THE SCHORR–WAITE MARKING ALGORITHM REVISITED *

Nachum DERSHOWITZ
Department of Computer Science, University of Illinois, Urbana, IL 61801, U.S.A.

Received 21 December 1979; revised version received 16 May 1980

Program correctness, marking algorithms, program transformation, Schorr–Waite algorithm, stepwise refinement

1. Introduction

There has been a recent wave of interest [2,4,5,13] in proving the correctness of Schorr and Waite's [10] algorithm for making all the nodes in a directed graph that are accessible from a given node (see also [7]). In this paper we show how, by developing that algorithm in a systematic manner (along the lines of [8]) from the standard depth-first search paradigm (see, for example, [12]), a simple proof of correctness follows naturally.

2. First version

Given a finite directed (not necessarily strongly connected) graph G, the goal is to set an 'active' bit a(v) to **true** for those nodes v in G that are accessible from some given node p. It is assumed that initially a(v) is **false** for all $v \in G$. Each node v has a given out-degree d(v) with pointers $G_1(v), G_2(v), \ldots, G_{d(v)}(v)$ to neighboring nodes.

The following recursive program accomplishes that task by performing a depth-first search of the graph:

```
call Mark(p)
procedure Mark(v):
        if ¬a(v)
          then a(v) := true
               for c := 1 to d(v)
                 do call Mark(G_c(v))
                 od
        fi
```

The program marks the nodes accessible from p by first marking p and then recursively calling the marking program for each of the neighbors of p.

To prove that this recursive program is correct, four things must be shown:

(a) executing the program leaves the structure of the graph G unchanged;

(b) only nodes accessible from p are activated;

(c) the program terminates;

(d) upon termination, all nodes accessible from p are active.

That (a) G is unchanged is obvious from the fact that the program contains no assignment to pointers in G. By induction on the computation it can be seen that (b) Mark is only called for accessible nodes: The argument p of the initial call is accessible. Furthermore, assuming that Mark(v) is called for an accessible node v, then the calls $\text{Mark}(G_c(v))$ are to the like-wise accessible neighbors of v.

The termination property (c) is proved by structural induction ([1]) on the number of unactivated nodes in G: Since in executing Mark(v) one previously inactive node in G, viz. v, is activated before calling $\text{Mark}(G_c(v))$, the inductive hypothesis asserts that each such call to Mark for a neighbor of v will terminate. Since Mark(v) calls $\text{Mark}(G_c(v))$ only a finite number of times, Mark(v) must terminate as well.

Lastly, the following line of reasoning demonstrates that (d) all accessible nodes are activated:

* This research was partially supported by the National Science Foundation under grants MCS-77-22830 and MCS-79-04897.

Clearly, the program never deactivates a node. Thus, after any execution of Mark(v), v must be active. Now, since all nodes are initially inactive, if upon termination a(v) holds for any node v, then it must be on account of the assignment a(v) := **true**, after which the **for**-loop was executed and consequently $a(G_c(v))$ must also hold for all the neighbors $G_c(v)$, $1 \leqslant c \leqslant d(v)$, of v. In particular, a(p) holds upon termination of Mark(p); it follows (by induction on the length of an access path) that a(v) holds for all nodes v accessible from p.

2. Intermediate version

As is well known, the underlying idea of the Schorr–Waite algorithm is to stack the arguments of the recursive calls within the graph G itself. The value of the loop counter c for a call Mark(v) is stored in a field c(v) that can take on values from 1 to d(v) + 1. (These c(v)'s could be placed on a stack, or stored within the nodes v.) Let u point to the head of the stack of arguments; the continuation of the stack from a node v is stored in the $c(v)^{th}$ pointer from v.

We are led to the program:

```
(u, v) := (null, p)
call Mark
procedure Mark:
  if ¬a(v)
    then a(v) := true
         for c(v) := 1 to d(v)
           do (v, u, G_c(v)(v)) := (G_c(v)(v), v, u)
              call Mark
              (G_c(u)(u), v, u) := (v, u, G_c(u)(u))
           od
    fi
```

What were implicit assignments v := p and $v := G_c(v)$ in the calls of the first version are now explicit. The stack u is set initially to some unique 'null' pointer (actually the initial value of u does not affect the computation); before the recursive call, the node v is added to the top of the stack u and the c^{th} pointer from v, $G_{c(v)}(v)$, is made to point to what was the top of the stack.

The proof of the correctness of this program is the same as that of the previous version except that to prove that the structure is unchanged, that hypothesis is combined with termination in the inductive argument: One may assume that the recursive call to Mark returns with the values of u, v, and G as they were before the call. Also, the counter c(v) is unaffected by the recursive call, since Mark can never be called again with v inactive. Thus, one need only verify that the effect of the multiple assignment

$$(v, u, G_{c(v)}(v)) := (G_{c(v)}(v), v, u)$$

is undone by the subsequent assignment

$$(G_{c(u)}(u), v, u) := (v, u, G_{c(u)}(u)).$$

But that is easy to see, e.g. the original value of u was in fact stored in what was $G_{c(v)}(v)$ and is now $G_{c(u)}(u)$.

3. Final version

In this section we derive a somewhat more efficient marking algorithm from the previous version by a series of program transformations. Each transformation is 'correctness-preserving' in that the transformed program is guaranteed to be correct if the untransformed program was. (See, for example, [3] for methods of demonstrating this property.)

As a first step, note that in the event that Mark is called for an active node, that call is essentially vacuous and all the stacking before the call and unstacking after are superfluous. This suggests separating out that case by replacing the body of the **for**-loop with

```
t := G_c(v)(v)
if ¬a(t)
  then (v, u, G_c(v)(v)) := (t, v, u)
       call Mark
       (G_c(u)(u), v, u) := (v, u, G_c(u)(u))
  fi
```

With this change, Mark can never be called for an active node. Therefore, that test at the beginning of Mark may be eliminated. (Similarly, if the node has no outgoing edges, then the call does no more than set the active bit, and that case, too, could have been separated out.)

Finally, one may remove the recursion from the program and replace it with two **goto**'s in the standard manner. To detect the return from the outermost call, one need only test for a **null** stack. Thus,

our final marking algorithm is:

```
(u, v) := (null, p)
Mark:
  a(v) := true
  for c(v) := 1 to d(v)
   do t := G_c(v)(v)
      if ¬a(t)
        then (v, u, G_c(v)(v)) := (t, v, u)
             goto Mark
             cont'd:
             (G_c(u)(u), v, u) := (v, u, G_c(u)(u))
        fi
   od
if u ≠ null then goto cont'd fi
```

This 'final' version can be further modified according to one's particular needs and tastes. It can, for example, be adapted to the case where all nodes have outdegree zero or two (as in LISP) by unwinding the **for**-loop.

4. Conclusion

We have shown how the sophisticated Schorr–Waite marking algorithm may be developed from a straightforward depth-first marking algorithm by a short sequence of relatively simple modifications. Other (more or less detailed) derivations of this or related algorithms are pursued in [6,9,11]; some of those references go on to develop more involved algorithms.

We have also seen how the correctness of the last version follows without difficulty from the (virtually obvious) correctness of the original version. Developing the program in steps has made it easier, in our view, to understand and validate the final result.

Acknowledgment

I thank Ed Reingold for suggesting that a simpler proof of correctness ought to be possible.

References

[1] R.M. Burstall, Proving properties of programs by structural induction, Comput. J. 12 (8) (1969) 41–48.
[2] A.G. Duncan and L. Yelowitz, Studies in abstract/concrete mappings in proving algorithm correctness, Proc. Sixth Colloq. on Automata, Languages and Programming, Graz, Austria (1979) 218–229.
[3] S.L. Gerhart, Correctness-preserving program transformations, Proc. Second Symposium on Principles of Programming Languages, Palo Alto, CA (1975) 54–66.
[4] S.L. Gerhart, Marking algorithms, in: F.L. Bauer and M. Broy, Eds., Program Construction International Summer School (Springer, Berlin, 1979) 472–492.
[5] D. Gries, The Schorr–Waite graph marking algorithm, Acta Informat. 11 (3) (1979) 223–232.
[6] M. Griffiths, Development of the Schorr–Waite algorithm, in: F.L. Bauer and M. Broy, Eds., Program Construction International Summer School (Springer, Berlin, 1979) 464–471.
[7] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (Addison-Wesley, Reading, MA, 1968) Section 2.3.5.
[8] D.E. Knuth, Structured programming with go to statements, Comput. Surveys 6 (4) (1974) 261–301.
[9] S. Lee, S.L. Gerhart and W.P. deRoever, The evolution of list-copying algorithms, Proc. Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX (1979) 53–67.
[10] H. Schorr and W.M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Comm. ACM 10 (8) (1967) 501–506.
[11] S. Soule, A note on the nonrecursive traversal of binary trees, Comput. J. 20 (4) (1977) 350–352.
[12] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[13] R.W. Topor, The correctness of the Schorr–Waite list marking algorithm, Acta Informat. 11 (3) (1979) 211–221.