Schorr-Waite Graph Marking Algorithm –Developed With Style

David Gries, 2006

The Schorr-Waite graph marking algorithm appeared first in about 1968; it is also attributed to Peter Deutsch. The idea is to mark the nodes of a graph using no space except one bit per node. The algorithm is extremely useful during garbage collection, where little space is available (that's the reason for garbage collection), and the first step is to mark all reachable nodes so that the unreachable, and thus unmarked, nodes can be collected and used again.

The traditional, recursive, depth-first search algorithm requires space proportional in the worst case to the size of the graph, because each recursive call requires a frame or activation record. Done iteratively, depth-first search requires a stack of nodes from the root to the node that is currently being processed. The Schorr-Waite algorithm ingeniously manipulates the pointers in the graph so that the stack of nodes is encoded in the graph itself. However, their presentation of the algorithm was horrible, very difficult to understand.

In the late 1970's, I developed the Schorr-Waite algorithm afresh, with the goal of formally proving correct an algorithm that used arrays. Along the way, I used principals and strategies of program development that were the hallmark of the people doing research in the formal development of programs. Out came a beautifully simple presentation of the algorithm. The formal proof is presented in my paper, "The Schorr-Waite graph marking algorithm", *Acta Informatica 11* (1979), 223-232. Today, I want to present the algorithm in a less formal way to show you what asking the right questions and paying attention to correctness at all stages can do.

The complete development and explanation are not presented here; they will be done in class.

Representation of nodes of the binary directed graph

Consider a binary directed graph, with each node `i`, for `i` in `1..n-1`, having the form:

	m	l	r
i	0..3	index of left successor	index of right successor

Mark field `m[i]` contains an integer in the range 0..3; initially, it is 0, and the purpose of the algorithm is to set `m[i]` to 3 for each node reachable from a given root (see below). A node may of course have other fields.

We eliminate some case analysis by assuming that the absence of a left `l[i]` or right successor `r[i]` is represented by `0` and that node `0` has the form shown below. Node `0` will get marked just like any other node.

	m	l	r	
0	0..3	0	0	represents null

Purpose of the algorithm

Let `root` be the index of the root of the graph, `1 ≤ root < n`. Initially, `m[i] = 0` for all `i`, `0 ≤ i < n`. The purpose of the algorithm is to set `m[i]` to `3` for all nodes reachable from node `root`.

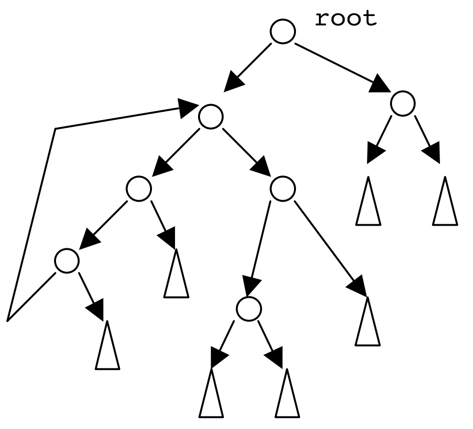
Depth-first traversal of the graph

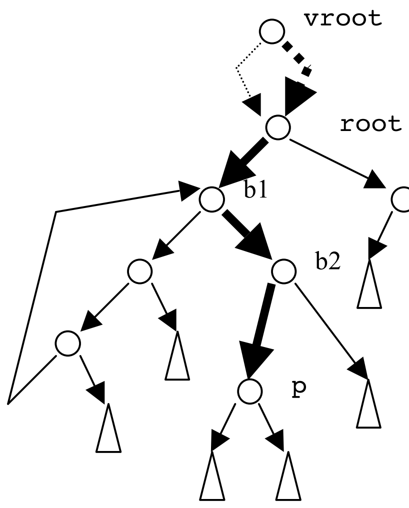
We can write easily the following recursive procedure:

```
/** Set m[i] to 3 for all nodes i reachable from node p along a path
   of nodes with 0 mark fields. Precondition: m[p] = 0.  */
public void mark(int p) {
  /* Visit node p */
 m[p]= 3;
  if m[l[p]] == 0 then mark(l[p]);
  /* Visit node p a second time */
  if m[r[p]] == 0 then mark(r[p]);
  /* Visit node p a third time */
}
```

This recursive procedure requires a frame for each recursive call. To remove this requirement, we write the algorithm iteratively. Note carefully what the recursive algorithm does. For each node `p`, it "visits" `p`, marks it, calls the procedure recursively to mark its left successor subgraph, "visits" `p` a second time, calls the procedure recursively to mark its right successor subgraph, and "visits" `p` for a third time. So, each node is visited 3 times. Each iteration of the loop of our iterative version will visit one node once.

To the left below is a directed graph. Its root is `root`. The circles represent nodes; left-successor arrows point to the left, and right-successor arrow point to the right. Triangles denote arbitrary parts of the graph, which may contain arrow to other nodes. To make sure you don't think we are working with trees, we show one directed arrow from a node that gives a cycle in the graph.





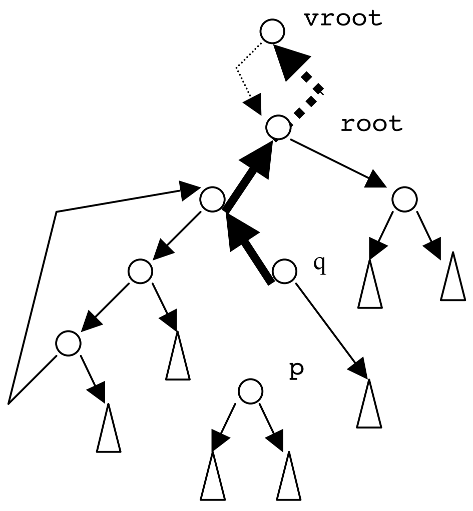
The graph on the right is the same graph but with an extra "virtual node" `vroot`, which will be used to help terminate the algorithm. We use `vroot = -1`. Node `-1` does not exist, of course, and the algorithm does not reference it. We also show a path `(vroot, root, b1, b2, p)`, which illustrates a typical state of execution of depth-first search. Here is part of the invariant of our algorithm:

`P0`: We stipulate that `m[vroot] = 2` (we can make it anything we wish, since it is virtual).
`P1`: For all nodes `i`, `m[i] =` number of times node `i` has been visited.
`P2`: There is a path `P: (vroot, root, …, p)`. `P` starts at `vroot` and has `root` as its second node (if `p` ≠ `vroot`). We state properties of path `P`:
(1) `0 ≤ m[p] ≤ 2`.
(2) For all nodes `i` on `P`, except `p`: (1) `1 ≤ m[i] ≤ 2`.
(3) if `m[i] = 1`, `i`'s successor on path `P` is `l[i]`; otherwise, it is `r[i]`.
`P3`: All unmarked nodes (with `m` field 0) of the graph reachable from `vroot` are reachable along a completely unmarked path from at least one node `i` on path `P`, as follows:
from node `p` if `m[p] = 0` or
from some `r[i]`, if `m[i] = 1`
`P4`: For nodes `i` not on path `P`, `m[i] = 0` or `m[i] = 3`.

Maintaining path P without using extra space

In the recursive depth-first search algorithm, the frames for the recursive calls that are not yet complete contain information about the nodes on path `P`, and this takes space.

Look at the figure in the upper right, above. Node `p` is being processed next. Suppose we maintain a single variable `q` to contain `p`'s predecessor on path `P`. Then, field `l[q]` is not needed, since it can be filled in at any time: we know that `l[q]` is supposed to contain `p`. Therefore, we can put something else in `l[q]`, and we suggest putting `q`'s predecessor on path `P` there. And so it goes. For each node `i` (say) on path `P`, its *backpointer* B_i is the index of its predecessor on `P`.



This requires changing the graph as the depth-first search proceeds. This is the clever idea of Schorr-Waite and Deutsch. But they were not able to present it clearly and simply. To present the idea clearly, we need some notation. For each node `i` of the graph:

- L_i is the value originally in `l[i]`,
- R_i is the value originally in `r[i]`,
- B_i is `i`'s predecessor on path `P`, if i is on path `P` (and if `i` is not `vroot`).

So, for nodes on path `P`, three values must be maintained. The following table describes what is in each node `i`, depending on the value of `m[i]`. The table also shows what is in variable `q`, which contains one of the three values for node `p`. Note that when `m[i]` is 0 or 3, `l[i]` and `r[i]` contain their initial values. So, if path `P` consists only of `vroot`, the graph links contain their initial values.

m[i]	**l[i]**	**r[i]**	**q**
0	L_i	R_i	B_p
1	R_i	B_i	L_p
2	B_i	L_i	R_p
3	L_i	R_i	

The table has one strange thing. Suppose `m[i] = 1`. When asked what value should go in `l[i]` and `r[i]`, practically everyone suggests `l[i]` = B_i and `r[i]` = R_i. When developing the iterative algorithm, recognizing that there is a choice, I tried both choices. The choice used in the table ends up yielding a simpler algorithm, one with less case analysis. The moral of the story is that, when faced with a choice, don't blindly take the "obvious" one but investigate all choices and see which is best.

The algorithm then takes this form:

```
p:= root; q:= vroot;
// invariant: P0, P1, P2, P3, P4, and the table
while p ≠ vroot {
  // Visit node p once
}
```

You can easily see that the invariant is true initially and that, upon termination, when the invariant is true and `p = vroot`, that the result holds: the mark fields of all nodes reachable from root contain 3. So, it remains to determine how to make progress in the repetend and how to keep the invariant true while making progress.

Making progress is easy: add 1 to `m[p`. How the invariant is maintained depends on node `p`, *after 1 is added to* `m[p]`.

1. Case `m[p] = 3`. This is the third visit to node `p`. From invariant `P3`, all unmarked nodes are reachable along an unmarked path from some node other than `p` or its two successors. So, `p` should be removed from path `P`, and its backpointer B_p should become the new node `p`. Also, the values of `l[p]`, `r[p]`, and `q` have to be changed because 1 was added to `m[p]`. Handled by the then-part of the conditional statement below.
2. Case `m[p] = 1` and `m[`L_p`] = 0`. This is the first visit to node `p`, and its left successor L_p is unmarked. L_p should be appended to path `p`. Also, the values of `l[p]`, `r[p]`, and `q` have to be changed because 1 was added to `m[p]`. Handled by the then-part of the conditional statement below.
3. Case `m[p] = 1` and `m[`L_p`] ≠ 0`. This is the first visit to node `p`, and its left successor L_p is either completely marked or is on path `P`. So, there is nothing to do with `p`'s left successor. However, the values of `l[p], r[p]`, and `q` have to be changed because 1 was added to `m[p]`. Handled by the else-part of the conditional statement below.
4. Case `m[p] = 2` and `m[`R_p`] = 0`. This is the second visit to node `p`, and its right successor R_p is unmarked. R_p should be appended to path `p`. Also, the values of `l[p]`, `r[p]`, and `q` have to be changed because 1 was added to `m[p]`. Handled by the then-part of the conditional statement below.
5. Case `m[p] = 2` and `m[`R_p`] ≠ 0`. This is the second visit to node p, and its right successor R_p is either completely marked or is on path P. So, there is nothing to do with p's right successor. However, the values of `l[p], r[p]`, and `q` have to be changed because 1 was added to `m[p]`. Handled by the else-part of the conditional statement below.

The algorithm is given on the next page. A number of choices led to the extreme simplicity of this algorithm.

- The decision to represent null, or the absence of a successor, by node 0.
- The decision on where to place L_i, R_i, and B_i, making them rotate through `l[i]` and `r[i]`.
- The decision to have a mark field be a value in 0..3 and to have its value be the number of times the node has been visited. Instead, most implementations use two bits, 1 is to be changed from 0 to 1 during the

marking phase and the other helps maintain knowledge of whether the left or the right successor of a node on path P is being processed. If you want to use this scheme with our algorithm, just change how the integers 0..3 are represented in two bits: Let 0 be 00, 1 be 10, 2 be 11, and 3 be 01. Then, the second bit is the mark bit.

- The observation that the recursive algorithm "visits" each node three times and the decision to write a single loop, each iteration of which has a distinct purpose: visit one node once.
- The introduction of virtual node `vroot = –1`, to provide a simple termination mechanism.
- Extreme care in writing down the invariant of the algorithm, in logically disjoint pieces.
- Use of the multiple assignment `x1, x2, x3, …:= e1, e2, e3, …`.

```
p:= root; q:= vroot;
// invariant: P0, P1, P2, P3, P4, and the table
while p ≠ vroot {
    // Visit node p once
    m[p]:= m[p] + 1;

    if m[p] = 3  or  m[l[p]] = 0
    then p, l[p], r[p], q:= l[p], r[p], q, p
    else l[p], r[p], q:= r[p], q, l[p]
}
```

Here is a *precise* calculation of the work involved in this marking algorithm. Suppose the reachable part of the graph has `n` nodes. Since each node is visited 3 times, there are `3n` loop iterations. How many four-way permutations are made? Well, `m[p]` becomes 3 exactly once, and if a successor to node `p` is found to have a zero mark field, it is made nonzero on the next iteration. So, in total, there are `2n` 4-way permutations. This means that there are `n` 3-way permutations. What could be simpler?

Below, I show one program taken off the web. It is for some course at some university, and the text around the program says that it is based on chapter 4 of Jones and Lins. I do not know that book. However, you can find on the web lots of presentations and proofs of the graph-marking algorithm that use the ideas mentioned above.

```
// assume all mark bits and all flag bits are 0
procedure mark(R):
   current= R;
   prev= null;
   while true do
      // follow left pointers
      while current != null && current->markBit == 0 do
            current->markBit = 1;
            if current refers to a non-atomic object then
               next= current->left; current->left= prev;
               prev= current; current= next;
      // end of while current
      // retreat
      while prev != null && prev->flagBit == 1 do
            prev->flagBit= 0; next= prev->right;
            prev->right= current; current= prev;
            prev= next;
      // end of while previous
      if prev == null then
          return;
      // switch to right subgraph
      prev->flagBit= 1;
      next= prev->left;
      prev->left= current;
      current= prev->right;
      prev->right= next;
   // end of while true
```