Information Processing Letters 22 (1986) 179–183
North-Holland

17 April 1986

TWO TAGLESS VARIATIONS ON THE DEUTSCH–SCHORR–WAITE ALGORITHM

Mark C. HAMBURG

University of Michigan, Ann Arbor, MI 48109, U.S.A.

Communicated by David Gries
Received 30 November 1984
Revised 5 June 1985

Keywords: Data management, list processing, data structures, marking, tags

1. Introduction

We describe two variations on the Deutsch–Schorr–Waite (henceforth, simply D–S–W) algorithm for marking finite, rooted, binary, directed graphs (henceforth simply referred to as graphs) which dispense with the tag bit while retaining the same overall structure. The first variation handles all graphs, just as D–S–W, but trades average $O(\log n)$ (worst case $O(n)$) workspace and $O(n)$ time for $O(1)$ workspace and average $O(n \log n)$ (worst case $O(n^2)$) time. The second variation achieves both $O(1)$ workspace and $O(n)$ time but at the expense of not being able to correctly handle cyclic graphs. The variations demonstrate the gains that can be made through careful analysis of loop invariants and, by keeping the overall structure of a well-known algorithm, represent a gain in clarity over previous tag-free marking algorithms [4]. We also present some arguments to support the conjecture that $O(1)$ workspace and worst case $O(n)$ time are not simultaneously achievable.

2. Comparison with previous work

Marking algorithms are extensively covered in several sources, for example, [1] and [2]. All these algorithms, however, require at least $O(n)$ (worst case) workspace or require the graph to be stored in a particular format in memory (e.g., all links must point downward in memory). The best known work on tagless marking was done by Lindstrom [1,2]. His first algorithm, while elegantly simple, suffers from its inability to handle cycles predictably and, because it does not recognize shared subtrees, has worst case $O(2^n)$ time. His second algorithm solves these problems and achieves the some workspace and time complexities as the first variation on D–S–W, which we present here, but suffers from a convoluted operation/presentation that makes comprehension difficult.

D–S–W and its link-reversal technique have already given rise to several variations [2]. In its original form, the algorithm requires one tag bit per node, i.e., $O(n)$ space. The tag bits can be replaced with a bit stack as deep as the deepest node in the graph, which reduces the workspace to average $O(\log n)$ but the worst case is still $O(n)$. Knuth proposes to get around the need for a tag bit by using the atom bit as a tag bit during marking, but an atom bit is not guaranteed to be part of the structure since atoms may be identified by the region they occupy in memory rather than by a bit in each cell.

Our first variation on D–S–W avoids both the clarity and the workspace problems and marks both cyclic and acyclic graphs but gives up $O(n)$ time. Our second variation on D–S–W regains $O(n)$ time but gives up the ability to mark cyclic graphs.

3. Analysis of the Deutsch–Schorr–Waite algorithm

In order to prepare for the discussion of our variations on D–S–W, in this section we present our version of D–S–W and prove it to be correct and linear.

We begin by describing our data structure. It consists of a collection of atoms and nodes and the special pointer value *vroot*. Let x be a pointer. If $x\hat{}$ is a node, then $x\hat{}.l$ and $x\hat{}.r$ are pointers to other nodes or atoms, and $x\hat{}.m1$ and $x\hat{}.m2$ are boolean mark fields (or a mark field and a tag field). Initially, $x\hat{}.l = L_x$ and $x\hat{}.r = R_x$. If $x\hat{}$ is an atom, then only the boolean mark field $x\hat{}.m$ exists. Furthermore, predicate *atom* ($x\hat{}$) has the value "$x\hat{}$ is an atom". Pointer value *nil* is assumed to point to an atom in order to avoid certain tests, but these tests can be added if this is not the case. Pointer value *vroot* points to a node with $vroot\hat{}.l$ pointing to the graph and $vroot\hat{}.r = vroot$. Finally, all the mark fields are assumed to be reset at the beginning of the algorithm.

Our version of D–S–W maintains two pointers *upper* and *lower*, which delimit a path in the original graph leading from $upper\hat{}$ to $lower\hat{}$. During operation, there is a path in the current graph from $upper\hat{}$ via left pointers to $vroot\hat{}$. We also use a switch *dir* with values *up* and *down* indicating the direction we are moving in the graph. Initially, $upper = vroot$, *lower* points to the root of the graph, and $dir = down$. Whenever we encounter a new node (i.e., an unmarked node), we set its *m1*-field, add it at the head of the path traced out from $upper\hat{}$, and go mark its left subgraph. Upon returning from the left subgraph, as identified by the *m1*-field being set but the *m2*-field still reset, we set the *m2*-field to indicate on return to this node that both subgraphs have been marked, save the pointer to the left subgraph in the *r*-field, and go mark the right subgraph. Upon returning from the right subgraph, we restore the pointers and ascend the path from $upper\hat{}$ in search of a node for which the right subgraph has not yet been marked. If we ascend all the way to $vroot\hat{}$, then we are done. Whenever we encounter an atom, we mark it and begin ascending. Whenever we encounter an old (previously marked) node during descent, we simply begin ascending.

The state of affairs created by the operation of D–S–W is captured in the following five-piece invariant:

(i) There is a path **P** of nodes in the current graph leading from $upper\hat{}$ to $vroot\hat{}$ via left pointers.

(ii) Every node $x\hat{}$ not on **P** has $x\hat{}.l = L_x$ and $x\hat{}.r = R_x$.

(iii) Every unmarked node reachable in the original graph is reachable in the current graph via a path of unmarked nodes starting with either $R_{x\hat{}}$ where $x\hat{}$ is a node on path **P** or with $lower\hat{}$ if $dir = down$.

(iv) If $upper\hat{}.m2$ is set, then $lower = R_{upper}$ and $upper\hat{}.r = L_{upper}$; otherwise, $lower = L_{upper}$ and $upper\hat{}.r = R_{upper}$.

(v) For all nodes $x\hat{}$ on path **P** except $vroot\hat{}$, *if* $x\hat{}.l\hat{}.m2$ is set, then $x = R_{x\hat{}.l}$ and $x\hat{}.l\hat{}.r = L_{x\hat{}.l}$; otherwise, $x = L_{x\hat{}.l}$ and $x\hat{}.l\hat{}.r = R_{x\hat{}.l}$.

D–S–W is then given as follows:

```
upper, lower, dir := vroot, vroot^.l, down;
do (upper < > vroot or dir < > down) →
   if descent-to-new-node →
      lower, lower^.l, upper, lower^.m1 :=
      lower^.l, upper, lower, true
   □ ascent-from-left-subgraph →
      lower, upper^.r, upper^.m2, dir :=
      upper^.r, lower, true,
   □ ascent-from-right-subgraph →
      lower, upper, upper^.l, upper^.r :=
      upper, upper^.l, upper^.r, lower
   □ descent-to-atom →
      lower^.m, dir := true, up
   □ descent-to-old-node → dir := up
   fi
od
```

where the tests in the guarded-command conditional statements are as follows (**cand** is a short-circuiting **and**):

```
descent-to-new-node = =
   dir = down and not atom (lower^) cand not
   lower^.m1
```

ascent-from-left-subgraph = =
  dir = *up* **and not** *upper*$^\wedge$.*m2*
ascent-from-right-subgraph = =
  dir = *up* **and** *upper*$^\wedge$.*m2*
descent-to-atom = =
  dir = *down* **and** *atom* (*lower*$^\wedge$)
descent-to-old-node = =
  dir = *down* **and not** *atom* (*lower*$^\wedge$) **cand**
  lower$^\wedge$.*m1*

To see that this loop is executed a linear number of times, we observe that success for the tests of the first two guarded-command conditional statements will result in a mark field being changed from ***false*** to ***true***, and hence each of these tests only succeeds n times. Success for the third test will result in one node being removed from path **P**. Since nodes are only added through the first guarded-command conditional statement, only n nodes will be added to path; this test, therefore, succeeds n times. The last two conditions only succeed when *dir* = *down* and change *dir* to *up*; therefore, *dir* must be reset to *down* between each success for either of these tests, i.e., the second test must have succeeded at some intervening time. Hence, together, these tests succeed $n + 1$ times. Thus, the total number of times any of the tests can succeed is $4n + 1$. The disjunction of the tests, however, is a tautology and, so, every time the loop is executed, one of them must succeed. The loop, hence, is executed at most $4n + 1$ times, and the algorithm has O(n) time.

Upon termination of the loop, *upper* = *vroot* and *dir* = *down*, so path **P** will consist only of *vroot*$^\wedge$, and by invariants (ii) and (iii), the data structure will be correctly marked (i.e., all nodes reachable in the original graph will be marked and all pointers will be returned to their original values).

4. Variations on the Deutsch–Schorr–Waite algorithm

Our first variation is the result of the observation that a sixth statement may be added to the above invariant:

(vi) For all nodes $y^\wedge$, $y^\wedge$.*m1* is set iff either $y^\wedge$.*m2* is set or $y^\wedge$ is on path **P**.

Using this new invariant, we see that we can remove all references to the *m1*-fields from the code for D–S–W. At termination, the *m2*-fields in all of the nodes reachable in the original graph will be set, so we can treat this field as the mark field rather than as the tag field as it is traditionally considered. (Hence, the choice of the names *m1* and *m2* rather than *mark* and *tag*.) In the first guarded-command conditional statement, we simply eliminate this field from the block set operation. In the first and fifth tests, we replace the test of *lower*$^\wedge$.*m1* with the disjunction (*lower*$^\wedge$.*m2* **or** *onpath* (*lower*)) where *onpath* (*lower*) is a predicate which scans path **P** searching for node *lower*$^\wedge$, returning ***true*** if it finds the node before reaching *vroot*$^\wedge$, and returning ***false*** otherwise. To avoid making the execution speed any worse than necessary, a short-circuiting **or** should be used. A further improvement in execution speed can be gained by keeping track of the shallowest node on path **P** for which the *m2*-field has not been set and stopping the search if this node is reached without finding node *lower*$^\wedge$. Even with these improvements, *onpath* (*lower*) takes time proportional to the depth of node *lower*$^\wedge$ in the original graph which averages O(log n) and worst case is n. Hence, this variation, while eliminating one of the mark fields (i.e., eliminating the tag), raises the time complexity from O(n) to average O(n log n) and worst case O(n^2).

Our next variation comes as a result of the observation that, by parts (iv) and (v) of the invariant, *onpath* (*lower*) will only succeed if a cycle exists in the original graph. Hence, our second variation simply eliminates the *onpath* tests from the preceding variation. This restores the algorithm to operating in O(n) time but at the cost of the additional precondition that the graph be known to be acyclic. Actually, this is not all that difficult to be sure of in a normal LISP-system. If *onpath* (*lower*) never succeeded during the marking phase of the previous garbage collection or the graph was known to be acyclic at that time and, since then, none of the primitive operations capable of forming cycles—and this set of operations is usually quite small (e.g., **rplaca, rplacd, rplacb,** and **nconc**)—have been used, then the graph must be acyclic.

Thus, analysis of loop invariants for D–S–W has enabled us to eliminate the tag bits from the algorithm. If we know the graph is acyclic, then this can be done without additional cost, but if cycles are possible in the graph, then linear time must be abandoned for worst-case quadratic time to eliminate the tag bits. This, unfortunately, makes the first variation only of academic interest.

5. A conjecture

One consolation is that it may be impossible to mark all cyclic structures using a bounded workspace while maintaining linear time complexity. We present the outline of an argument (not a proof) as to why this conjecture might be true. The argument is based on the following summary of the operation of any marking algorithm:

> The algorithm in some way maintains a number of sets of nodes. These sets are in a specific order, and each step in the algorithm involves modifying a node and/or moving it to a higher order set. Upon reaching the highest set, the pointers in the node are the same as they were initially, and the mark field in the node is set. During any given step, when a node is processed, one of its links may be followed to find a possibly new node. (The only node in the sets initially is *vroot*^.) If the node is not already in one of the sets, it is added to one of the sets. This process continues until all of the nodes are in the final set.

In order for the algorithm to operate in linear time, each of the steps above (getting a node from one of the sets other than the final set, moving it to a new set, testing possibly new nodes for membership in the sets, and adding definitely new nodes to the sets) must operate in average time independent of the number of nodes and of the graph structure. Hence, constant time may only be exceeded for a decreasing fraction of the operations on nodes as n becomes large. For example, Morris's tree-traversal algorithm contains two nested loops each of which may take up to n cycles to complete, but maintains $O(n)$ time because the inner-loop can execute at most $2n$ times [5]. Hence, the number of times the inner loop can take say $O(\log n)$ time to execute is $O(n/\log n)$. Thus, as n gets large, the fraction of the executions of the outer loop for which the inner loop takes greater than constant time gets small. The argument is based on indications that these tasks cannot all be achieved with constant average time if the workspace is restricted to the pointer and mark fields and a fixed number of additional pointers and other constant width variables.

To see that this is likely to be the case, we make the following initial observations. First, because only one link may be processed at a time on a node, there must be at least three sets being manipulated by the algorithm (i.e., no links processed, one link processed, and both links processed). D–S–W and our variations all use four sets. To test membership in a fixed time, the algorithm must be able to check a boolean field in each node which is set when the node is added to one of the sets. (This can be shown to be the case by noting that, otherwise, the algorithm would have to scan through all of its pointers and all of its sets looking for the node since, in a cyclic graph, a pointer can lead anywhere, and hence, no nodes could be excluded from the search.) This field can only be the mark field. With the mark field in use, the algorithm must encode al other necessary information (e.g., which of the three or more sets the node is a member of) in the link fields of the data structure. Studies of the information needed in these fields merely to define the structure, however, indicate that the space is probably not available in a form which can be accessed in fixed time. The possibility of merely maintaining a fixed average time for each of the operations which does not vary when n is changed is also unlikely to be possible, since, as indicated above, as n increases, the nodes requiring nonconstant time for any of the operations must be a decreasing fraction of all the nodes, but the structures which require nonconstant time can appear with increasing frequency in the graph. Because these last two points have not been formally proven yet, this limitation on marking algorithms is now only a conjecture.

6. Conclusion

We have demonstrated that a tagless variation on the Deutsch–Schorr–Waite algorithm exists

which has O(1) space, but O(n log n) (worst case O(n^2)) time. We have also demonstrated that a second variation which only works on acyclic graphs can achieve linear time as well as constant workspace. We have also indicated why linear time and constant workspace may not be simultaneously attainable by any algorithm on all graphs. If true, this would imply that, while there is space for additional information in the pointer fields of the nodes, this information is not available for rapid exploitation without placing additional constraints on the storage format. The questions that remain to be answered are whether the conjecture is correct, and whether or not an algorithm can be constructed having either better average or better worst-case time complexity than our variations on D–S–W. If it turns out that linearity is impossible, a next best step might be an algorithm that could be applied to all graphs and that was linear for a significant subset of them (e.g., all acyclic graphs). This is a research problem, too.

Acknowledgment

I am grateful to Gary Knott who introduced me to this problem and has been encouraging me to write this paper for the past three years. David Gries, in his initial critique of my writing, was also of immeasurable assistance in helping me to see what was needed for the presentation of these results. Finally, I would like to thank Westinghouse and the Science Service for the financial support for college they have given me based on the results presented here.

References

[1] J. Cohen, Garbage collection of linked data structures, Computing Surveys 13 (3) (1981) 341–367.

[2] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (Addison-Wesley, Reading, MA, 2nd ed., 1973).

[3] G. Lindstrom, Scanning list structures without stacks or tag bits, Inform. Process. Lett. 2 (2) (1973) 47–51.

[4] G. Lindstrom, Copying list structures using bounded workspace, Comm. ACM 17 (4) (1974) 198–202.

[5] J.M. Morris, Traversing binary trees simply and cheaply, Inform. Process. Lett. 9 (5) (1979) 197–200.