数理解析研究所講究録
436巻 1981年 233-260

Lazy Representation for A Proof of List Marking Algorithms

Satoru KAWAI

Department of Information Science
Faculty of Science
University of Tokyo
Tokyo 113
Japan

Keywords

Schorr-Waite algorithm, spanning tree,
program transformation

1. Introduction

Verifying algorithms which dynamically update data structures is a hard problem because of the difficulty of finding out invariants for the conventional inductive assertions. As a fairly simple instance of such algorithms, the Schorr-Waite graph marking algorithm is widely known [5]. This algorithm traverses a part of directed graph and marks all nodes reachable from a fixed root-node without using a stack for keeping track in the traversing. Instead of the stack, the algorithm uses the chain of nodes of the graph being traversed, altering the structure of the graph itself.

Among many authors who presented the correctness proofs of the Schorr-Waite algorithm, Gries [2] and Topor [6] attacked this problem with clear and intelligible formalisms giving convincing proofs, though their principles of proof are different to each other. Gries proved the algorithm mainly by reformulating the program and data structure using the usual inductive assertions. Topor, on the other hand, first presented some properties of list structures, then proved the algorithm by the method of intermittent assertion.

We present here a kind of proof of the correctness of the algorithm with a different approach in which algorithms with static data structure are first presented, later to be transformed into one with dynamic data. The key of our approach is that proofs should be made with simple and well-known concepts such as trees if the algorithms to be proved essentially work on

them. The mapping from simple data used for the proof to the possibly complex data, together with the corresponding transformation on algorithms, are then obtained.

2. Graph-Marking

We consider a directed graph on which the problem of marking is defined in terms of graph theory and sequential algorithms. As the basis of discussion, let G be a directed graph defined as follows.

(i) The set $V(G)$ of <u>nodes</u> of G is $\{z_1, z_2, \ldots, z_n\}$.

(ii) The set $A(G)$ of <u>directed lines</u> of G is the union of two sets $L(G)$ and $R(G)$ whose members are of the form (z_i, z_j) $(z_i, z_j \in V(G))$ and labeled by ℓ and r , respectively. For each node z in $V(G)$, it is required that there exist one and only one line of the form $(z, z') \in L(G)$ and one and only one line of the the form $(z, z'') \in R(G)$.

It is seen from the above definition that the out-degree of any node in G is 2 and the two outgoing lines are labelled differently. In what follows, we will use the notation $V(X)$ and $A(X)$ to denote the point set and the line set of a graphical object X , respectively. Next, we define the reachability in conventional fashion as follows.

For two nodes z_i and z_j in $V(G)$, we will write

$$z_i \overset{\ell}{\to} z_j \quad \text{if} \quad (z_i, z_j) \in L(G) ,$$
$$z_i \overset{r}{\to} z_j \quad \text{if} \quad (z_i, z_j) \in R(G) ,$$
$$z_i \to z_j \quad \text{if} \quad z_i \overset{\ell}{\to} z_j \text{ or } z_i \overset{r}{\to} z_j \text{ or both.}$$

A <u>path</u> s is a sequence of nodes $(u_0, u_1, \ldots, u_m)$ with the properties,

(i) $m \geq 0$,

(ii) $u_k \rightarrow u_{k+1}$ for $k=0, 1, \ldots, m-1$, and

(iii) $u_k \neq u_k$, if and only if $k \neq k'$.

If there is a path from z_i to z_j $(z_i, z_j \in V(G))$, we will write,

$$z_i \overset{*}{\rightarrow} z_j$$

which means that z_j is 'reachable from' z_i in G .

Let H be the maximal subgraph of G , whose set of nodes $V(H)$ is defined as $\{z_i | z_i \in V(G) , z_1 \overset{*}{\rightarrow} z_i\}$. The task of marking in this paper is, as in all other works on the list marking, to mark in some fashion all nodes in $V(H)$.

It is worthwhile to note that the graph defined here is slightly different from that of Lisp which includes another kind of nodes, atom, with no out-going lines. The graph of Lisp, however, can be simulated with our graph by introducing a special node "atommark" which is marked from the beginning. The two lines in an atom node are both made point to atommark. All algorithms in this paper behave at any atom node thus defined as if there were no out-going lines from it.

3. Spanning Tree

A sequential algorithm which performs the task of marking would traverse all the nodes in $V(H)$ in some order, tracing some spanning subgraph of H. We consider here a subgraph with a special property, called the L-spanning tree of H.

A spanning subgraph T_0 of H is called the <u>L-spanning tree</u> with respect to z_1 if the following conditions are satisfied.

(i) T_0 is a directed tree with z_1 being its root.

(ii) Let $s_0=(u_0=z_1, u_1, \ldots, u_p=z_k)$ and $s_1=(v_0=z_1, v_1, \ldots, v_q=z_k)$ be the paths from z_1 to z_k in T_0 and in H, respectively. Then, for any $i<p$ and $j<q$ such that $u_i=v_j$, the following condition is satisfied.

$$u_i \overset{\ell}{\rightarrow} u_{i+1}$$

$$\text{or} \quad u_i \overset{r}{\rightarrow} u_{i+1} \quad \text{and} \quad v_j \overset{r}{\rightarrow} v_{j+1}$$

That is, the unique path in T_0 from z_1 to some $z_k \in V(T_0)$ is the 'leftmost path' among all the paths from z_1 to z_k in H. The target algorithm of this paper essentially traverses T_0 though the target structure is dynamically modified.

Finally, let T be another directed graph defined as follows.

(i) $V(T) = V(T_0) = V(H)$

(ii) $A(T) = A(T_0) \cup P(T_0)$

where $P(T_0)$ is the set of directed lines labeled by P, and defined as

$$P(T_0) = \{(z_i, z_j) \mid (z_j, z_i) \in A(T_0)\}.$$

We will write, as before,

$$z_i \overset{p}{\prec} z_j \quad \text{if} \quad (z_i, z_j) \in P(T_0).$$

T is a tree in which every node except for the root is given a 'back pointer' to its parent, besides those to its left and right sons (c.f. Fig. 1).

4. Basic Marking Algorithm

In order to construct marking algorithms, let the original graph G be represented by the four integer arrays m , ℓ , r , p of size n+1 , indexed from 0 , as follows.

(i) α[i]=j if and only if $z_i \overset{\alpha}{\to} z_j$, where α=ℓ , r , or p .

(ii) m[i]=0 for all i at the beginning of the algorithms. The purpose of list marking is to set m[j]=1 for all j such that $z_j \in V(H)$ and to keep other m's unchanged.

(iii) ℓ[0]=r[0]=1 , p[1]=p[0]=0 . This setting is equivalent to introducing a new node z_0 into V(G) , the line (z_0, z_1) into both L(G) and R(G) , and the lines (z_1, z_0) and (z_0, z_0) into $P(T_0)$. This introduction is for simplification of the marking algorithms.

In what follows, algorithms are written in ALGOL68, with slight modification where needed.

Algorithm A0

```
[0:n]int m,ℓ,r,p;
#m,ℓ,r,p  are appropriately initialized#
int q:=1;
while q≠0
do m[q]:=1;
     if   m[ℓ[q]]=0  then  q:=ℓ[q]
     elif m[r[q]]=0  then  q:=r[q]
                     else  q:=p[q]
     fi
od
```

By defining that a node z_i is being *visited* when $q=i$, this algorithm can be considered to traverse the graph G in some way. The traversing of a line (z_i, z_j) is similarly defined.

Lemma 1. Algorithm A0 traverses only the lines in $A(T)$.

Proof. Suppose that the lemma is not true. Let (z_x, z_y) be the first non $A(T)$ line traversed by A0 . Note that either $z_x \overset{\ell}{\to} z_y$ or $z_x \overset{r}{\to} z_y$ because a p-line is included only in $A(T)$. Consider the paths

$$s_x=(z_1=u_0, u_1, \ldots, u_p=z_x)$$

and $s_y=(z_1=v_0, v_1, \ldots, v_q=z_y)$

in T_0 from z_1 to z_x and z_y , respectively.

Case 1. If z_y is on s_x , $m[y]$ has been set to 1 before (z_x, z_y) is traversed because s_x is the unique path from z_1 to z_x . Then it is seen from the algorithm

that $z_x \overset{p}{\to} z_y$.

Case 2. When z_y is not on s_x , let z be the common point between s_x and s_y , which is the farthest from z_1 . Let z be $u_k=v_k$ $(0 \le k \le p, q-1)$ and u_{p+1} be z_y . Since both s_y and $(u_0, u_1, \ldots, u_p, u_{p+1}=z_y)$ are paths from z_1 to z_y in H , it is seen from the definition of L-spanning tree that

$$z \overset{\ell}{\to} v_{k+1} \quad \text{and} \quad z \overset{r}{\to} u_{k+1} .$$

It is seen from the algorithm that (z, u_{k+1}) can be traversed only after (z, v_{k+1}) is traversed. Then, since z_y is included in a subtree in T_0 whose root is v_{k+1} , m[y] has been set to 1 before (z_x, z_y) is traversed. Then $z_x \overset{p}{\to} z_y$.

Both cases lead to contradiction.

Lemma 2. If A0 traverses only the lines in A(T) , it visits all nodes in H and no other nodes in G , and always terminates.

It is easily proved from the general property of tree and from the fact that every line in A(T) can be traversed at most once.

Proposition 1. A0 performs the task of marking H and always terminates.

5. Reference restriction

Next, we consider the following algorithm which acts exactly the same as A0 <u>with respect to the way of traversing</u>.

<u>Algorithm A1</u>

```
[0:n]int m,ℓ,r,p,h;
#m,ℓ,r,p  are appropriately initialized#
for i from 0 to n do h[i]:=3 od;
int s,t;
int q:=1;
while q≠0
do m[q]:=1;
   if           h[q]:=0; s:=q; t:=ℓ[q]; h[q]:=1;
        m[t]=0 then q:=t
   elif         h[q]:=0; s:=q; t:=r[q]; h[q]:=2;
        m[t]=0 then q:=t
   else         h[q]:=0; s:=p[q]; t:=q; h[q]:=3;
        q:=s
   fi
od
```

A new array h is introduced in order to indicate which one of the three pointers from each node is <u>inhibited to be referenced</u>. The values 0, 1, 2, and 3 in h indicate that none, ℓ , r , and p , respectively, cannot be referenced. When a reference to a pointer is required, the 'h-lock' is released at first (h[q]:=0) . It is easy to show that the following three lemmas hold.

Lemma 3. No reference to $\ell[q]$, $r[q]$, and $p[q]$, is made in A1 when $h[q]=1,2$, and 3, respectively.

Lemma 4. During the execution of A1, it is always satisfied that $k=q$ if $h[k]=0$.

Lemma 5. Algorithm A1 traverses T_0 with the same node order as A0.

6. Marking Algorithm with dynamic data

Up to the previous section we have developed marking algorithms which uses the data structure with static pointers, and proved the correctness of the algorithms. Our next step is to prepare another data structure and to define a dynamic mapping from the original structure onto the new one. More exactly, we prepare new integer arrays a and b , and let the following conditions always be satisfied (Fig. 2).

For every node z_i in H ,

when h[i]=3, a[i]=ℓ[i] and b[i]=r[i] hold,

when h[i]=1, a[i]=p[i] and b[i]=r[i] hold,

when h[i]=2, a[i]=ℓ[i] and b[i]=p[i] hold,

when h[i]=0, the values of a[i] and b[i] are in some transient state from and to one of the above three states.

In what follows, we will use the simultaneous assignation of the form " a,b,c := d,e,f " in order to simplify the description of algorithms, though it is not permitted in ALGOL68.

Algorithm A2

```
[0:n]int m,ℓ,r,p,h,a,b;
# m,ℓ,r,p,h  are initialized as in A1 #
for  i  from 0 to  n  do  a[i],b[i]:=ℓ[i],r[i] od;
int s:=0, t:=1, q:=1;
while  q≠0
do  m[q]:=1;
    if   h[q]:=0; s,t,a[q],b[q] := q,ℓ[q],p[q],r[q];
         h[q]:=1; m[t]=0 then q:=t
    elif h[q]:=0; s,t,a[q],b[q] := q,r[q],ℓ[q],p[q];
         h[q]:=2; m[t]=0 then q:=t
    else h[q]:=0; s,t,a[q],b[q] := p[q],q,ℓ[q],r[q];
         h[q]:=3; q:=s
    fi
od
```

Lemma 6. At the start of the loop in A2, one of the following three conditions is satisfied.

C_1 = {s=p[q], t=q, a[q]=ℓ[q], b[q]=r[q]} when m[q]=0 ,

C_2 = {s=q, t=ℓ[q], a[q]=p[q], b[q]=r[q]} when m[q]=h[q]=1 ,

C_3 = {s=q, t=r[q], a[q]=ℓ[q], b[q]=r[q]} otherwise.

Proof If m[q]=0 at the start of the loop, the node z_q has not been visited before. Then a[q]=ℓ[q] and b[q]=r[q] are assured by the initialization. Moreover, both s=p[q] and t=q are satisfied

(i) at the very beginning of the loop by the initial setting of s , t and q , and

(ii) by the condition (m[t]=0) inspected in the previous

execution of the loop.

Thus C_1 is satisfied when m[q]=0 .

On the other hand, m[q]=h[q]=1 imply that the last execution of the loop with respect to z_q ended with the first branch of the if clause (h[q]=1) , and that the latest execution of the loop ended with the last branch (m[q]=1) . Thus it is seen that C_2 is satisfied in this case. The last case for C_3 is similarly treated. It is worth while to note that, in the above proof, we happily use the property of traversing binary trees.

Lemma 6 guarantees the important fact that, when returning to a node through a p-line, the condition satisfied at the end of the previous visit to the node is recovered. This fact is a base for the next algorithm. Another important fact is that, when C_i (i=1,2,3) is satisfied, only cyclic shifts of values among three of s , t , a[q] and b[q] are required in order to make C_j (j = i mod 3 + 1) be satisfied. For example, when C_1 is realized, C_2 is satisfies by the simultaneous assignment

t, a[q], b[q] := b[q], t, a[q].

Moreover, the value of q is contained either t (C_1) or s (C_2 and C_3) . Then it is seen that all information for Algorithm A2 to work correctly can be obtained from the two arrays a and b , together with s and t . Our next algorithm uses only a , b , s , and t , omitting the accesses to ℓ , r , p , and q .

Algorithm A3

```
[0:n]int m,ℓ,r,p,h,a,b;
#arrays are initialized as in  A2#
int s:=0, t:=1;  #q  is omitted from here#
while t≠0
do if m[t]=0
     then  #C1  is satisfied#
       m[t]:=1;
       if    P1; m[t]=0 then skip
       elif  P2; m[t]=0 then skip
                         else  P3
       fi
   elif h[s]=1
     then  #C2  is satisfied#
       if    P2; m[t]=0 then skip
                        else  P3
       fi
     else  #C3  is satisfied#
             P3
   fi
od
where
     P1 = {h[t]:=0; a[t],t,s := s,a[t],t; h[s]:=1},
     P2 = {h[s]:=0; a[s],b[s],t := t,a[s],b[s]; h[s]:=2},
     P3 = {h[s]:=0; b[s],s,t := t,b[s],s; h[s]:=3};
```

The construct skip is equivalent to the null statement which appears in A3 in place of the now unnecessary statement "q:=t" .

250

In algorithm A3, the values of m and h are used in order to know how many times the node has been visited. The case selection by these values is necessary for adjusting conditions, C_1, C_2 or C_3.

7. The target algorithm

Algorithm A3 is the result of a direct transformation applied to A2 depending on Lemma 6. We now simplify and fold A3 in order to obtain the target algorithm.

First, we simplify the range of the values in h to {0, 1} because they are inspected only in the form

```
    if h[s]=1 then... .
```

Thus the value 0 is used in place of 2 and 3 for the assigments in P_2 and P_3 of the algorithm A3.

Next, consider the following two pieces of code.

```
B1: {if m[t]=0
       then m[t]:=1; P1;
            if m[t]=0 then skip
                      else S23 fi
       elif h[s]=1
         then S23
         else P3
       fi}
B2: {if m[t]=0
       then m[t]:=1; P1
      elif h[s]=1
       then S23
       else P3
      fi}
```

where

S_{23} = {*if* P_2; m[t]=0 *then* *skip*

else P_3

fi}

Note that B_1 is the loop body of A_3. B_1 and B_2 can be considered equivalent as the loop body of A_3 if, when they are applied to the data structures of the same status, the sequences of execution of the components, "m[t]:=1" , P_1 , S_{23} , and P_3 are identical. This equivalence is seen by checking a "piecewise equivalence" as follows.

Case 1. m[t]≠0. In this case, it is seen that the two identical *elif* brances are executed by a single application of B_1 or B_2 .

Case 2. m[t]=0 and m[a[t]]≠0 . A single application of B_1 in this situation executes a component sequence

"m[t]:=1", P_1 , S_{23}

in this order. On the other hand, the same sequence is executed by applying B_2 twice. By the first application of B_2

"m[t]:=1", P_1

is executed. The above P_1 realizes the situation m[t]=0 and h[s]=1 at the beginning of the second application, yielding the execution of S_{23} .

Case 3. m[t]=m[a[t]]=0. In this case, a single application of B_1 or B_2 executes

"m[t]:=1", P_1

yielding new situations of either Case 2 or Case 3.

This "folding" is also applicable with respect to S_{23} and P_3 , resulting the following loop body.

```
B3: {if   m[t]=0  then  m[t]:=1; P1
     elif h[s]=1  then  P2
                  else  P3
     fi}
```

The final algorithm A4 is shown below. This algorithm is obtained by replacing the loop body of A3 with B3. Algorithm A4 is an exact copy of the Schorr-Waite list marking algorithm without atomic cells.

Algorithm A4

```
[0:n]int m,h,a,b;
#arrays are initialized as in A3#
int s:=0, t:=1;
while t≠0
do if m[t]=0
      then m[t]:=1;
           a[t],t,s := s,a[t],t;
           h[s]:=1
   elif h[s]=1
      then a[s],b[s],t := t,a[s],b[s];
           h[s]:=0
      else b[s],s,t := t,b[s],s
   fi
od
```

8. Discussion

As shown above, our proof consists of two parts. In the first part, a straightforward data structure (bi-directional tree) is introduced, on which simple algorithms A0 and A1 work. It is easy to prove that they do the task of marking with respect to the given data structure, owing to the well-known simple properties of the structure of bi-directional tree. The second part of the proof is for a dynamic data structure which is first manipulated through the pure side-effect of the revised algorithm A2, thus giving no influence on the correctness of the marking algorithm. This data structure, together with the two auxiliary variables, is then shown to have full necessary information for the algorithm to work correctly. Algorithm A3 relys on this point and works solely on the dynamic data structure. At this stage of the proof, the original static data structure (for A0) is transformed into the dynamic one (for A3). The techniques of program transformations are used in order to obtain the target algorithm A4. That is the story of our proof.

As stated before, Gries gave a proof of the Schorr-Waite algorithm using inductive assertions [2, 3]. Though he used a slightly modified formulation of the original algorithm in order to make his proof convincing and easy to follow, it is believed to be the first proof with convincingly detailed reasoning using inductive assertions with a termination proof. On the other hand, Topor [6] presented an interesting proof of the algorithm, in which a series of facts (propositions) on the properties of

(partially) marked list structure is first introduced and proved. The facts are stated in terms of the status of the whole data structure. Then the correctness of the algorithm is proved by the method of intermittent assertions [4].

It is not our intention to revive the controversy about the superiority between the two methods; inductive and intermittent assertions. What we want to say is that investigation of the behaviour of the algorithm to be proved should be made carefully in order to present convincing proofs. For example, both the above authors presented stack-based marking algorithms at the beginning of their papers, stating that in the Schorr-Waite algorithm the graph itself does the job of stack. It is true that this is an aspect of the algorithm and many other authors have tried to give the proof by transforming stack-based or recursive algorithms into the target algorithm [1]. A father inspection of the algorithm, however, would reveal slight but non-trivial difference between the stack mechanism and the use of graph nodes in the Schorr-Waite algorithm. During the course of execution, the stack contains only the minimum information for backtracking necesssary for the task of marking. A node is removed away from the stack immediately after its right pointer is traced. It is seen, therefore, that only a subset of nodes on the path from the root to the "current node" are retained in the stack. In the Schorr-Waite algorithm, on the other hand, all the nodes on the path are placed in the so-called "stack".

The above consideration made us to realize another aspect of the algorithm presented in this paper, which is based on bi-

directional tree and reference restriction. The analysis of the Schorr-Waite algorithm through the latter aspect did, we believe, help both the algorithm and the spaghetti of the data structure to be well structured, though a kind of multistep proof is required.

It is worthwhile to compare the two data structures manipulated by the first algorithm A0 and the target algorithm A4. It could easily be recognized that the final structure represents in some way a part of the first structure. The way of representation, however, is not static but dynamic. The essential point is when a full representation of a node is required in the course of algorithm application, that representation is realized by the use of auxiliary variables, s and t . From the analogy of lazy evaluation with respect to algorithms, the above scheme of data manipulation could be called the Lazy Representation of data structures. In lazy representation scheme, the full form of data structure of a part of whole data is constructed from other information when the full form is truely required. Our proof in this paper reveals the existence of this scheme in the Schorr-Waite algorithm. It would be helpful to introduce the concept of lazy representation which has been unconsciously confused to and mixed up in the term, lazy evaluation.

Acknowledgement

The author gratefully acknowledges the many helpful discussions with Dr. Teruo Hikita of Tokyo Metropolitan University, who pointed out the importance of program transformation in the proof method presented here.

References

1. Dershowitz, N.: The Schorr-Waite marking algorithm revisited, Information Processing Letters, 11, 141-143 (1980).

2. Gries, D.: The Schorr-Waite Graph Marking Algorithm, Acta Informatica, 11, 223-232 (1979).

3. Hoare, C.A.R.: An axiomatic basis for computer programming, CACM, 12, 576-580, 583 (1969).

4. Manna, Z. and Waldinger, R.: Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness, CACM, 21, 159-172 (1978).

5. Schorr, H. and Waite, W. H.: An efficient machine-independent procedure for garbage collection on various list structures, CACM, 10, 501-506 (1967).

6. Topor, R. W.: The Correctness of the Schorr-Waite List Marking Algorithm, Acta Informatica, 11, 211-221 (1979).

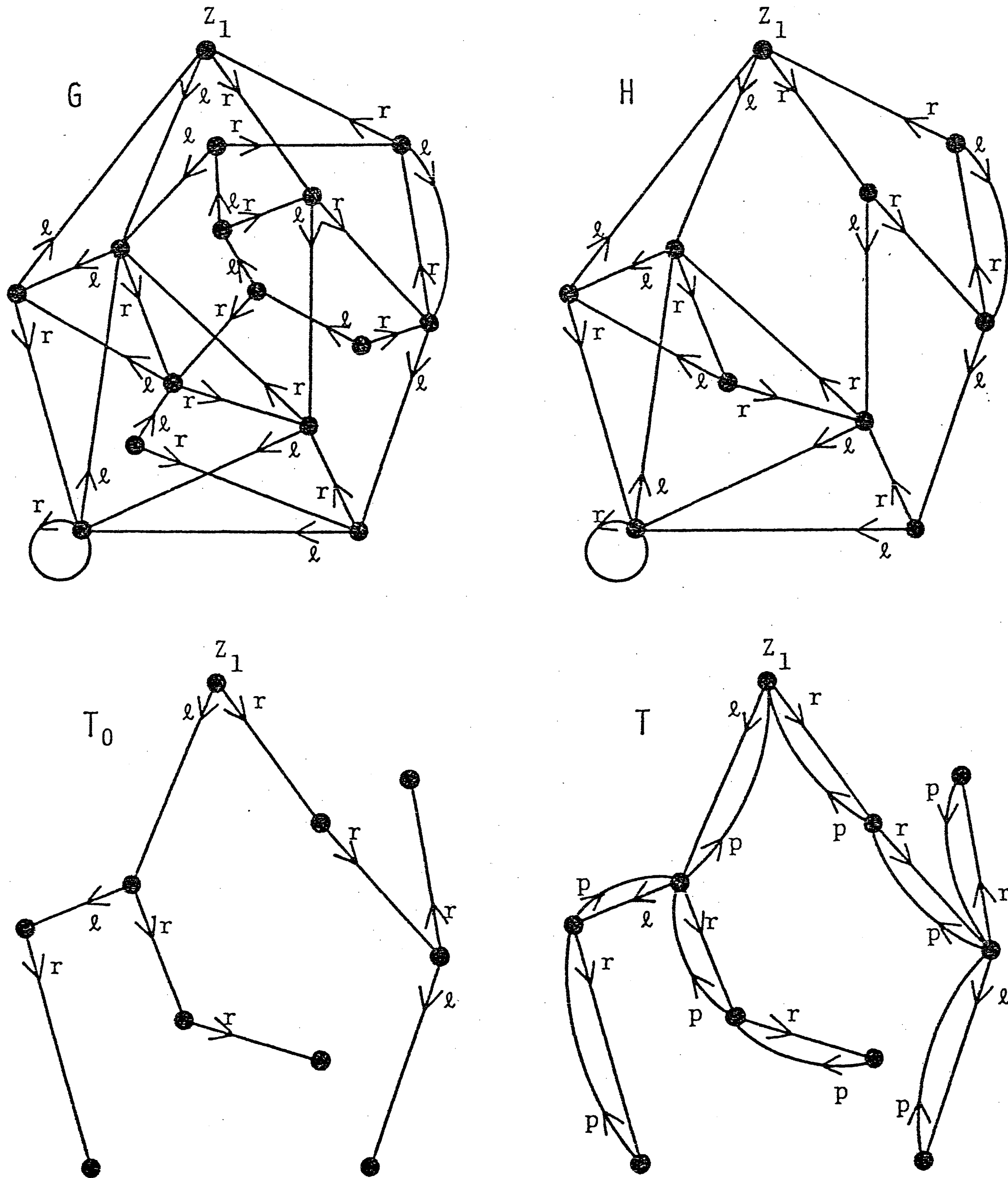

Figure 1. Graphs and trees in list marking algorithm.

G: original graph with a root z_1 and line labelling

H: maximal subgraph reachable from z_1

T_0: L-spanning tree of H

T: bi-directional L-spanning tree

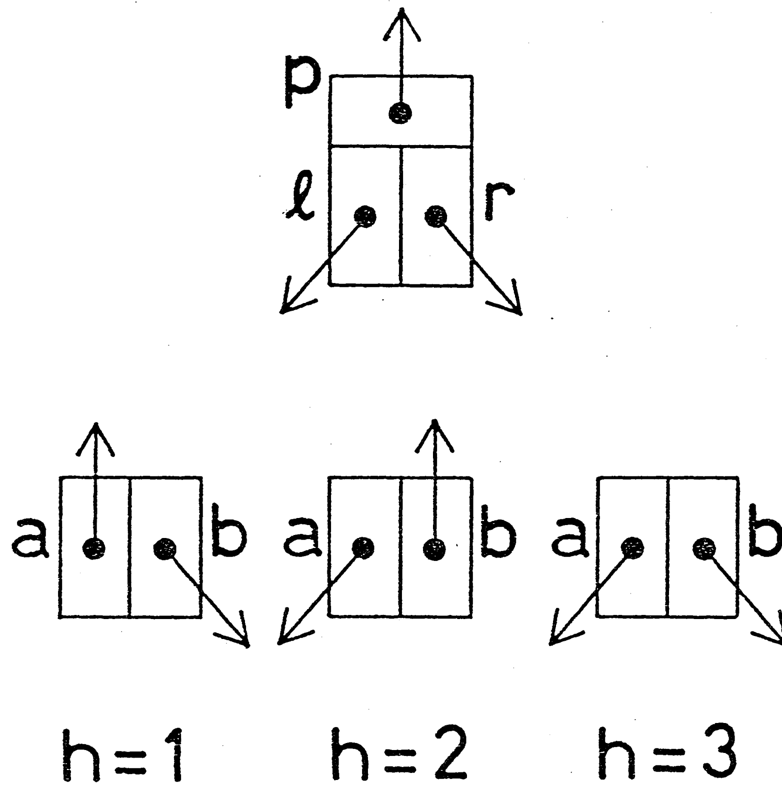

Figure 2. Relation of static (p, ℓ, r) and dynamic (a, b) data.