportional to the number of comparisons needed by the Komlós algorithm to find the heaviest edges in $2m$ half-paths of the tree in the worst case. Summed over all nodes, this comes to $O(n\log((m+n)/n))$ as Komlós has shown.

The only additional costs are in forming the *LCA*s which take $O(m + n)$ and in processing the tables which takes $O(n)$, and comparing the heaviest edges in each half-path, which takes $O(m)$.

Finally, to complete the minimum spanning tree verification algorithm, the weight of each nontree edge is compared with the weight of the heaviest tree edge in the tree path connecting its endpoints, for an additional $O(m)$ cost.

5. Conclusion and Open Problems. We have reduced Komlós's algorithm to the simpler case of the full branching tree. We have also devised a novel data structure which gives the first algorithm with linear-time overhead for its implementation.

It is still an open question whether a linear-time algorithm can be found for a pointer machine. Such a result would imply a linear-time algorithm for a pointer machine that can compute the lowest common ancestor. None is known for that problem which seems easier.

Given a static tree, Schieber and Vishkin's lowest common ancestor algorithm can process on-line query paths in constant time for each. An open problem is to solve the tree-path problem in constant time per query path, where the query paths are given on-line.

The functions we use are in some sense natural. It is possible that they may be useful for implementing other algorithms which are not known to have linear implementations or whose implementations involve more specialized table lookup functions, as Dixon *et al*.'s implementation did. (See [1] for references to some of these algorithms.)

Finally, any other applications of Theorem 1 would be of interest.

Acknowledgment. I would like to thank Phil Klein for his careful reading of the manuscript.

References

[1] B. Dixon, M. Rauch, and R. Tarjan, Verification and sensitivity analysis of minimum spanning trees in linear time, *SIAM J. Comput*., **21**(6) (1992), 1184–1192.

[2] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput*., **13** (1984), 338–355.

[3] D. Karger, P. N. Klein, and R. E. Tarjan, A randomized linear-time algorithm to find minimum spanning trees, *J. Assoc. Comput. Mach*., **42** (1995), 321–328.

[4] J. Komlós, Linear verification for spanning trees, *Combinatorica*, **5** (1985), 57–65.

[5] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput*., **17** (1988), 1253–1262.

[6] R. Tarjan, Applications of path compressions on balanced trees, *J. Assoc. Comput. Mach*., **26** (1979), 690–715.

[7] R. Tarjan, *Data Structures and Network Algorithms*, CBMS–NSF Regional Conference Series in Applied Mathematics, Vol. 44, SIAM, Philadelphia, PA, 1983, p. 73.