Identifying Loops In Almost Linear Time

G. RAMALINGAM
IBM T.J. Watson Research Center

Loop identification is an essential step in performing various loop optimizations and transformations. The classical algorithm for identifying loops is Tarjan's interval-finding algorithm, which is restricted to reducible graphs. More recently, several people have proposed extensions to Tarjan's algorithm to deal with irreducible graphs. Havlak presents one such extension, which constructs a loop-nesting forest for an arbitrary flow graph. We show that the running time of this algorithm is quadratic in the worst-case, and not almost linear as claimed. We then show how to modify the algorithm to make it run in almost linear time. We next consider the quadratic algorithm presented by Sreedhar et al. which constructs a loop-nesting forest different from the one constructed by the Havlak algorithm. We show that this algorithm too can be adapted to run in almost linear time. We finally consider an algorithm due to Steensgaard, which constructs yet another loop-nesting forest. We show how this algorithm can be made more efficient by borrowing ideas from the other algorithms discussed earlier.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; E.1 [**Data**]: Data Structures—*graphs*; *trees*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Irreducible flowgraphs, loops

1. INTRODUCTION

Loop identification is an interesting control flow analysis problem that has several applications. The classical algorithm for identifying loops is Tarjan's interval-finding algorithm [Tarjan 1974], which is restricted to reducible graphs. More recently, several people have proposed extensions to Tarjan's algorithm to deal with irreducible graphs. In this article, we study and improve three recently proposed algorithms for identifying loops in an irreducible graph.

The first algorithm we study is due to Havlak [1997]. We show that the running time of this algorithm is quadratic in the worst case, and not almost linear as claimed. we then show how to modify the algorithm to make it run in almost linear time.

We next consider the quadratic algorithm presented by Sreedhar et al. [1996] which constructs a loop-nesting forest different from the one constructed by the Havlak algorithm. We show that this algorithm too can be adapted to run in almost linear time.

Author's address: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: rama@watson.ibm.com

In the final section, we present yet another loop-nesting forest defined by Steensgaard [1993], and discuss how aspects of the Sreedhar et al. algorithm can be combined with Steensgaard's algorithm to improve the efficiency of Steensgaard's algorithm.

2. TERMINOLOGY AND NOTATION

A flowgraph is a connected directed graph $(V, E, START, END)$ consisting of a set of vertices V, a set of edges E, with distinguished start and end vertices $START, END \in V$. We will assume, without loss of generality, that $START$ has no predecessors. We will denote the number of vertices in the given graph by n and the number of edges in the given graph by m.

We assume that the reader is familiar with depth-first search [Hopcroft and Tarjan 1973] (abbreviated DFS) and depth-first search trees. (See Cormen et al. [1990], for example.) An edge $x \rightarrow y$ in the graph is said to be a DFS tree edge if x is the parent of y in the DFS tree, a DFS forward edge if x is an ancestor other than the parent of y in the DFS tree, a DFS backedge if x is a descendant of y in the DFS tree, and a DFS cross edge otherwise. (We will omit the prefix "DFS" if no confusion is likely.) It is straightforward to augment DFS to compute information that will help answer ancestor relation queries (of the form "is u an ancestor of v in the DFS tree") in constant time. (See Havlak [1997], for example.) We will refer to the order in which vertices are visited during DFS as the "DFS order."

We also assume that the reader is familiar with the concepts of reducible and irreducible flowgraphs. See Aho et al. [1986] for a discussion of these concepts.

We denote the inverse Ackermann function by $\alpha(i, j)$. The inverse Ackermann function is a very slowly growing function and may be considered to be a constant for practical purposes. See Cormen et al. [1990] for a discussion of this function.

There does not appear to be any single well-accepted definition what a loop is. For certain irreducible flowgraphs, each of the three algorithms considered in this article will identify a different set of loops. The suitability of each of these algorithms depends on the intended application. However, the following few facts hold true for all these three algorithms. A loop corresponds to a set of vertices in the flowgraph. If L_x and L_y are two loops identified by one of these algorithms, then either L_x and L_y will be mutually disjoint or one will be completely contained in the other. Hence, the nesting (or containment) relation between all the loops can be represented by a *forest*, which we refer to as the loop-nesting forest (identified by the corresponding algorithm).

A vertex belonging to a loop is said to be an *entry* vertex for that loop if it has a predecessor outside the loop.

Given a flowgraph, the algorithms described in this article (conceptually) modify the flowgraph as the execution proceeds. Thus, when we refer to the flowgraph, it is worth remembering that we do not mean a fixed input flowgraph, but a flowgraph that constantly changes during the course of the execution. The changes to the flowgraph, however, are not explicitly represented. Instead, a UNION-FIND data structure is used to implicitly represent the changes in the flowgraph.

3. TARJAN'S ALGORITHM FOR REDUCIBLE GRAPHS

We begin with a brief description of Tarjan's loop-nesting forest and his algorithm to construct it. Consider a reducible graph. Every vertex w that is the target of a backedge identifies a loop L_w with w as its header. Let B_w be the set $\{z|z \rightarrow w \text{ is a backedge}\}$. The loop L_w consists of w and all other vertices in the graph that can reach some vertex in B_w without going through w. For any two loops L_x and L_y, either L_x and L_y must be disjoint, or one must be completely contained in the other. Hence, the nesting (or containment) relation between all the loops can be represented by a forest, which yields the loop-nesting forest. This provides the definition of Tarjan's loop-nesting forest (for reducible graphs), and let us now see how this forest can be constructed efficiently.

Tarjan's algorithm performs a bottom up traversal of the depth-first search tree, identifying inner (nested) loops first. When it identifies a loop, the algorithm "collapses" it into a single vertex. (If X is a set of vertices, then by collapsing X we mean replacing the set of vertices X by a single representative vertex r_X in the graph. Any other vertex y is a successor or predecessor of r_X in the collapsed graph if and only if y is a successor or predecessor of some vertex in X in the original graph.) A vertex w visited during this traversal is determined to be a loop header if it has any incoming backedges. As explained above, let B_w be the set $\{z|z \rightarrow w \text{ is a backedge}\}$. The children of w in the loop-nesting forest[1] are identified by performing a backward traversal of the *collapsed* graph, identifying vertices that can reach some vertex in B_w without going through w. Once the children of w have been identified, w and all its children are merged (collapsed) together into a single vertex that identifies the newly constructed loop L_w. The traversal then continues on to the next vertex.

In the implementation, the collapsing of vertices is achieved using the classical UNION-FIND data structure. (See Tarjan [1983] and Cormen et al. [1990].) Thus, the outermost loops identified so far are each maintained as a set. The FIND operation on any vertex x returns the header of the outermost loop containing x (or the vertex x itself if it is not in any loop). A set of vertices is collapsed by performing a UNION operation on all the vertices in the set. A complete description of the algorithm in pseudocode appears in Figure 1.

Let us analyze the complexity of this algorithm. Procedure `findloop` is invoked exactly once for every vertex. Hence, line [8] is executed once for every vertex. Lines [10]-[16] are executed at most once for every vertex y, when the innermost loop containing y is identified. As a result, the total cost of executing lines [8] and [10]-[16] is to perform at most one FIND operation per edge in the original graph. Similarly, lines [3]-[4] are executed at most once for every vertex z, which costs one UNION operation. The whole algorithm performs at most n UNION operations and at most m FIND operations, where n denotes the number of vertices in the graph, and m denotes the number of edges in the graph. Hence, the whole algorithm runs in time $O((m+n)\alpha(m+n,n))$, if UNION-FIND is implemented using the standard path compression and union-by-rank techniques [Tarjan 1983; Cormen et al. 1990].

[1]Strictly speaking, we mean the children of the node representing the loop L_w. However, we simplify matters somewhat by using the header vertex w to represent the loop L_w in the loop-nesting forest. We can do this here, since a vertex is the header of at most one loop.

```
[1]     procedure collapse($loopBody$, $loopHeader$)
[2]         for every $z \in loopBody$ do
[3]             loop-parent($z$) := $loopHeader$;
[4]             LP.union($z$, $loopHeader$); // Use $loopHeader$ as representative of merged set
[5]         end for

[6]     procedure findloop($potentialHeader$)
[7]         $loopBody = \{\}$;
[8]         $worklist = \{$ LP.find($y$) $|$ $y \rightarrow potentialHeader$ is a backedge $\} - \{potentialHeader\}$;
[9]         while ($worklist$ is not empty) do
[10]            remove an arbitrary element $y$ from $worklist$;
[11]            add $y$ to $loopBody$;
[12]            for every predecessor $z$ of $y$ such that $z \rightarrow y$ is not a backedge do
[13]                if (LP.find($z$) $\notin$ ($loopBody \cup \{potentialHeader\} \cup worklist$)) then
[14]                    add LP.find($z$) to $worklist$;
[15]                end if
[16]            end for
[17]        end while
[18]        if ($loopBody$ is not empty) then
[19]            collapse ($loopBody$, $potentialHeader$);
[20]        end if

[21]    procedure TarjansAlgorithm ($G$)
[22]        for every vertex $x$ of $G$ do loop-parent($x$) := NULL; LP.add($x$); end for
[23]        for every vertex $x$ of $G$ in reverse-DFS-order do findloop($x$); end for
```

Fig. 1. Tarjan's algorithm for constructing the loop-nesting forest of a reducible graph. LP is a partition of the vertices of the graph. The function LP.add(z) initially places z in an equivalence class by itself. The function LP.union(u, v) merges u's and v's classes into one, using v as the representative element for the merged class. The function LP.find(z) returns the representative element of z's equivalence class.

4. HAVLAK'S ALGORITHM

Havlak [1997] recently presented an extension of Tarjan's algorithm that handles irreducible graphs as well. We show here that this algorithm is potentially quadratic, even though Havlak describes the algorithm as being almost linear. More precisely, we show that the algorithm, in the worst case, may take $\theta(n^2)$ time, even for graphs in which the number of edges is $O(n)$.

Havlak's extension of Tarjan's algorithm modifies the loop body identification step as follows. Given a vertex *potentialHeader*, the children of *potentialHeader* (in the loop-nesting forest) are identified by performing a backward traversal of the collapsed graph, as before, *but the traversal is restricted to the set of descendants of* potentialHeader *in the DFS tree.* In particular, lines [13]-[15] of Tarjan's algorithm (Figure 1) are modified so that these lines are executed only if z is a descendant of *potentialHeader* in the DFS tree; if z is not a descendant of *potentialHeader* in the DFS tree, then the edge $z \rightarrow y$ is ignored and replaced by the edge $z \rightarrow$ *potentialHeader* (in the collapsed flowgraph). (Note that in a reducible graph, z is guaranteed to be a descendant of *potentialHeader* in the DFS tree.)

The last step described is precisely the source of the problem. It is possible for a single edge $z \rightarrow y$ to be processed multiple times, each time as an edge of the form $z \rightarrow w$, where w is the header of a loop containing y. The example shown

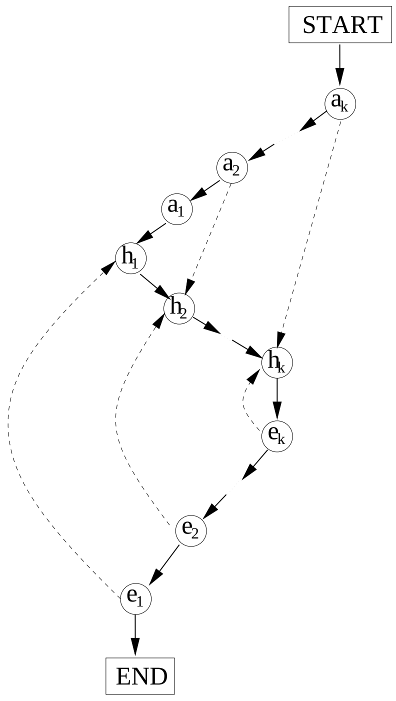

Fig. 2. A counterexample illustrating that Havlak's algorithm may perform a quadratic number of UNION-FIND operations. Solid lines indicate DFS tree edges, while dashed lines indicate the remaining graph edges.

in Figure 2 illustrates this. Note that the vertices h_k down to h_1 are targets of backedges and, hence, identified as loop headers in that order. Their loop bodies are also constructed in that order. The edge $a_k \rightarrow h_k$ will be processed $k-1$ times, as it is replaced successively by edges $a_k \rightarrow h_i$ for every $i < k$. Similarly, every edge $a_j \rightarrow h_j$ will be processed $j-1$ times. Thus, the algorithm will end up performing $\theta(n^2)$ UNION-FIND operations in this example.

The above example presents a lower bound on the complexity of Havlak's algorithm. This is also an upper bound for Havlak's algorithm. In particular, the modified loop in lines [12]-[15] will perform at most n FIND operations. Since lines [10]-[16] may be performed once for every vertex, the whole algorithm performs $O(n)$ UNION operations and $O(n^2)$ FIND operations, which implies an upper bound of $O(n^2\alpha(n^2, n))$ on the running time of the algorithm. Since $\alpha(n^2, n)$ is $O(1)$ (see Tarjan [1983]), the upper bound simplifies to $O(n^2)$.

5. AN ALMOST LINEAR TIME VERSION OF HAVLAK'S ALGORITHM

We now describe a modification of Havlak's algorithm that does run in almost linear time.

Given a vertex a_0 in a control flow graph, consider a_0's ancestors in Havlak's loop nesting forest. (See Figure 3.) In particular, for every $i \geq 0$, let a_{i+1} denote the header of the innermost loop containing a_i (in Havlak's loop-nesting forest). Thus,

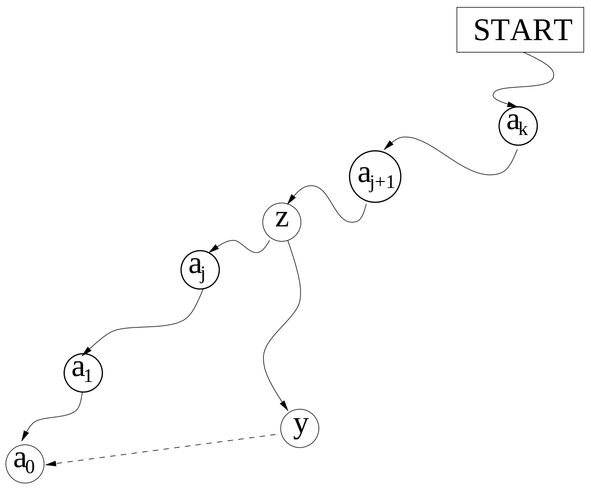

Fig. 3. Modifying Havlak's algorithm to run in almost linear time. $y \to a_0$ is an edge in the flowgraph. $a_1, \cdots, a_k$ (shown as bold vertices) are the ancestors of a_0 in the DFS tree that are identified as loop headers by Havlak's algorithm. z is the least common ancestor of a_0 and y (in the DFS tree). a_j is a proper descendant of z, which is a descendant of a_{j+1}. In Havlak's algorithm, the edge $y \to a_0$ will not be "used" while constructing the bodies of loops with headers a_1 through a_j. It will be used only in the construction of the loop body of a_{j+1}.

$a_1, a_2, \cdots, a_k$ is the sequence of loops containing a_0, from innermost to outermost, as identified by their headers. Note that each a_{i+1} must be an ancestor of a_i in the DFS tree. Now, consider any edge $y \to a_0$. Consider the largest j such that y is not a descendant of a_j in the DFS tree. In other words, y is a descendant of a_{j+1}but not a_j in the DFS tree.

Consider how Havlak's algorithm processes the edge $y \to a_0$. For every $1 \leq i \leq j$, when the body of the loop with header a_i is constructed, the edge $y \to a_{i-1}$ will be considered. Since the source of the edge y is not a descendant of a_i, the edge will be replaced by edge $y \to a_i$. Finally, when the body of the loop with header a_{j+1} is constructed, the edge $y \to a_j$ will appear to be a "proper edge," and vertex y will be added to the loop body.

What would be desirable is to replace the edge $y \to a_0$ by the edge $y \to a_j$ in one step instead of in j steps. It turns out that we can do this. Let z denote the least common ancestor of a_0 and y in the DFS tree. Note that z must lie between a_j and a_{j+1} (in the DFS tree). Consider the moment when Havlak's algorithm visits z in the bottom-up traversal of the DFS tree. At this point, loops with headers a_1 through a_j have been identified, and the loop with header a_{j+1} is yet to be constructed. A FIND operation on a_0 will return a_j at this stage. This suggests the following algorithm.

In an initial pass, we remove every cross edge and forward edge $y \to x$ in the graph and attach it to a list associated with the least common ancestor of y and x. We can do this in almost linear time (see Tarjan [1979] or Cormen et al. [1990, problem 22-3]). We then run Havlak's algorithm, modified as follows. Whenever the main bottom-up traversal visits a vertex w, it processes the list of cross/forward

```
[1]     procedure markIrreducibleLoops($z$)
[2]         $t$ := loop-parent($z$);
[3]         while ($t \neq$ NULL) do
[4]             $u$ = RLH.find($t$);
[5]             mark $u$ as irreducible-loop-header;
[6]             $t$ := loop-parent($u$);
[7]             if ($t \neq$ NULL) then RLH.union ($u$, $t$); end if
[8]         end while

[9]     procedure processCrossFwdEdges($x$)
[10]        for every edge $y \rightarrow z$ in crossFwdEdges[$x$] do
[11]            add edge $find(y) \rightarrow find(z)$ to the graph;
[12]            markIrreducibleLoops($z$);
[13]        end for

[14]    procedure ModifedHavlakAlgorithm ($G$)
[15]        for every vertex $x$ of $G$ do
[16]            loop-parent($x$) := NULL; crossFwdEdges[$x$] := {};
[17]            LP.add($x$); RLH.add($x$);
[18]        end for
[19]        for every forward edge and cross edge $y \rightarrow x$ of $G$ do
[20]            remove $y \rightarrow x$ from $G$ and add it to crossFwdEdges[least-common-ancestor($y$,$x$)];
[21]        end for
[22]        for every vertex $x$ of $G$ in reverse-DFS-order do
[23]            processCrossFwdEdges($x$);
[24]            findloop($x$); // Procedure findloop is the same as in Figure 1
[25]        end for
```

Fig. 4. The modified version of Havlak's algorithm. RLH is a second UNION-FIND data structure used to map loop headers to the header of the innermost reducible loop containing them.

edges $y \rightarrow x$ associated with it (by the first pass) and adds the edge $FIND(y) \rightarrow FIND(x)$ to the graph. (It is immaterial whether we add the edge $y \rightarrow FIND(x)$ or the edge $FIND(y) \rightarrow FIND(x)$ to the graph.) The modified algorithm appears in Figure 4. Note that this modification implies that we can use procedure *findloop* of Tarjan's algorithm unchanged.

The modified algorithm runs in almost linear time and constructs the same loops and loop-nesting forest as Havlak's algorithm. However, it is not quite complete yet. In addition to constructing the loop-nesting forest, Havlak's algorithm also marks loops as being reducible or irreducible. It is not as straightforward to distinguish reducible loops from irreducible loops in the modified algorithm described above. We now show how this extra piece of information can be computed, if desired.

Consider the example in Figure 3. The presence of the edge $y \rightarrow a_0$ means that the loops with headers a_1 through a_j are irreducible. Hence, when our algorithm replaces edge $y \rightarrow a_0$ by the edge $FIND(y) \rightarrow a_j$, as explained above, we need to mark the loop headers a_1 through a_j as being irreducible. Procedure *markIrreducibleLoops* of Figure 4 does this by walking up the loop-nesting tree containing a_0. If we do this naively, as explained, the algorithm will end up being quadratic again. We avoid such a quadratic behavior using the standard path compression technique. In particular, consider lines [5]-[6], which mark a vertex u as an irreducible loop header and traverses up to its parent t. Let us say that this step *scans* the loop-tree edge $u \rightarrow t$. We utilize a second UNION-FIND data structure so that we *scan* every loop-tree edge at most once. In particular, the UNION operation in line [7] ensures that the tree edge $u \rightarrow t$ will never be scanned again, since the FIND operation in line [4] skips past all previously scanned edges. This is safe, since there is no reason to mark a vertex (as irreducible) again if it has already been marked. The resulting algorithm runs in almost linear time.

6. THE SREEDHAR-GAO-LEE ALGORITHM

Sreedhar et al. [1996] present a different algorithm for constructing a loop-nesting forest. This algorithm utilizes the DJ graph, which essentially combines the control flow graph and its dominator tree into one structure. We will, however, simplify our discussion of this algorithm by using just the control flow graph and the dominator tree instead of the DJ graph. Let $level(u)$ denote the depth of node u from the root of the dominator tree, with the root being at level 0. Let V_i denote the set of vertices at level i (i.e., the set of vertices u such that $level(u) = i$). Let p denote the maximum level in the dominator tree.

The Sreedhar et al. algorithm processes the vertices in the *dominator tree* bottom up. In particular, each level l from p down to 1 is processed as follows. The first step identifies all *reducible* loops at level l. All vertices at level l are scanned, and any vertex n that has one or more incoming backedges whose source is dominated by n is identified as the header of a reducible loop. The body of such a reducible loop is identified just as in Tarjan's algorithm, traversing the graph backward from the sources of the backedges, identifying vertices that can reach these backedges without going through n. The reducible loop is then collapsed into a single vertex, just as in Tarjan's algorithm.

If any vertex n at level l has one or more incoming backedges whose source is *not* dominated by n, then n is one of the entries to an *irreducible* loop. Once all vertices at level l have been processed to identify reducible loops, we construct the irreducible loops of level l. (We do this only if some vertex n at level l has one or more incoming backedges whose source is *not* dominated by n.) This requires processing the subgraph of the (collapsed) flowgraph consisting of all vertices at a level greater than or equal to the current level l (that is, the set of vertices $\bigcup_{j \geq l} V_j$) to identify its strongly connected components (SCCs). Each nontrivial strongly connected component of this graph is an irreducible loop of level l and is collapsed to a single vertex. By a "nontrivial SCC" we mean an SCC consisting of more than one vertex.

We now establish a property of the loops identified by this algorithm, which will be useful subsequently.

LEMMA 1. *A vertex can be an entry vertex of at most one irreducible loop.*

PROOF. Note that any two irreducible loops (identified) at the same level l are mutually disjoint. Hence any two such loops cannot have a common entry vertex. Let L be an irreducible loop identified at level l. We show below that any entry vertex of L must also be a vertex at level l. This immediately implies that irreducible loops belonging to different levels cannot share a common entry vertex either, and the lemma follows.

Let F denote the subgraph *of the dominator tree* consisting only of vertices at

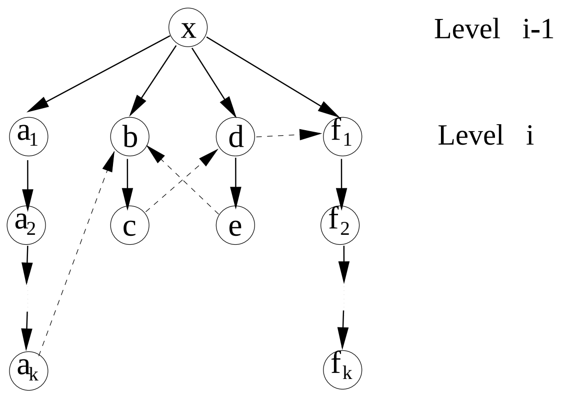

Fig. 5. An example illustrating the source of the quadratic behavior in the Sreedhar et al. algorithm. Solid edges belong to both the control flow graph and the dominator tree, while the dashed edges are control flow graph edges that are not in the dominator tree.

a level greater than or equal to l. Thus, F is a forest consisting of subtrees of the dominator tree. First note that the loop L must consist of vertices from at least two different trees in F–a loop consisting of vertices from only one tree must be a reducible loop of level l or a loop of level greater than l.

Let u and v be vertices belonging to different trees in F, and let w be the root of the tree containing v. Then, any path (in the flowgraph) from u to v must pass through w. (Otherwise, w would not be a dominator of v.) Hence, if the loop L contains a vertex v, it must also contain the root of the tree in F that contains v.

Now, let v be a vertex in loop L. Let w be the root of the tree containing v, and assume that $v \neq w$. Then, any predecessor x of v (in the flowgraph) must also be in the tree rooted at w. This is a straightforward property of the dominator tree. It follows that x must also be in the loop L, since there is a path from x to a vertex in the loop (namely v), and there is a path from a vertex in the loop (namely w) to x.

This establishes that any vertex in loop L with a predecessor outside L must be the root of a tree in F. But the roots of the trees in F are precisely the vertices at level l. The result follows. $\square$

Sreedhar et al. show that the algorithm described above runs in time $O(m\alpha(m,n) + km)$, where k is the number of levels at which the strongly connected component algorithm had to be invoked. In the worst case, k can be $O(n)$, resulting in a quadratic algorithm.

The example in Figure 5 illustrates the source of the quadratic behavior in this algorithm, which is the repeated application of the SCC algorithm. Consider the processing done at level i for this example. This level contains an irreducible loop consisting of the vertices b, c, d, and e. Constructing this irreducible loop requires identifying the SCCs of the graph consisting of vertices a_1 through a_k, b, c, d, e, and vertices f_1 through f_k. Notice that vertices a_1 through a_k and f_1 through f_k are visited, but they do not belong to any nontrivial SCC. When we similarly apply the SCC algorithm at level $i-1$ (or at any lower level), we may end up visiting vertices a_1 through a_k and f_1 through f_k again. In the worst case we may end up visiting these vertices i times, resulting in the quadratic complexity.

We now show that a careful implementation of the SCC identification phase can ensure that the algorithm runs in almost linear time. Observe that once the vertices b, c, d, and e are collapsed into a single vertex, say L, representing the irreducible loop, these vertices will never be visited again. (It is true that the edges $a_k \rightarrow b$ and $d \rightarrow f_1$ may be visited later on; however, these edges actually represent the edges $a_k \rightarrow L$ and $L \rightarrow f_1$ in the collapsed graph, and the cost of visiting these edges can be attributed to the cost of visiting vertex L.)

Our goal is to perform the irreducible loop construction at level i such that a vertex x at a level $j > i$ is visited only if x belongs to some irreducible loop at level i. We do this as follows.

Consider any strongly connected component. Consider the vertex u of the component that is visited first during depth-first search. Clearly, all other vertices in the component will be descendants of this vertex in the DFS tree. Thus, if we start with the set of incoming backedges of u and traverse the graph backward, restricting the traversal to vertices that are descendants of u in the DFS tree, we can identify all the vertices belonging to u's strongly connected component without visiting vertices not in u's SCC.

The above process is very similar to the one used by Havlak's algorithm (and Tarjan's algorithm) to identify the loop body corresponding to a potential header vertex. However, note that if we apply the same process, but start from a vertex that was not the first one in its SCC to be visited during DFS, then we will not identify the *complete* SCC. Thus, while Havlak's and Tarjan's algorithms visit potential header vertices in *reverse* DFS order, we do not want to visit vertices in that order.

Instead, we perform the irreducible loop construction at a level l by visiting the set of vertices at level l *in DFS order*. If the visited vertex u belongs to an irreducible loop (of level l) that has already been constructed, we skip the vertex and continue on to the next vertex. Otherwise, if it has an incoming backedge, then it belongs to an irreducible loop. The body of this loop is identified by traversing backward from the sources of all such backedges, restricting the traversal to descendants of u in the DFS tree.

The modified algorithm appears in Figure 6. A few words of explanation are needed, however. Both Tarjan's algorithm and Havlak's algorithm identify at most one loop per *header* vertex. This allowed us to represent a loop by its header vertex in the loop-nesting forest. However, the Sreedhar-Gao-Lee algorithm may identify up to *two* loops per header vertex: a *reducible* loop and an *irreducible* loop. Consequently, we do not use the header vertex itself to represent the loop in the loop-nesting forest; instead, we use a new representative vertex to do this. Further, the algorithm does not explicitly identify loops consisting of a single vertex, but can be modified to do so if desired.

Note that we can construct, for each level, a list of all the vertices in that level in DFS order easily enough. We just initialize all such lists to be empty and visit all vertices in DFS order, appending the visited vertex to the end of the list corresponding to its level.

Let us now analyze the complexity of the algorithm. Observe that lines [6]-[15] get executed at most once for every vertex y, and that these lines perform at most *indegree*(y) FIND operations. However, these lines are executed not only for the

```
[1]  procedure findloop($header$, $worklist$)
[2]    if $worklist$ is not empty then
[3]      create a new vertex $loopRep$ with same predecessors as $header$;
[4]      $loopBody = \{ header \}$;
[5]      while ($worklist$ is not empty) do
[6]        remove an arbitrary element $y$ from $worklist$;
[7]        add $y$ to $loopBody$;
[8]        processed[$y$] := true;
[9]        for every predecessor $z$ of $y$ do
[10]         if (LP.find($z$) is not a descendant of $header$ in the DFS tree) then
[11]           add edge LP.find($z$) $\rightarrow$ $loopRep$ to graph;
[12]         elsif (LP.find($z$) $\notin$ ($loopBody \cup worklist$)) then
[13]           add LP.find($z$) to $worklist$;
[14]         end if
[15]       end for
[16]     end while
[17]     collapse ($loopBody$,$loopRep$);
[18]   end if

[19] procedure ModifiedSreedharGaoLeeAlgorithm ($G$)
[20]   for every vertex $x$ of $G$ do
[21]     loop-parent($x$) := NULL; processed[$x$] := false; LP.add($x$);
[22]   end for
[23]   for $i = p$ down to 1 do
[24]     for every $x \in level[i]$ do
[25]       $W = \{$ LP.find($y$) $|$ ($y \rightarrow x$ is a backedge) and ($x$ dominates $y$) $\}$;
[26]       findloop($x$, $W - \{x\}$);
[27]     end for
[28]     for every $x \in level[i]$ in DFS-order do
[29]       if (not processed[$x$]) then
[30]         $W = \{$ LP.find($y$) $|$ ($y \rightarrow x$ is a backedge) and not ($x$ dominates $y$) $\}$;
[31]         findloop($x$, $W - \{x\}$);
[32]       end if
[33]     end for
[34]   end for
```

Fig. 6. The modified version of the Sreedhar-Gao-Lee algorithm

vertices that exist in the original graph, but also for the vertices that are created in line [3]. The vertices created in line [3] are representatives of loops in the *collapsed* graph. Hence, the complexity of the algorithm depends on the number of such representatives created and their in-degrees.

The created vertices fall into two categories: *reducible loop representatives* and *irreducible loop representatives.* Every vertex h in the original graph is the header of at most one reducible loop, which results in the creation of at most one reducible loop representative r_h, whose in-degree is bounded by the in-degree of h.

Every irreducible loop has two or more entry vertices (which are vertices of the original graph), and the in-degree of the representative of an irreducible loop is bounded by the sum of the in-degrees of its entry vertices. A vertex (in the original graph) can be the entry vertex of at most *one* irreducible loop. Hence, the sum of the in-degrees of all irreducible loop representatives is bounded by m, the number of edges in the original graph.

As a result, the whole algorithm performs $O(n)$ UNION operations and $O(m)$ FIND operations, resulting in a complexity of $O(m\alpha(m,n))$.

7. STEENSGAARD'S LOOP-NESTING FOREST

In this section, we consider yet another loop-nesting forest, defined by Steensgaard [1993]. We outline Steensgaard's algorithm for constructing this forest, as it also serves as a constructive definition of this structure. Steensgaard identifies the loops in a graph in a top-down fashion, identifying outer loops first. The nontrivial strongly connected components of the given graph constitute its outermost loops. A vertex of a loop is said to be a generalized entry node of that loop if it has a predecessor outside the strongly connected component. Any edge from a vertex inside a loop to one of its generalized entry nodes is said to be a generalized backedge. The "inner loops" contained in a given loop are determined by identifying the strongly connected components of the subgraph induced by the given loop, after all its generalized backedges are eliminated. This iterative process yields the loop-nesting forest.

Let us briefly consider the differences between the forest created by Steensgaard's algorithm and the forest created by the Sreedhar-Gao-Lee algorithm. One difference, explained in Sreedhar et al. [1996], is that the Sreedhar-Gao-Lee algorithm may identify some more reducible loops than Steensgaard's algorithm. If the extra step in the Sreedhar-Gao-Lee algorithm to construct reducible loops is eliminated, this difference disappears. However, it is also possible for the Sreedhar-Gao-Lee algorithm to identify fewer loops than Steensgaard's algorithm does.

The problem is that it is not possible, in the Sreedhar-Gao-Lee forest, for one irreducible loop to be nested inside another irreducible loop, if their entry vertices are at the same level in the dominator tree. The example shown in Figure 7 illustrates this. In this example, the Steensgaard algorithm identifies an outer loop consisting of vertices u, v, w, and x and an inner loop consisting of vertices w and x. In contrast, the Sreedhar-Gao-Lee algorithm identifies only the one loop consisting of u, v, w, and x.

We now show that Steensgaard's loop-nesting forest can be constructed more efficiently by borrowing the ideas described in Section 6 and Sreedhar et al. [1996]. We simply modify the irreducible-loop construction phase of the algorithm described in Section 6 as follows: instead of stopping after identifying the strongly connected components, we use a Steensgaard-like algorithm to iteratively find other loops nested inside the irreducible loop. In other words, instead of applying the *strongly connected components algorithm* to the subgraph (of vertices at a level greater than or equal to the current level), we apply *Steensgaard's algorithm* to the subgraph. (Symmetrically, it is also possible to modify Steensgaard's algorithm by replacing the use of a strongly connected components algorithm by the algorithm presented in Section 6.)

The resulting algorithm has the same asymptotic worst-case complexity as Steensgaard's original algorithm, which is quadratic in the size of the graph. However, in practice, it can potentially be more efficient than Steensgaard's original algorithm, since the number of iterations Steensgaard's algorithm performs within a single irreducible loop (identified by the Sreedhar et al. algorithm) is likely to be much smaller than the number of iterations it would perform for the whole graph.

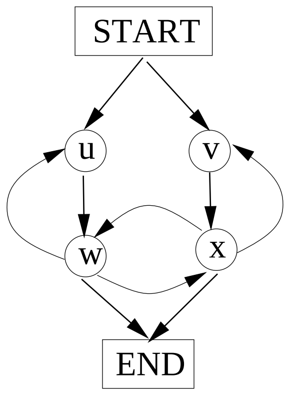

Fig. 7. An example illustrating that the Sreedhar et al. algorithm does not identify all loops identified by the Steensgaard algorithm.

8. CONCLUSION

In this article, we have examined three algorithms for identifying loops in irreducible flowgraphs and shown how these algorithms can be made to be more efficient. As these three algorithms construct potentially different loop-nesting forests, the question arises as to what the relative advantages of these different algorithms are.

Havlak's approach has the disadvantage that the set of loops found and the loop-nesting forest constructed are dependent on the depth-first spanning tree, which is itself dependent on the ordering of the outgoing edges of every vertex. In particular, what is represented as a single irreducible loop with k entry vertices in the Sreedhar-Gao-Lee forest may be represented as k irreducible loops nested within each other in some arbitrary order in Havlak's forest. Hence, we believe that the Sreedhar-Gao-Lee loop-nesting forest is more natural than Havlak's loop-nesting forest.

However, (the modified version of) Havlak's algorithm is simpler to implement than the (modified version of the) Sreedhar-Gao-Lee algorithm, since it does not require the construction of the dominator tree. It would be a worthwhile exercise to adapt Havlak's algorithm to directly construct the Sreedhar-Gao-Lee loop-nesting forest.

On the other hand, the Sreedhar-Gao-Lee loop-nesting forest and Steensgaard's loop-nesting forest are somewhat incomparable. As explained in Section 7, the ideas behind both these approaches can be combined to construct a loop-nesting forest more refined than either one, but the resulting algorithm is more expensive than the almost linear time variation we have presented for constructing the Sreedhar-Gao-Lee forest. Whether this more refined forest is worth the increased algorithm complexity depends on the intended application. We refer the reader to Ramalingam [1999] for a more detailed comparison of these different loop-nesting forests in the context of two applications.

ACKNOWLEDGMENTS

We thank John Field, V. Sreedhar, and the anonymous referees for their helpful comments.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, Reading, MA.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

HAVLAK, P. 1997. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst. 19,* 4 (July), 557–567.

HOPCROFT, J. E. AND TARJAN, R. E. 1973. Efficient algorithms for graph manipulation. *Commun. ACM 16,* 6, 372–378.

RAMALINGAM, G. 1999. On loops, dominators, and dominance frontiers. Tech. Rep. RC21513, IBM Research Division. June.

SREEDHAR, V. C., GAO, G. R., AND LEE, Y.-F. 1996. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst. 18,* 6 (Nov.), 649–658.

STEENSGAARD, B. 1993. Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Wash. Oct.

TARJAN, R. E. 1974. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9*, 355–365.

TARJAN, R. E. 1979. Applications of path compression on balanced trees. *J. ACM 26,* 4, 690–715.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

Received February 1998; revised September 1998; accepted October 1998