

Lowest Common Ancestor Queries

Michiel Smid*

March 4, 2002

1 Traversing a binary tree

In many algorithms that operate on trees, it is necessary to traverse all nodes of the tree. The order in which the nodes are visited during this traversal depends on the problem at hand. In this section, we introduce three different orderings. We will see in Section 2 how they can be used to solve some specific problems.

Let $T = (V, E)$ be a rooted tree. We assume for simplicity that T is binary. Furthermore, we assume that each internal node of T has pointers to its two children, and that each node (except the root) has a pointer to its parent.

The three traversals are defined recursively. If T contains only one node, then each of the three traversals is this node itself. Otherwise, the *postorder traversal* is the recursively defined postorder traversal of the left subtree of T 's root, followed by the recursively defined postorder traversal of the right subtree of T 's root, followed by the root itself. The *inorder traversal* is the recursively defined inorder traversal of the left subtree of T 's root, followed by the root, followed by the recursively defined inorder traversal of the right subtree of T 's root. Finally, the *preorder traversal* is the root, followed by the recursively defined preorder traversal of the left subtree of T 's root, followed by the recursively defined preorder traversal of the right subtree of T 's root.

It is easy to see that the amount of time needed by each of these three traversals is proportional to the number of nodes of T .

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6.
E-mail: michiel@scs.carleton.ca.

2 Lowest common ancestors

Let T be a rooted tree. As in the previous section, we assume for simplicity that the tree is binary. The *lowest common ancestor* of any two nodes u and v is the node w that is an ancestor of both u and v and that is furthest away from the root of T . Equivalently, it is the node in which the two paths from the root to nodes u and v diverge.

In this section, we show how to represent the tree T in such a way that lowest common ancestor queries can be answered efficiently. In such a query, we get two arbitrary nodes u and v of T , and have to compute their lowest common ancestor. Clearly, we may assume w.l.o.g. that $u \neq v$.

Let n denote the number of leaves of the tree T . For any node u of T , let $size(u)$ denote the number of leaves in the subtree of u , and let $\ell(u) := \lfloor \log size(u) \rfloor$. Observe that $\ell(u)$ is an integer in the range from zero to $\lfloor \log n \rfloor$. Moreover, on the path from any leaf to the root of T , the $\ell(u)$ -values form a non-decreasing sequence.

We will use the values $\ell(u)$ to partition the tree T into a collection of pairwise disjoint paths. For any node u of T , let P_u be the subgraph of T consisting of all nodes v for which (i) $\ell(v) = \ell(u)$, and (ii) $\ell(w) = \ell(u)$ for all nodes w on the path in T between u and v . It is not difficult to prove that P_u is a path.

We extend the tree T by storing with each node u the value of $size(u)$, and a pointer to the node $gpar(u)$ of P_u that is closest to the root. We call $gpar(u)$ the *group parent* of u . We denote the parent of node u by $par(u)$.

Let u be any node of T , and consider the sequence $u, gpar(u), par(gpar(u)), gpar(par(gpar(u))), par(gpar(par(gpar(u))))$, ... of nodes, which terminates at the root of T . This sequence consists of at most $2\lfloor \log n \rfloor + 1$ nodes. Hence, we can walk from any node to the root of T in $O(\log n)$ time. We can generalize this approach to solve lowest common ancestor queries. The algorithm, which we denote by $lca(u, v)$, is given in Figure 1.

It is easy to see that the running time of algorithm lca is $O(\log n)$. It remains to show how to preprocess the tree T . That is, we have to show how to compute the values $size(u)$ and $\ell(u)$, and the group parents $gpar(u)$.

We start with the computation of the values $size(u)$ and $\ell(u)$. The values $size(u)$ can be computed by traversing the tree T in postorder. From these, we can compute $\ell(u)$ as $\ell(u) = \lfloor \log size(u) \rfloor$. The disadvantage of this approach is that we need the floor and logarithm functions for this. The following observation implies that we can avoid these functions.

Observation 2.1 *Let u be an internal node of T , and let v and w be the two children of u . Assume that $\text{size}(v) \leq \text{size}(w)$.*

1. *If $\text{size}(v) + \text{size}(w) \geq 2^{\ell(w)+1}$, then $\ell(u) = \ell(w) + 1$.*
2. *If $\text{size}(v) + \text{size}(w) < 2^{\ell(w)+1}$, then $\ell(u) = \ell(w)$.*

Our algorithm traverses the tree T in postorder. This algorithm, which we denote by *compute_size_and_l_values*, is given in Figure 2. A call to *compute_size_and_l_values*(u) computes the values $\text{size}(v)$ and $\ell(v)$ of all nodes v that are in the subtree of u , and returns the value $2^{\ell(u)}$. Hence, we obtain the values $\text{size}(v)$ and $\ell(v)$ of all nodes v of T , by calling this algorithm with u being the root. The running time for this call is $O(n)$.

The algorithm that computes the group parents is based on a preorder traversal of the tree T . This algorithm, which we denote by *compute_group_parents*(u, x), takes as input two nodes u and x such that $\text{gpar}(u) = x$. It computes the group parents of all nodes in the subtree rooted at u . The algorithm is given in Figure 3. If we denote the root of T by r , then a call to *compute_group_parents*(r, r) computes the group parents of all nodes of the tree. The time for this call is $O(n)$.

We have described all algorithms for binary trees. Observe that a binary tree with n leaves has exactly $2n - 1$ nodes. We leave it to the reader to generalize the algorithms to arbitrary rooted trees. This gives the following result.

Theorem 2.2 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time, such that lowest common ancestor queries can be answered in $O(\log n)$ time.*

3 A faster algorithm for lowest common ancestor queries

The algorithm in Section 2 works in the algebraic decision-tree model. In this section, we add the *indirect-addressing* operation as a unit-time operation to this model. We will show that in this more powerful model, lowest common ancestor queries can be answered in $O(1)$ time, after an $O(n)$ -time

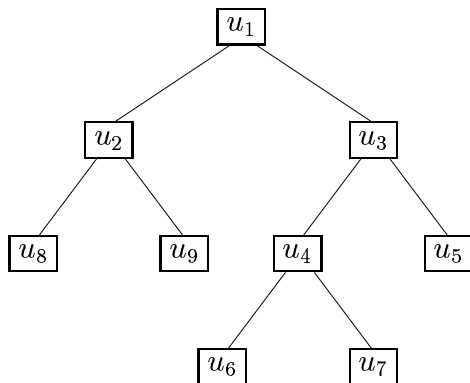
preprocessing step. (This algorithm can in fact be implemented in the algebraic decision-tree model such that lowest common ancestor queries can be answered in $O(\log \log n)$ time.)

We start by reducing the lowest common ancestor problem on a tree with n nodes to the so-called range minimum problem on an array of $O(n)$ real numbers. In Section 3.2, we show how range minimum queries can be answered in $O(1)$ time, after an $O(n)$ -time preprocessing of the array.

3.1 Reduction to the range minimum problem

Let T be a rooted tree with n nodes. We again assume for simplicity that T is a binary tree. The *level* of any node u of T is the number of edges on the path from the root to u .

We number the nodes of T arbitrarily as u_1, u_2, \dots, u_n . Consider an *Euler tour* of the “double-tree” obtained by doubling each edge of T . This tour starts at the root, visits each edge of T exactly twice, and returns to the root. Let $E[1 \dots 2n - 1]$ be the array that stores the nodes of T in the order in which they occur in the Euler tour. To give an example, consider the following tree T with root u_1 .



The corresponding array E is as follows. (The array indices are in the bottom row, whereas the array entries are in the top row.)

u_1	u_2	u_8	u_2	u_9	u_2	u_1	u_3	u_4	u_6	u_4	u_7	u_4	u_3	u_5	u_3	u_1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Observe that each internal node occurs exactly three times in E , whereas each leaf occurs exactly once. Let $L[1 \dots 2n - 1]$ be the array, where $L[i]$ stores the level of node $E[i]$, $1 \leq i \leq 2n - 1$. For our example tree, we obtain the following array L .

0	1	2	1	2	1	0	1	2	3	2	3	2	1	2	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Finally, let $R[1 \dots n]$ be the array in which $R[i]$ is equal to the smallest integer ℓ for which $E[\ell] = u_i$, $1 \leq i \leq n$. In other words, $R[i]$ is equal to the first “time” that node u_i is visited during the Euler tour of T . For our example tree, we get the following array R .

1	2	8	9	15	10	12	3	5
1	2	3	4	5	6	7	8	9

The following lemma states how the arrays E , L and R can be used to obtain the lowest common ancestor of any two distinct nodes of the tree T .

Lemma 3.1 *Let i and j be distinct integers with $1 \leq i \leq n$ and $1 \leq j \leq n$, and assume that $R[i] < R[j]$. Let k be the integer such that $R[i] \leq k \leq R[j]$ and $L[k]$ is minimum. The lowest common ancestor of the nodes u_i and u_j is equal to $E[k]$.*

Proof. The lowest common ancestor of u_i and u_j is visited during the portion of the Euler tour that starts at the first visit to u_i and ends at the first visit to u_j . Among all nodes visited during this portion, it is that node that is closest to the root of T . This portion of the Euler tour is stored in $E[R[i] \dots R[j]]$, and the levels of the nodes of this portion are stored in $L[R[i] \dots R[j]]$. ■

Hence, we can answer lowest common ancestor queries, by answering a so-called *range minimum query* in the array L . In Section 3.2, we will show how such queries can be answered in $O(1)$ time, after an $O(n)$ -time preprocessing step, by using the following property of L .

Observation 3.2 *The array L satisfies the (± 1) -property, i.e., $L[i + 1]$ is equal to either $L[i] + 1$ or $L[i] - 1$, for each i , $1 \leq i \leq 2n - 2$.*

Given the tree T , the arrays E , L and R can be computed in $O(n)$ time. Moreover, the reduction given above can easily be extended to arbitrary trees. Therefore, we will obtain the following result.

Theorem 3.3 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time such that lowest common ancestor queries can be answered in $O(1)$ time. The algorithms work in the algebraic decision-tree model, extended with indirect-addressing.*

The $O(n)$ -time preprocessing algorithm of Section 3.2 can be implemented in the algebraic decision-tree model (i.e., without using indirect-addressing). In this weaker model, the query time becomes $O(\log \log n)$. This will imply the following result.

Theorem 3.4 *Let T be a rooted tree with n nodes. We can preprocess T in $O(n)$ time such that lowest common ancestor queries can be answered in $O(\log \log n)$ time. The algorithms work in the algebraic decision-tree model.*

3.2 Solving the range minimum problem

In the *range minimum problem*, we are given an array $A[1 \dots n]$ of real numbers. We want to preprocess A such that for any two given integers i and j with $1 \leq i < j \leq n$, we can efficiently compute an integer k , $i \leq k \leq j$, for which $A[k]$ is minimum.

In the application to the lowest common ancestor problem, we have to solve this problem for the array $L[1 \dots 2n - 1]$. Observe that for this application, we need the *index* of the minimal element rather than the minimal element itself.

We start by giving two simple solutions to this problem, both of which do not assume any restriction on the array A . These solutions use $O(n^2)$ and $O(n \log n)$ preprocessing time, respectively. Afterwards, we show how these solutions can be combined to obtain an $O(n)$ -time preprocessing algorithm for arrays that satisfy the (± 1) -property.

A first solution: In this solution to the range minimum problem, we store all $\binom{n}{2}$ possible answers. That is, for each i , $1 \leq i < n$, there is an array $X_i[i + 1 \dots n]$, where for each j , $i + 1 \leq j \leq n$, $X_i[j]$ is equal to the integer k , $i \leq k \leq j$, for which $A[k]$ is minimum. It is clear that these arrays can be used to answer a range minimum query in $O(1)$ time. Using the relations

$$X_i[i + 1] = \begin{cases} i & \text{if } A[i] < A[i + 1], \\ i + 1 & \text{otherwise,} \end{cases}$$

for $1 \leq i < n$, and

$$X_i[j] = \begin{cases} X_i[j-1] & \text{if } A[X_i[j-1]] < A[j], \\ j & \text{otherwise,} \end{cases}$$

for $1 \leq i < n-1$ and $i+2 \leq j \leq n$, all arrays X_i , $1 \leq i < n$, can be computed in $O(n^2)$ time.

A second solution: We improve upon the previous solution, by storing for each i , $1 \leq i < n$, the answer to only $O(\log n)$ different range minimum queries. To be more precise, for each i , $1 \leq i < n$, there is an array $Y_i[1 \dots \lfloor \log(n-i+1) \rfloor]$, where for each ℓ , $1 \leq \ell \leq \lfloor \log(n-i+1) \rfloor$, $Y_i[\ell]$ is equal to the integer k , $i \leq k \leq i+2^\ell-1$, for which $A[k]$ is minimum.

Let us see how these arrays can be used to answer a range minimum query. Let i and j be two integers with $1 \leq i < j \leq n$, and let $h := \lfloor \log(j-i) \rfloor$. We observe that

$$\{i, i+1, \dots, i+2^h-1\} \cup \{j-2^h+1, j-2^h+2, \dots, j\} = \{i, i+1, \dots, j\}.$$

Let $k := Y_i[h]$ and $k' := Y_{j-2^h+1}[h]$. If $A[k] < A[k']$, then $A[k]$ is a minimal element in the subarray $A[i \dots j]$ and, therefore, k is the answer to the query. Otherwise, k' is the answer. Hence, assuming that the value of h can be computed in $O(1)$ time, we can answer a range minimum query in $O(1)$ time.

If we initialize an array $Z[1 \dots n]$, where $Z[m] = \lfloor \log m \rfloor$, $1 \leq m \leq n$, then h is obtained by looking up the value $Z[j-i]$. This array can be computed in $O(n)$ time, without using the floor and logarithm functions.

The relations

$$Y_i[1] = \begin{cases} i & \text{if } A[i] < A[i+1], \\ i+1 & \text{otherwise,} \end{cases}$$

for $1 \leq i < n$, and

$$Y_i[\ell] = \begin{cases} Y_i[\ell-1] & \text{if } A[Y_i[\ell-1]] < A[Y_{i+2^{\ell-1}}[\ell-1]], \\ Y_{i+2^{\ell-1}}[\ell-1] & \text{otherwise,} \end{cases}$$

for $1 \leq i < n-1$ and $2 \leq \ell \leq \lfloor \log(n-i+1) \rfloor$, imply that all arrays Y_i , $1 \leq i < n$, can be computed in $O(n \log n)$ time.

Hence, we have shown that the array A can be preprocessed in $O(n \log n)$ time, such that range minimum queries can be answered in $O(1)$ time.

An optimal solution: We now assume that the array $A[1 \dots n]$ satisfies the (± 1) -property. We assume for simplicity that $\log n$ is an even integer. If n is not a multiple of $(1/2) \log n$, then we add (less than $(1/2) \log n$) elements to the end of A such that the resulting array still satisfies the (± 1) -property and has a length that is a multiple of $(1/2) \log n$. (Hence, the new array has length $(1/2) \log n \lceil 2n / \log n \rceil$.)

We partition A into *blocks* of length $(1/2) \log n$, where the h -th block, for $1 \leq h \leq \lceil 2n / \log n \rceil$, is equal to

$$A[((h-1)/2) \log n + 1, ((h-1)/2) \log n + 2, \dots, (h/2) \log n].$$

Let $B[1 \dots \lceil 2n / \log n \rceil]$ be the array, where $B[h]$ is equal to the minimal element of the h -th block, $1 \leq h \leq \lceil 2n / \log n \rceil$. The array B can be computed in $O(n)$ time. We use our second solution to preprocess B for range minimum queries. This takes $O(n)$ time, after which queries in B can be answered in $O(1)$ time.

We also initialize an array $C[1 \dots n]$, where $C[i] = \lfloor 2i / \log n \rfloor$, $1 \leq i \leq n$. We will use this array to compute, given any i , $1 \leq i \leq n$, the block that contains $A[i]$. The array C can be computed in $O(n)$ time, without using the floor function.

Let i and j be two integers with $1 \leq i < j \leq n$, let $h := \lceil 2i / \log n \rceil$, and let $h' := \lceil 2j / \log n \rceil$. Hence, $A[i]$ and $A[j]$ are contained in the h -th and h' -th blocks of A , respectively. Let us first consider the case when $h < h'$. The minimal value in the subarray $A[i \dots j]$ is equal to the minimum of the following three numbers.

1. The minimum in the portion of the h -th block that starts at position i and ends at the end of this block.
2. The minimum in the portion of the h' -th block that starts at the beginning of this block and ends at position j .
3. The minimum in the subarray $B[h+1 \dots h'-1]$. (Let us say that this minimum is ∞ if $h' = h+1$.)

The third minimum can be computed in $O(1)$ time using the data structure for the array B . In order to compute the first and second minima, we have to preprocess the blocks so that *in-block* range minimum queries can be answered. If $h = h'$, then $A[i]$ and $A[j]$ are contained in the same block, and the query on A reduces to an in-block query.

Let us see how the blocks can be preprocessed for in-block queries. Observe that there are $\lceil 2n/\log n \rceil$ blocks, each having length $(1/2)\log n$. We claim, however, that the number of “distinct” blocks is only $O(\sqrt{n})$. To see this, let us replace for each h , $1 \leq h \leq \lceil 2n/\log n \rceil$, the h -th block by a *normalized* block having the same length and that is obtained by subtracting $A[\lceil (h-1)/2 \rceil \log n + 1]$ from each element of the h -th block. This gives $\lceil 2n/\log n \rceil$ normalized blocks, each one

- having length $(1/2)\log n$,
- starting with a zero, and
- satisfying the (± 1) -property.

We observe that such a normalized block can be uniquely encoded by a string of length $(1/2)\log n - 1$ over the alphabet $\{+1, -1\}$. Since the number of such strings is only $2^{(1/2)\log n - 1} = (1/2)\sqrt{n}$, it follows that we only have to preprocess up to $(1/2)\sqrt{n}$ many “distinct” blocks.

Our preprocessing proceeds as follows. For each h , $1 \leq h \leq \lceil 2n/\log n \rceil$, let $s^h := s_1 s_2 \dots s_{(1/2)\log n - 1}$ be the (± 1) -string corresponding to the h -th block of A , and let the integer x^h be defined by

$$x^h := \sum_{i=1}^{(1/2)\log n - 1} \frac{s_i + 1}{2} \cdot 2^i.$$

That is, we replace in s^h each occurrence of -1 by 0 , and each occurrence of $+1$ by 1 . The resulting binary string is the binary representation of the integer x^h . Each of the $\lceil 2n/\log n \rceil$ integers x^h can be computed in $O(\log n)$ time. Also, we can sort all these integers in $O(n)$ time. After this sorting step, we know the “distinct” blocks of A . For each value of x^h , we preprocess the corresponding normalized block for range minimum queries using our first solution. Since each normalized block has length $O(\log n)$, and since we do this for only $O(\sqrt{n})$ blocks, this part of the preprocessing takes $O(\sqrt{n} \log^2 n) = O(n)$ time.

Finally, for each h , $1 \leq h \leq \lceil 2n/\log n \rceil$, we store with the h -th block of A a pointer to the data structure for the normalized block that corresponds to the integer x^h .

We summarize the query algorithm. Given two integers i and j with $1 \leq i < j \leq n$, we compute $h := \lceil 2i/\log n \rceil$ and $h' := \lceil 2j/\log n \rceil$. If $h < h'$, then we do the following.

1. We follow the pointer to the data structure for the normalized block that corresponds to x^h , and compute the index of the minimal element in the portion that starts at position i and ends at the end of this normalized block.
2. We follow the pointer to the data structure for the normalized block that corresponds to $x^{h'}$, and compute the index of the minimal element in the portion that starts at the beginning of this normalized block and ends at position j .
3. If $h < h' - 1$, then we use the data structure for B to compute the index of the minimal element in the subarray $B[h + 1 \dots h' - 1]$.

It should be clear that, given this information, the index of the minimal element in the subarray $A[i \dots j]$ can be computed in $O(1)$ time. If $h = h'$, then we follow the pointer to the data structure for the normalized block that corresponds to x^h , and answer the query within this normalized block. The answer to this query allows us to find, in $O(1)$ time, the index of the minimal element in the subarray $A[i \dots j]$. We have proved the following result.

Theorem 3.5 *Let $A[1 \dots n]$ be an array of real numbers satisfying the (± 1) -property. We can preprocess A in $O(n)$ time such that range minimum queries can be answered in $O(1)$ time. The algorithms work in the algebraic decision-tree model, extended with indirect-addressing.*

In the algorithms given above, indirect-addressing is a crucial operation. We claim that these algorithms can be implemented without using indirect-addressing, at the expense of an increase in the query time:

Theorem 3.6 *Let $A[1 \dots n]$ be an array of real numbers satisfying the (± 1) -property. We can preprocess A in $O(n)$ time such that range minimum queries can be answered in $O(\log \log n)$ time. The algorithms work in the algebraic decision-tree model.*

```

Algorithm  $lca(u, v)$ 
(*  $u$  and  $v$  are nodes of the binary tree  $T$ ,  $u \neq v$  *)
 $g_{-1}^u := u$ ;  $g_0^u := gpar(u)$ ;  $i := 0$ ;  $p_0^u := u$ ;
while  $g_i^u \neq \text{root of } T$ 
do  $p_i^u := par(g_i^u)$ ;  $g_{i+1}^u := gpar(p_i^u)$ ;  $i := i + 1$ 
endwhile;
 $g_1^v := v$ ;  $g_0^v := gpar(v)$ ;  $j := 0$ ;  $p_0^v := v$ ;
while  $g_j^v \neq \text{root of } T$ 
do  $p_j^v := par(g_j^v)$ ;  $g_{j+1}^v := gpar(p_j^v)$ ;  $j := j + 1$ 
endwhile;
(* at this moment,  $g_i^u = g_j^v$  *)
while  $g_i^u = g_j^v$ 
do  $i := i - 1$ ;  $j := j - 1$ 
endwhile;
if  $i = -1$  and  $j = -1$ 
then if  $size(u) \geq size(v)$ 
then return  $u$ 
else return  $v$ 
endif
else if  $i = -1$ 
then if  $size(u) \geq size(par(g_j^v))$ 
then return  $u$ 
else return  $par(g_j^v)$ 
endif
else if  $j = -1$ 
then if  $size(v) \geq size(par(g_i^u))$ 
then return  $v$ 
else return  $par(g_i^u)$ 
endif
else if  $size(par(g_i^u)) \geq size(par(g_j^v))$ 
then return  $par(g_i^u)$ 
else return  $par(g_j^v)$ 
endif
endif
endif

```

Figure 1: The algorithm for solving lowest common ancestor queries.

```

Algorithm compute_size_and_l_values( $u$ )
(*  $u$  is a node of the binary tree  $T$  *)
if  $u$  is a leaf
then  $size(u) := 1$ ;
       $\ell(u) := 0$ ;
       $x := 1$ ;
      return  $x$ 
else  $v :=$  left child of  $u$ ;
       $w :=$  right child of  $u$ ;
       $x :=$  compute_size_and_l_values( $v$ );
       $y :=$  compute_size_and_l_values( $w$ );
      (*  $x = 2^{\ell(v)}$  and  $y = 2^{\ell(w)}$  *)
       $size(u) := size(v) + size(w)$ ;
      if  $size(v) \leq size(w)$ 
      then if  $size(u) \geq 2y$ 
        then  $\ell(u) := \ell(w) + 1$ ;
               $z := 2y$ ;
              return  $z$ 
        else  $\ell(u) := \ell(w)$ ;
              return  $y$ 
      endif
      else if  $size(u) \geq 2x$ 
        then  $\ell(u) := \ell(v) + 1$ ;
               $z := 2x$ ;
              return  $z$ 
        else  $\ell(u) := \ell(v)$ ;
              return  $x$ 
      endif
      endif
endif

```

Figure 2: Computing the values $size(v)$ and $\ell(v)$ for all nodes in the subtree rooted at node u .

```

Algorithm compute_group_parents( $u, x$ )
(*  $u$  and  $x$  are nodes of the binary tree  $T$ ,  $gpar(u) = x$  *)
 $gpar(u) := x$ ;
if  $u$  is not a leaf
then  $v :=$  left child of  $u$ ;
       $w :=$  right child of  $u$ ;
      if  $\ell(u) = \ell(v)$ 
      then compute_group_parents( $v, x$ )
      else compute_group_parents( $v, v$ )
      endif;
      if  $\ell(u) = \ell(w)$ 
      then compute_group_parents( $w, x$ )
      else compute_group_parents( $w, w$ )
      endif
endif

```

Figure 3: *Computing the group parents for all nodes in the subtree rooted at u .*