On Loops, Dominators, and Dominance Frontiers

G. RAMALINGAM
IBM T.J. Watson Research Center

This paper explores the concept of loops and loop nesting forests of control-flow graphs, using the problem of constructing the dominator tree of a graph and the problem of computing the iterated dominance frontier of a set of vertices in a graph as guiding applications. The contributions of this paper include: (1) An axiomatic characterization, as well as a constructive characterization, of a family of loop nesting forests that includes various specific loop nesting forests that have been previously defined. (2) The definition of a new loop nesting forest, as well as an efficient, almost linear time, algorithm for constructing this forest. (3) An illustration of how loop nesting forests can be used to transform arbitrary (potentially irreducible) problem instances into equivalent acylic graph problem instances in the case of the two problems of (a) constructing the dominator tree of a graph, and (b) computing the iterated dominance frontier of a set of vertices in a graph, leading to new, almost linear time, algorithms for these problems.

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

General Terms: Algorithms, languages

Additional Key Words and Phrases: Dominator, graph transformation, irreducible graph, iterated dominance frontier, least common ancestor, loop, loop nesting forest, sparse evaluation

1. INTRODUCTION

This paper illustrates the use of *loop nesting forests* in two applications. The first is a new algorithm for computing the iterated dominance frontier of a set of vertices in a graph, which can be used to construct representations such as the SSA form [Cytron et al. 1991] and Sparse Evaluation Graphs [Choi et al. 1991]. The second is a new algorithm for constructing the dominator tree [Lengauer and Tarjan 1979] of a graph. The new algorithms run in almost linear time.

The primary contributions of the paper, however, are not these end results (linear time algorithms are already known for these problems), but the means used to achieve these ends. In particular, the paper illustrates for these two problems how arbitrary problem instances (including those based on irreducible graphs) can be transformed into equivalent problem instances based on acyclic graphs. This lets us generalize simpler algorithms that work only for acyclic graphs to work for arbitrary graphs. Such approaches have previously been used for reducible graphs, and this paper generalizes such approaches to irreducible graphs.

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

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1999 ACM 0164-0925/99/0100-0111 $00.75

Our problem reduction strategy utilizes loop nesting forests, a data structure that represents the loops in a control-flow graph and the containment relation between them. The concept of loops is widely used in optimizing compilers [Morgan 1998]. However, while there is a well accepted notion of what the loops in a reducible graph are [Tarjan 1974], there is less agreement about how the loop nesting forest should be defined for arbitrary graphs. For instance, Steensgaard [1993], Sreedhar *et. al.* [1996], and Havlak [1997] each provide a different definition.

Which of these definitions is the "right" one? One could argue that each of these definitions has its merits, and that different applications of loop nesting forest may find it advantageous to use different definitions. Rather than restrict ourselves to one of these definitions, we introduce an axiomatic definition of a loop nesting forest that all three forests satisfy. We also introduce an equivalent constructive characterization of a family of loop nesting forests. We study the properties shared by this family of forests and show that our problem reduction strategy works correctly with any forest belonging to this family.

This implies we can *safely* use any of the three forests mentioned above in our two applications. However, these three forests turn out to be less than ideal for our purpose. Our applications run in linear time if they utilize Steensgaard's forest, but generating Steensgaard's forest takes quadratic time in the worst case. Havlak's forest can be generated in almost linear time [Ramalingam 1999], but our applications run in quadratic time (in the worst case) if they use Havlak's forest. The Sreedhar-Gao-Lee forest poses no such efficiency dilemma: it can be generated in almost linear time [Ramalingam 1999], and our applications run in linear time if they utilize this forest. However, the algorithm for generating the Sreedhar-Gao-Lee forest *requires the dominator tree*. This makes the Sreedhar-Gao-Lee forest inappropriate for the application of *generating the dominator tree*!

We introduce yet another loop nesting forest, one that fits into the same framework as the other three forests but does not posess any of the above mentioned disadvantages. Our two applications run in linear time if they utilize this new forest. The new forest can be generated in almost linear time, using a simple bottom up traversal of the depth-first search tree.

In conclusion, this paper (a) introduces a problem reduction strategy for transforming arbitrary graph problem instances into equivalent acyclic graph problem instances, a technique which may be useful in other applications, (b) improves our understanding of loops, (c) introduces a new, easy-to-construct, loop nesting forest that has some advantages over previous loop forests, and (d) presents new algorithms for constructing the dominator tree and the SSA form (iterated dominance frontiers). Though other algorithms are already known for these two problems, the new algorithms are valuable because they provide compiler implementors with more choices. For instance, if the loop forest and the SSA form are required in a compiler, the algorithms show how the two can be constructed *without constructing the dominator tree*. (The previous algorithms for constructing the SSA form require the dominator tree.) Eliminating the need for an intermediate data structure (i.e., the dominator tree) not only reduces the programming effort but may also improve compile-time space requirements (a possibly significant factor for dynamic and just-in-time compilers).

2. TERMINOLOGY AND NOTATION

A control-flow graph G is a directed graph with a distinguished *entry* vertex. We will denote the vertex set of G by $V(G)$ (or V), the edge set of G by $E(G)$ (or E), and the entry vertex by *entry*(G). For convenience, we assume that the entry vertex has no predecessors and that every vertex in the graph is reachable from the entry vertex.

If Y is a set of vertices in a graph G, let $\langle Y \rangle_G$ denote the subgraph of G induced by Y. Thus, $\langle Y \rangle_G$ consists of the set of vertices Y and the set of edges $E(G) \cap (Y \times Y)$. If no confusion is likely, we will omit the subscript G.

If α and β are two paths such that the last vertex of α is the first vertex of β, then we use $\alpha\beta$ to denote the path obtained by concatenating these two paths. α is said to be a *prefix* and β a *suffix* of $\alpha\beta$. Occasionally, when the last vertex u of α is a predecessor of the first vertex v of β, then we will abuse the notation and use $\alpha\beta$ to denote the path obtained by concatenating α, the edge $u \rightarrow v$ and β. Further, if the first vertex v of a path β is a successor of a vertex u, then we will use $u\beta$ as short hand notation for the concatenation of the two paths $[u, v]$ and β.

A set X of vertices is said to be *strongly connected* if there exists a path, consisting only of vertices in X, between any two vertices of X. Further, X is said to be a strongly connected *component* of the graph if X is strongly connected and no proper superset of X is strongly connected. (Thus, it is a maximal strongly connected set of vertices.) A strongly connected component is said to be *non-trivial* if it consists of two or more vertices or if it consists of one vertex that has a self-loop (an edge from the vertex to itself). Given a graph G, let $SCC(G)$ denote the set of non-trivial strongly connected components of G.

We say that a vertex x dominates a vertex y if every path from the entry vertex to y passes through x. We say that x strictly dominates y if x dominates y and $x \neq y$. Domination is a special kind of relation, one that can be concisely represented by a tree, called the dominator tree, such that x strictly dominates y iff x is a proper ancestor of y in the dominator tree. We refer to the parent of a vertex u in the dominator tree as u's immediate dominator.

The dominance frontier of a vertex x in a graph G, denoted $DF_G(x)$, is the set of all y such that x dominates some predecessor z of y but does not strictly dominate y. The dominance frontier of a set of vertices is defined to be the union of the dominance frontiers of its elements. Let X be a set of vertices. Define $DF_G^i(X)$ to be $DF_G(X)$ if $i = 1$ and $DF_G(X \cup DF_G^{i-1}(X))$ if $i > 1$. The limit of this sequence is the iterated dominance frontier of X, denoted $DF_G^+(X)$.

A vertex w is said to be a join node for two distinct vertices u and v, if there exist two non-null paths (i.e., paths with at least one edge) $\alpha : u \rightarrow^+ w$ and $\beta : v \rightarrow^+ w$ such that the only vertex common to the two paths is the vertex w (i.e., the two paths are disjoint except for vertex w). In such a case, the two paths are said to converge at w. Given a set of vertices X, we say that a vertex w is a join node for X if w is a join node for some $u, v \in X$, where $u \neq v$. The join set $J_G(X)$ is defined to be the set of all join nodes of X.

$$J_G(X) = \{w \mid \exists u, v \in X. u \neq v \text{ and } w \text{ is a join node for } u \text{ and } v\}$$

Define $J_G^i(X)$ to be $J_G(X)$ if $i = 1$ and $J_G(X \cup J_G^{i-1}(X))$ if $i > 1$. The limit of this sequence is called the iterated join set of X, denoted $J_G^+(X)$. We will drop the

subscript G if no confusion is likely.

For any set X that includes the entry vertex of the graph, Cytron *et al.* [1991] show that $J^+(X) = DF^+(X)$. Weiss [1992] establishes a stronger result, namely that $J(X) = DF^+(X)$.

The construction of the SSA form and Sparse Evaluation Graphs, both of which are widely used intermediate representations, reduces essentially to the problem of computing iterated dominance frontiers. We refer the reader to [Cytron et al. 1991; Choi et al. 1991] for further details.

3. MOTIVATION

In this section, we consider the problem of constructing the dominator tree of a graph and the problem of computing the iterated dominance frontier of a set of vertices in a graph. We describe simple, previously known, algorithms for solving these problems when the input graph is reducible. The questions that motivate the rest of the paper is: can we generalize these algorithms to handle irreducible graphs, and if so, how?[1]

3.1 Constructing the Dominator Tree

We first consider the problem of constructing the dominator tree of a control-flow graph. The algorithm described below appears in [Ramalingam and Reps 1994] and [Alstrup and Lauridsen 1996].

3.1.1 *Acyclic Graphs.* We begin by looking at acyclic graphs first. Let u be a vertex with predecessors $v_1, \cdots, v_k$. Note that a vertex w strictly dominates u iff w dominates all the vertices v_1 through v_k. In other words, w is a proper ancestor of u in the dominator tree iff w is an ancestor of v_1 through v_k in the dominator tree. This implies that the immediate dominator of u must be the least common ancestor (also referred to as the nearest common ancestor) of v_1 through v_k in the dominator tree.

This suggests the following algorithm for constructing the dominator tree of an acyclic control-flow graph, which builds the tree incrementally, processing the vertices in topological sort order. (We will consider the question of how to compute least common ancestors later on.)

function ConstructDominatorTree(G : Acyclic Graph)
  `DomTree` := an empty tree;
  add *entry*(G) as the root of `DomTree` ;
  for every vertex u in $V(G) - \{$ *entry*(G) $\}$ in topological sort order **do**
    let z denote the least-common-ancestor, in `DomTree`, of u's predecessors;
    add u as a child of z in `DomTree` ;
  end for
  return `DomTree`;

3.1.2 *Reducible Graphs.* An edge $u \to v$ is said to be a *backedge* if v dominates u. It can be shown that removing the backedges of a graph does not change the

[1]Note that other, completely independent, algorithms are already known for solving these problems over arbitrary graphs.

domination relation on vertices. In other words, the dominator tree of the graph obtained by removing the backedges is the same as the dominator tree of the original graph.

A control-flow graph is said to be *reducible* if the graph obtained by removing the backedges is acyclic. (There are several other equivalent characterizations of a reducible graph.)

Consequently, the dominator tree of a reducible graph can be constructed by first identifying and eliminating all its backedges and then applying the algorithm described above for acyclic graphs. (An edge in a reducible graph is a backedge iff it is a dfs-backedge, i.e., an edge from a vertex to one of its ancestors in the depth first search tree. Hence, the backedges of a reducible graph can be identified easily, in linear time, using depth first search.)

3.2 Computing the Iterated Dominance Frontier

We now consider the problem of constructing the iterated dominance frontier of a set of vertices. The algorithm presented below appeared in [Cytron et al. 1986], buried in the middle of an algorithm for code motion, but does not seem to be widely known.

3.2.1 *Acyclic Graphs.* Let G be an acyclic graph. The iterated dominance frontier of a set of vertices X in G is the same as the iterated join set $J_G^+(X \cup \{\ entry(G)\ \})$. Let us refer to elements of $X \cup \{\ entry(G)\ \} \cup J_G^+(X \cup \{\ entry(G)\ \})$ as "definition" nodes (borrowing terminology used in the context of the SSA form). Let us say that a definition node d "reaches" another node u if there is a path in the graph from d to u which does not contain any definition node other than d or u. Note that we assume that there is a path from $entry(G)$ to every vertex in the graph. Hence, at least one definition node must reach any vertex $u \neq entry(G)$. If at least two definitions reach a node u, then u must be a join node for $X \cup \{\ entry(G)\ \} \cup J_G^+(X \cup \{\ entry(G)\ \})$ and, hence, must be in $J_G^+(X \cup \{\ entry(G)\ \})$. In other words, vertices not in $J_G^+(X \cup \{\ entry(G)\ \})$ must have exactly one reaching definition.

This suggests the following algorithm, which visits vertices in topological sort order and computes their reaching definitions. Vertices with more than one reaching definition are added to the iterated dominance frontier. The computation is simplified by the fact that any vertex is either a definition node or has a unique reaching definition.

```
function ComputeIteratedDomFrontier(G: Acyclic Graph, X : Subset of V(G))
    ItDomFr := {};
    for every vertex u of V(G) − { entry(G) } in topological sort order do
        reachingDefs := {};
        for every predecessor v of u do
            if v ∈ (ItDomFr ∪ X ∪ {entry(G)})
                then add v to reachingDefs;
                else add uniqueReachingDef(v) to reachingDefs;
            end if
        end for
        if |reachingDefs| = 1
            then uniqueReachingDef(u) := the only element of reachingDefs;
```

```
        else add u to ItDomFr;
      end if
    end for
    return ItDomFr;
```

The above algorithm can be trivially adapted to construct the sparse evaluation graph with respect to a set of vertices X. In particular, the sparse evaluation graph consists of the set of vertices $X \cup DF^+(X)$. For every vertex u in the sparse evaluation graph, the set of predecessors is given by the set `reachingDefs` computed by the above algorithm. This algorithm runs in linear time, provided a suitable implementation of the various set operations is used.

3.2.2 *Reducible Flow Graphs.* Let us now consider a reducible graph G. Let us denote the acyclic graph obtained by the backedges of G by $\mathcal{F}(G)$. Our goal is to transform the problem of computing the iterated dominance frontier of a set of vertices X in G into a problem of identifying the iterated dominance frontier of a set of vertices in the acyclic graph $\mathcal{F}(G)$, and to account for the contribution of the backedges to the iterated dominance frontier in some way.

We can identify the contribution of backedges to the iterated dominance frontier of a set of vertices using the notion of loops. (For now, we assume that a "loop" has some obvious properties. We will formalize these assumptions later on in Section 4.) If a vertex x is contained in a loop, then the iterated dominance frontier of x will include the entry vertex of the loop. (A vertex is said to be an entry vertex of the loop if it has a predecessor outside the loop.) The reason for this is that, since a loop is strongly connected, there exists a path from x to the loop's entry vertex consisting only of vertices in the loop. On the other hand, there is a path from the entry vertex of the graph to the loop's entry vertex consisting only of vertices outside the loop. Hence, the loop's entry vertex is a join node for x and $entry(G)$.

For any vertex u, let $HLC(u)$ denote the entries of the set of loops containing vertex u. Given a set of vertices X, define $HLC(X)$ to be $\cup_{u \in X} HLC(u)$.

Clearly, the iterated dominance frontier of a set X includes $HLC(X)$. It turns out that (in a reducible graph) this precisely identifies the contribution of backedges to the iterated dominance frontier. In particular, we can show that:

THEOREM 1. $DF^+_G(X) = HLC(X) \cup DF^+_{\mathcal{F}(G)}(X \cup HLC(X))$.

(We do not present a proof of the above theorem here as we will be proving a generalized version of the theorem for arbitrary graphs later on.)

This shows how the algorithm outlined earlier for acyclic graphs can be utilized to compute $DF^+_G(X)$ using a "loop nesting forest'. Consider the control-flow graph shown in Figure 1(a). There are three loops in this graph, one with entry vertex a, one with entry vertex c and one with entry vertex j. The loops and the vertices they contain can be represented by a "loop nesting forest", as shown in Figure 1(b). The internal vertices of the forest, shown as ellipses, denote loops, with the loop's entry vertex shown inside the ellipse. The leaves of the forest, shown as rectangles, identify vertices in the control-flow graph. The ancestor-descendant relation in this forest captures the loop containment relation.

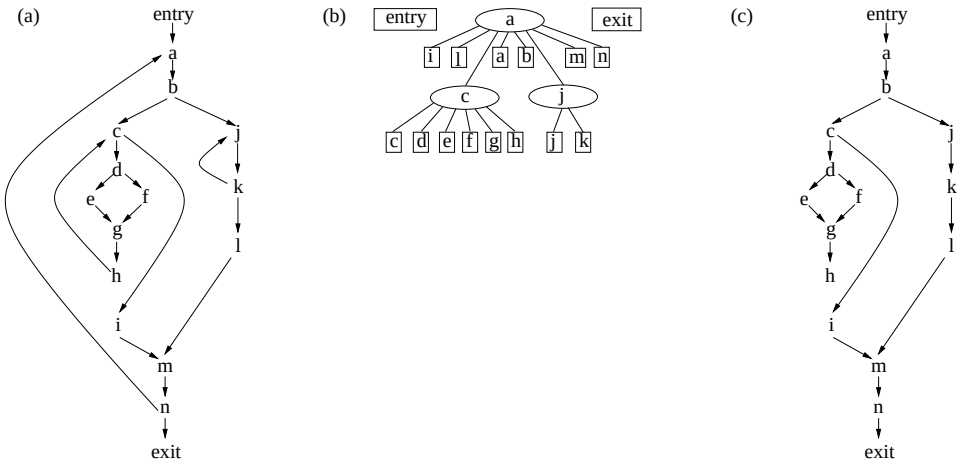

Fig. 1. (a) A reducible control-flow graph G. (b) Its loop nesting forest. (c) The acyclic graph obtained by dropping the backedges of G.

The set of loops containing any given vertex is given by the ancestors of the vertex in the loop nesting forest. Given a vertex x, $HLC(x)$ can be determined efficiently by "walking up" this forest, starting from x. Given a set of vertices X, $HLC(X)$ can be identified by performing an upward traversal of the loop nesting forest starting from all vertices in X. This takes time proportional to the size of $HLC(X)$. Once we determine $HLC(X)$, we can identify the iterated dominance frontier of $X \cup HLC(X)$ in the acyclic graph $\mathcal{F}(G)$ using the algorithm presented in the previous section. Theorem 1 then yields us the iterated dominance frontier of X in the reducible graph G.

As an example, let us consider the computation of the iterated dominance frontier of the vertex e in the reducible graph shown in Figure 1(a). The set $HLC(e)$ is $\{c, a\}$, as can be identified by "walking up" the forest shown in Figure 1(b) from the leaf labelled e. Figure 1(c) shows the acyclic graph $\mathcal{F}(G)$ obtained by dropping all the backedges in the graph G. To identify the iterated dominance frontier of vertex e in G, we first identify the iterated dominance frontier of $\{e\} \cup HLC(e)$ $= \{e, a, c\}$ in $\mathcal{F}(G)$. If we apply our earlier algorithm to the acyclic graph, we can determine that the iterated dominance frontier of $\{e, a, c\}$ in $\mathcal{F}(G)$ is $\{g, m\}$. Hence, the iterated dominance frontier of vertex e in G is given by $HLC(e) \cup \{g, m\}$ $= \{c, a, g, m\}$.

Since the loop nesting forest of a reducible graph can be constructed in almost linear time [Tarjan 1974], this gives us a simple almost linear time for computing the iterated dominance frontier of a set of vertices in a reducible graph.

4. LOOPS IN IRREDUCIBLE GRAPHS

How can we generalize the algorithms described in the previous section to handle irreducible graphs? Before we can answer this question, we need to understand what the loops in an irreducible graph are.

The classical algorithm for identifying loops is Tarjan's interval finding algorithm

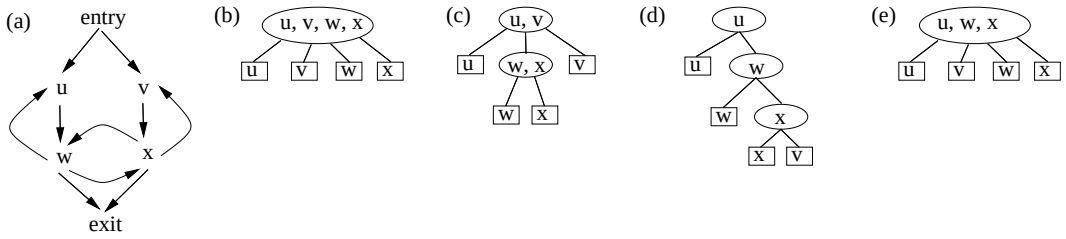

Fig. 2. An example illustrating different loop nesting forests for a single control-flow graph, shown in (a). Ellipses denote loops, with the loop's headers shown inside the ellipse. Rectangles identify vertices in the control-flow graph. The ancestor-descendant relation in this forest captures the loop containment relation. (b) The Sreedhar-Gao-Lee forest. (c) Steensgaard's forest. (d) Havlak's forest. (e) Reduced Havlak's forest.

[Tarjan 1974], which is restricted to reducible graphs. Recently, several algorithms have been proposed for identifying loops in arbitrary graphs. For certain irreducible graphs, each algorithm identifies a different set of loops. Consider the control-flow graph shown in Figure 2(a). In this example, the Sreedhar-Gao-Lee algorithm [Sreedhar et al. 1996] identifies a single loop $\{ u, v, w, x \}$, while Steensgaard's algorithm [Steensgaard 1993] identifies two loops $\{ u, v, w, x \}$ and $\{ w, x \}$, while Havlak's algorithm [Havlak 1997] identifies three loops $\{ u, v, w, x \}$, $\{ w, x, v \}$ and $\{ x, v \}$. (The loops identified by Havlak's algorithm actually depend on the order in which vertices are visited during depth-first search; in the above example, we assume that vertex u is visited before v.) The notion of a loop's headers, which are depicted in the figure, will be discussed later.

Which, if any, of these loop nesting forests can we use to generalize the algorithms of Section 3 to irreducible graphs? Though it is conceptually simpler to work with Steensgaard's definition for this purpose, it turns out that the generalization can be made to work with any of these three forests. We will now present an axiomatic definition of a loop nesting forest that is satisfied by the forest constructed by each of these three algorithms. We will subsequently show how the algorithms of Section 3 can be generalized to work for irreducible graphs using any loop nesting forest that satisfies these axioms.

4.1 Loop Nesting Forests: An Axiomatic Characterization

A loop nesting forest for a graph is essentially the set of all loops in the graph. At first blush, it seems reasonable to consider a loop to be a set of vertices (namely, the loop's "body") that posesses certain special properties. Loops also have special vertices, called "headers". The different definitions mentioned above may associate different "headers" with the same loop "body". We will, hence, consider a loop in a graph G to be a pair (B, H) of non-empty sets of vertices B and H, with $H \subseteq B$, where B is the *body* of the loop and H is the set of *headers* of the loop. Thus, a loop is an element of $2^{V(G)} \times 2^{V(G)}$. (Note that $2^{V(G)}$, the powerset of $V(G)$, denotes the set of all subsets of $V(G)$.) A loop nesting forest itself is a set of loops. Thus, a loop nesting forest is a subset of $2^{V(G)} \times 2^{V(G)}$.

Example. Consider the control-flow graph shown in Figure 2(a). We can more precisely describe the sets of loops identified by the three different algorithms as

follows. The Sreedhar-Gao-Lee algorithm [Sreedhar et al. 1996] identifies a single loop ($\{ u, v, w, x \}, \{ u, v, w, x \}$); Steensgaard's algorithm [Steensgaard 1993] identifies two loops ($\{ u, v, w, x \}, \{ u, v \}$) and ($\{ w, x \}, \{ w, x \}$); Havlak's algorithm [Havlak 1997] identifies three loops ($\{ u, v, w, x \}, \{ u \}$), ($\{ w, x, v \}, \{ w \}$) and ($\{ x, v \}, \{ x \}$).

It will sometimes be more convenient to represent a loop nesting forest as a pair $\langle \mathcal{B}, \mathcal{H} \rangle$, where $\mathcal{B} \subseteq 2^{V(G)}$ is the set of all loop bodies, and $\mathcal{H} \in \mathcal{B} \rightarrow 2^{V(G)}$ is a function that maps each loop body to the set consisting of its loop headers. More precisely, we use $\langle \mathcal{B}, \mathcal{H} \rangle$ as an alternative representation for the set $\{(B, \mathcal{H}(B)) \mid B \in \mathcal{B}\}$.

Clearly, not every pair $\langle \mathcal{B}, \mathcal{H} \rangle$ of the above form is a meaningful loop nesting forest for G. We now describe some properties we expect a loop nesting forest to satisfy.

First we concentrate on the loop body. The first, and obvious, property we expect of loops is that the body of a loop should be strongly connected.

The second property we assume of the loop nesting forest is that it has the *proper nesting* property: a set of loops is said to have the *proper nesting* property if for any two loops in the set, either the loop bodies of the two loops are mutually disjoint or the loop body of one is contained within the loop body of the other. Note that a set of loops with this property can be represented compactly by a forest over the set of loops and vertices, where a loop L_x is a descendant of another loop L_y iff the body of L_x is contained in the body of L_y and a vertex u is a descendant of a loop L_y iff u is contained in the body of L_y. We will usually use the term "loop nesting forest" to refer to a collection of loops satisfying the various properties described here, but occasionally we will use the term to denote the forest that represents such a collection (as explained above). In particular, it should be clear what it means for one loop to be the "parent" of another loop.

What exactly is a *header* of a loop? It turns out that the various definitions of loop nesting forests differ primarily in their notion of what a loop header is. Rather than choose a specific notion of a loop header, we just assume that a vertex that is dominated by some other vertex in the loop cannot be selected as a header of the loop. Given a set of vertices X, let UnDominated(X) denote the set of vertices in X that are not dominated by any other vertex in X. Then, for any loop (B, H) we assume that $H \subseteq \text{UnDominated}(B)$.

Finally, we expect the set of loops to be "complete" in some sense (i.e., account for all cycles or strongly connected sets in the graph). We say that a pair (B, H) *covers* a set X if $B \supseteq X$ and $X \cap H \neq \phi$. We will require that every non-trivial strongly connected set in the given graph be covered by some loop in the forest. The idea is that if a loop (B, H) covers a strongly connected set X, then (B, H) can be viewed as a representative of the strongly connected set X. Theorem 2 will partially illustrate in what sense this condition implies "completeness". Note that this requirement links the notion of a loop header to the set of loops that are selected to make up the loop nesting forest. Choosing a different notion of a loop header might force the selection of additional loops to satisfy this requirement.

We combine these assumptions into the following definition:

DEFINITION 1. *A pair* $\langle \mathcal{B}, \mathcal{H} \rangle$*, where* $\mathcal{B} \subseteq 2^{V(G)}$ *and* $\mathcal{H} \in \mathcal{B} \rightarrow 2^{V(G)}$*, is said to be a loop nesting forest for* G *if:*

(a) $\forall B \in \mathcal{B}$. B *is a non-trivial strongly connected set.*

(b) Every non-trivial strongly connected set X *in the graph is covered by* $(B, \mathcal{H}(B))$ *for some* $B \in \mathcal{B}$*.*

(c) The set of loops have the proper nesting property: $\forall B_1 \in \mathcal{B}. \forall B_2 \in \mathcal{B}.\ B_1 \cap B_2 = \phi$ *or* $B_1 \subseteq B_2$ *or* $B_2 \subseteq B_1$*.*

(d) A header of a loop is not dominated by any other vertex in the loop: $\forall B \in \mathcal{B}.\ \phi \neq \mathcal{H}(B) \subseteq \text{UnDominated}(B)$*.*

Let $\mathcal{L}$ be a loop nesting forest for a graph G. We refer to an edge from some vertex in a loop (body) to one of its headers as a *loopback edge* of the loop and the graph. (The term *back edge* has at least two different, but related, meanings, one in the context of depth first search, and another in the context of domination. The concept of loopback edge, too, is related to but different from these concepts. We use the name "loopback edge" to avoid any possible confusion.) Let $\mathcal{F}_{\mathcal{L}}(G)$ denote the graph obtained by removing all the loopback edges of G.

THEOREM 2. *If* $\mathcal{L}$ *is a loop nesting forest for* G *then,* $\mathcal{F}_{\mathcal{L}}(G)$ *is an acyclic graph.*

PROOF. Consider any cycle C in G. Since C is a non-trivial strongly connected set, it must be covered by some loop $(B, H) \in \mathcal{L}$ (condition (b) of Definition 1). This implies that C contains some header $h \in H$. Since $C \subseteq B$, the incoming edge of vertex h in cycle C is a loopback edge (by definition). This implies that every cycle C in graph G must contain at least one loopback edge, and the theorem follows. □

Thus, a loop nesting forest lets us decompose a graph into an acyclic graph and a set of loopback edges. The definition actually implies a stronger property, which is the basis of the algorithms we present later and will become clearer when we discuss the algorithms. The above theorem and its proof, however, partially illustrate the motivation behind the definition of a "covering", and explains the sense in which condition (b) of Definition 1 ensures that all cycles (and strongly connected sets) in the graph are represented by the loop nesting forest.

DEFINITION 2. *A loop nesting forest is said to be a* minimal *loop nesting forest if no loop body in the forest is covered by another loop in the forest.*

The above definition is equivalent to requiring that every non-trivial strongly connected set in the graph be covered by *exactly* one loop in the loop nesting forest. The definition also has a simple interpretation in terms of "headers". It essentially implies that a "header" of a loop L should not be contained in any inner loop of L.

The adjective "minimal" used in the above definition is meant to be suggestive, but it is important not to be misled by the word. The minimality condition does *not* apply to *the set of loop bodies*. In other words, a loop nesting forest $\langle \mathcal{B}, \mathcal{H} \rangle$ may be minimal even though there exists another loop nesting forest $\langle \mathcal{B}', \mathcal{H}' \rangle$ such that $\mathcal{B}'$ is a proper subset of $\mathcal{B}$. As an example, we will later see that the forests constructed by Steensgaard's algorithm, the Sreedhar-Gao-Lee algorithm, and Havlak's algorithm are all minimal loop nesting forests according to the above definition, even though

the set of loop bodies in Havlak's forest can be a superset of the set of loop bodies in Steensgaard's forest, which can itself be a superset of the set of loop bodies in the Sreedhar-Gao-Lee forest.

It is important to note that the definition of a loop nesting forest presented above is not intended to be "complete". The above definition was partially guided by the applications we are interested in. As we will see later, the definition of a loop nesting forest is general enough to include various specific forests that have been previously defined, and the definition of a minimal loop nesting forest is strong enough our intended application and to establish several interesting properties. However, other specific applications of loop nesting forests may require stronger properties than those implied by the above definitions.

4.2 Loop Nesting Forests: A Constructive Characterization

We now present a constructive scheme for generating a family of loop nesting forests. The following definition generalizes definitions due to Steensgaard [1993] and Havlak [1997], by abstracting the header function.

Consider any function $\mathcal{H} : 2^{V(G)} \rightarrow 2^{V(G)}$. We will show that this identifies a collection of loops, with $\mathcal{H}$ as its header function, satisfying Definition 1, as long as $\mathcal{H}$ satisfies a simple property.

Consider $SCC(G)$, the collection of non-trivial strongly connected components of G. We identify elements of $SCC(G)$ as the outermost loops of G. Consider any X in $SCC(G)$. The set $\mathcal{H}(X)$ is the set of headers for X. Recall that an edge from a vertex inside X to one of its headers is referred to as a *loopback edge* of X. Let $\mathcal{F}^1_{\mathcal{H}}(G)$ denote the graph obtained by dropping the loopback edges of the outermost loops of G (*i.e.*, $SCC(G)$). For $i > 1$, define $\mathcal{F}^i_{\mathcal{H}}(G)$ to be $\mathcal{F}^1_{\mathcal{H}}(\mathcal{F}^{i-1}_{\mathcal{H}}(G))$. For the sake of convenience, let $\mathcal{F}^0_{\mathcal{H}}(G)$ denote the original graph G.

DEFINITION 3. $\text{Loops}_{\mathcal{H}}(G) = \bigcup_{i \geq 0} SCC(\mathcal{F}^i_{\mathcal{H}}(G))$

This set of loop bodies, along with function $\mathcal{H}$, identifies a loop nesting forest:

DEFINITION 4. $\text{LoopForest}_{\mathcal{H}}(G) = \langle \text{Loops}_{\mathcal{H}}(G), \mathcal{H} \rangle$

THEOREM 3. *If* $\mathcal{H} \in 2^{V(G)} \rightarrow 2^{V(G)}$ *is such that for every* $X \in \text{Loops}_{\mathcal{H}}(G)$, $\phi \neq \mathcal{H}(X) \subseteq \text{UnDominated}(X)$, *then* $\text{LoopForest}_{\mathcal{H}}(G)$ *is a minimal loop nesting forest for* G.

PROOF. We first show that $\text{LoopForest}_{\mathcal{H}}(G)$ satisfies all the four conditions of Definition 1.

(a) Every $B \in \text{Loops}_{\mathcal{H}}(G)$ is a non-trivial strongly connected set in some $\mathcal{F}^i_{\mathcal{H}}(G)$. But every $\mathcal{F}^i_{\mathcal{H}}(G)$ is a subgraph of G. Hence, every $B \in \text{Loops}_{\mathcal{H}}(G)$ is a non-trivial strongly connected set in G.

(b) Let X be any non-trivial strongly connected set of vertices in G. Consider the maximum i such that X is a non-trivial strongly connected set in $\mathcal{F}^i_{\mathcal{H}}(G)$. Then, X must be contained in some strongly connected component B of $\mathcal{F}^i_{\mathcal{H}}(G)$. Further, X must contain at least one of B's headers. Otherwise, X would be a non-trivial strongly connected set in $\mathcal{F}^{i+1}_{\mathcal{H}}(G)$ also. Hence, $(B, \mathcal{H}(B))$ covers X.

(c) Note that any two distinct elements of $SCC(\mathcal{F}^i_{\mathcal{H}}(G))$ are disjoint. Further, if $i < j$ then for every element Y of $SCC(\mathcal{F}^j_{\mathcal{H}}(G))$ there exists an element Y' of

$SCC(\mathcal{F}^i_{\mathcal{H}}(G))$ such that Y' is a proper superset of Y. Hence, the set $\text{Loops}_{\mathcal{H}}(G)$ clearly has the proper nesting property.

(d) This follows directly from our assumption about $\mathcal{H}$.

It follows that $\text{LoopForest}_{\mathcal{H}}(G)$ is a loop nesting forest. The proof of minimality is similar to the proof of (b). Let X be any non-trivial strongly connected set of vertices in G. Consider the minimum j such that X is covered by some strongly connected component B of $\mathcal{F}^j_{\mathcal{H}}(G)$. By definition, X must contain at least one of B's headers, say h. Note that all loopback edges directed to h are dropped in generating $\mathcal{F}^{j+1}_{\mathcal{H}}(G)$. Hence, no loop identified in $\mathcal{F}^k_{\mathcal{H}}(G)$, for any $k > j$, can contain h or cover X. Hence, X is covered by exactly one loop in $\text{LoopForest}_{\mathcal{H}}(G)$. Hence, $\text{LoopForest}_{\mathcal{H}}(G)$ is a minimal loop nesting forest.

□

Conversely, every minimal loop nesting forest can be generated using the above scheme. Note that the above scheme can, in fact, be used even if $\mathcal{H}$ is a partial function, as long as $\mathcal{H}$ is defined for the necessary elements, namely the elements of $\text{Loops}_{\mathcal{H}}(G)$. We can show that if $\mathcal{L} = \langle \mathcal{B}', \mathcal{H}' \rangle$ is a minimal loop nesting forest for G, then $\mathcal{B}'$ must be $\text{Loops}_{\mathcal{H}'}(G)$. Firstly note that $\mathcal{L}$ must, in fact, include all elements of $SCC(G)$ as "loop bodies" to ensure that these elements are "covered". Secondly note that if (B, H) is a loop in $\mathcal{L}$, then $\mathcal{L}$ must also include all elements of $SCC(\langle B - H \rangle)$ as "loop bodies" to ensure that these elements are "covered". It follows by simple induction that $\text{Loops}_{\mathcal{H}'}(G)$ is well-defined (even though $\mathcal{H}'$ is a partial function) and that $\mathcal{B}' \supseteq \text{Loops}_{\mathcal{H}'}(G)$. Since $\text{Loops}_{\mathcal{H}'}(G)$ covers all strongly connected sets in G (as shown in the proof of Theorem 3) and since $\mathcal{L}$ is a minimal loop nesting forest, we must have $\mathcal{B}' = \text{Loops}_{\mathcal{H}'}(G)$.

Let us now look at the "forest" representation of a set of loops $\mathcal{L}$ generated as above. Consider any loop L. Let G_L denote the subgraph induced by the vertices in L, but without the loopback edges of L. The "children" of L in the "forest" representation are the strongly connected components of G_L. The non-trivial strongly connected components of G_L denote inner loops (which become internal nodes in the "forest" representation), while the trivial strongly connected components of G_L denote vertices belonging to L but not to any inner loop of L, and these become "leaves" of the "forest".

Clearly, the children of L can be ordered in "topological sort order" such that there is an edge from (a vertex belonging to) one child C_1 to (a vertex belonging to) another child C_2 only if C_1 comes before C_2 in the ordering. We will refer to a loop nesting forest with children ordered this way as an *ordered loop nesting forest*.

THEOREM 4. *Let $\mathcal{L}$ be a minimal loop nesting forest for G. There exists a topological sort ordering of the vertices in $\mathcal{F}_{\mathcal{L}}(G)$ in which vertices belonging to any loop appear consecutively.*

PROOF. Consider an "ordered forest" representation of $\mathcal{L}$. The leaves of this ordered forest, enumerated in order, satisfies the requirement. □

4.3 Previously Defined Loop Nesting Forests

We now show how the three *specific* loop nesting forests mentioned previously can be obtained as special cases of our parametric definition (in Section 4.2) by choosing

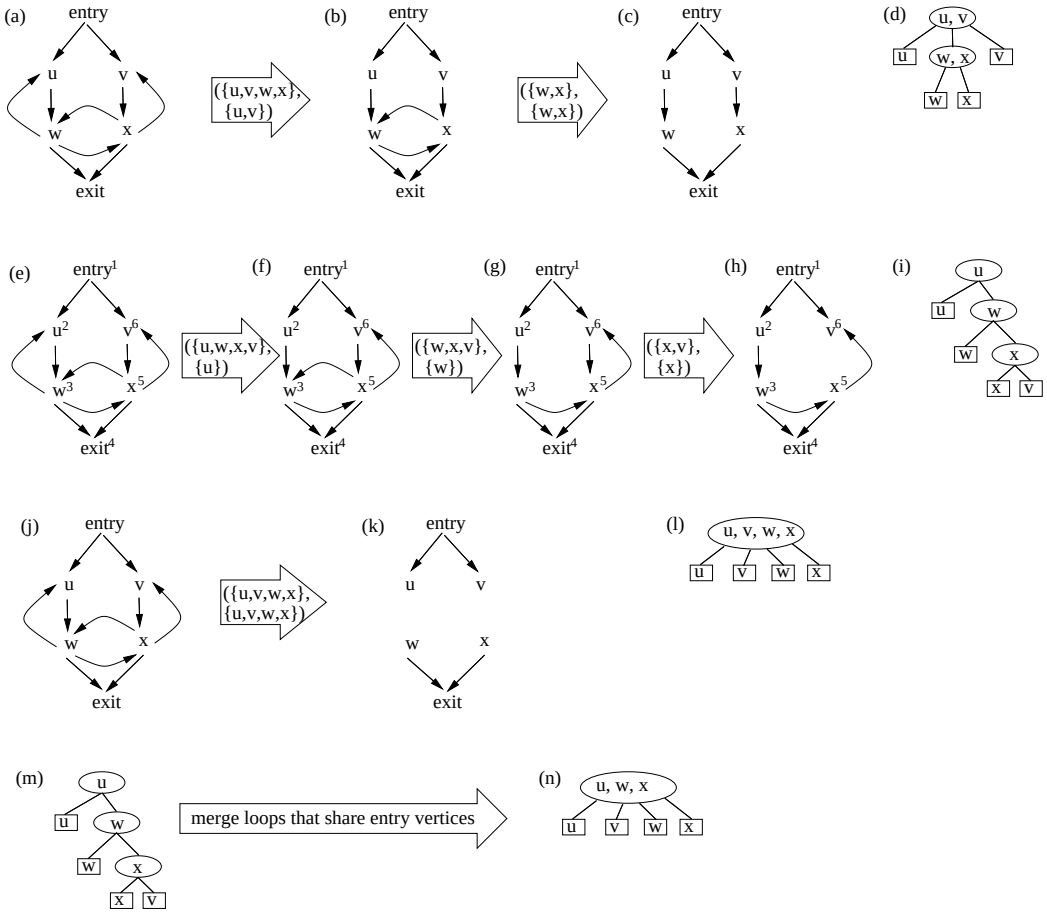

Fig. 3. An example illustrating different loop nesting forests for a single control-flow graph, shown in (a). Steps (a)-(c) illustrate the generation of Steensgaard's forest, using the scheme presented in Section 4.2. The arrows indicate the loop identified in every step, the removal of whose loopback edges yields the subgraph shown in the next step. The corresponding forest is shown in (d). Steps (e)-h) illustrate the generation of Havlak's forest, which is shown in (i). Steps (j)-(k) illustrate the generation of the Sreedhar-Gao-Lee forest, which is shown in (k). Steps (l)-(m) illustrate the generation of the Reduced Havlak forest by merging loops that share a common entry vertex.

suitable header functions $\mathcal{H}$.

Given a set of vertices X, a vertex in X is said to be an *entry* vertex of X if it has a predecessor outside X. Steensgaard's forest [Steensgaard 1993] is the forest obtained from our parametric definition by defining $\mathcal{H}(X)$ to be the set of entry vertices of X.

Consider the graph in Figure 3(a). Identifying the non-trivial SCCs of this graph yields one outer loop $\{u, v, w, x\}$. The headers of this loop are its entry vertices u and v. Dropping the loopback edges of the loop yields the graph in Figure 3(b). Identifying the non-trivial SCCs of this graph yields one inner loop $\{w, x\}$. The headers of this loop are its entry vertices w and x. Dropping the loopback edges

of this loop yields the graph in Figure 3(c). As this graph is acyclic, the process halts, yielding the loop nesting forest in Figure 3(d). The internal vertices of the forest, shown as ellipses, denote the loops in the program. The headers of a loop are shown inside the ellipse. The leaves of the forest, shown as rectangles, identify vertices in the control-flow graph.

The Havlak forest is the one generated by the scheme described in Section 4.2 when we define $\mathcal{H}(X)$ to be the singleton set consisting of the vertex in X that is visited first during depth first search (DFS). (Obviously, the loops identified by Havlak's algorithm [Havlak 1997] depend on the order in which vertices are visited DFS.)

Consider the graph in Figure 3(e). Assume that the vertices are visited in the order *entry, u, w, exit, x, v* during depth first search. (The superscripts attached to the vertices in the figure indicate the vertices' depth first numbering.) The graphs in Figure 3(f)-(h) illustrate the generation of the Havlak forest, which consists of three loops: an outermost loop $\{u, v, w, x\}$, an intermediate loop $\{w, x, v\}$, and an innermost loop $\{x, v\}$. The resulting loop nesting forest is shown in Figure 3(i).

The definition presented in Section 4.2 is constructive, but it is not the most efficient way to construct Havlak's forest. Havlak's forest can be constructed in almost linear time using a bottom up traversal of the DFS tree [Havlak 1997; Ramalingam 1999].

Sreedhar, Gao and Lee [1996] outline an algorithm, based on a bottom up traversal of the dominator tree, for identifying the loops in a graph. The set of primary loops identified by the Sreedhar, Gao, and Lee algorithm can be shown to be the same as the loop nesting forest generated by the scheme of Section 4.2 when we define $\mathcal{H}(X)$ to be the set of vertices in X that are not strictly dominated by any vertex in X.

The Sreedhar-Gao-Lee forest of the graph in Figure 3(j) consists of only one loop, namely $\{u, v, w, x\}$, and is shown in Figure 3(l). The Sreedhar, Gao, and Lee algorithm has a worst case quadratic complexity but can be improved to run in almost linear time [Ramalingam 1999].

It follows from Theorem 3 that all three of these forests are minimal loop nesting forests, as established in the appendix. Both Havlak as well as Sreedhar-Gao-Lee outline extensions to their respective algorithms that construct a more "refined" forest with more loops. It can be shown that these refined forests (constructed by both extensions) are loop nesting forests according to our definition (i.e. they satisfy Definition 1) but are not minimal loop nesting forests. See the appendix for a more complete description of all these previously defined loop nesting forests.

4.4 A New Loop Nesting Forest

As explained earlier, our goal is to explore the use of loop nesting forests in solving two problems, namely the computation of the iterated dominance frontier of a set of vertices and the construction of the dominator tree. The approach we take is to generalize simple algorithms that are restricted to acyclic graphs by showing how a problem instance involving an arbitrary graph can be transformed into equivalent problem instance over an acyclic graph. We will later show that such a transformation is feasible given *any* minimal loop nesting forest for the given graph.

Though our approach can be used with any minimal loop nesting forest in prin-

ciple, not all minimal loop nesting forests are equally appropriate for use from the point of view of practicality and efficiency. In fact, all the three forests we have looked at so far turn out to be unattractive for the applications we consider.

The Steensgaard's loop nesting forest is unattractive because the best known algorithm for generating this forest takes quadratic time in the worst case. On the other hand, both Havlak's forest and the Sreedhar-Gao-Lee forest can be generated in almost linear time. The algorithm for constructing the Sreedhar-Gao-Lee forest, however, requires the dominator tree. The first application we consider later on is the construction of the dominator tree itself, and, hence, using the Sreedhar-Gao-Lee forest is meaningless in the context of that application.

The algorithm for generating Havlak's forest is simple and based on depth-first search. Thus, it seems like an ideal candidate for our approach. Unfortunately, it turns out that the size of the transformed acyclic graph generated by our approach can be quadratic in the size of the initial graph (in the worst case) if we use Havlak's forest to perform the transformation. (In contrast, the size of the transformed graph is at most twice the size of the initial graph if we use either Steensgaard's forest or the Sreedhar-Gao-Lee forest to perform the transformation.)

We will now show how we can construct a smaller version of Havlak's loop nesting forest that eliminates this disadvantage of Havlak's forest, by merging loops that have a common entry vertex. (Recall that a vertex in a loop is said to be an entry vertex of that loop if it has a predecessor outside the loop.)

LEMMA 1. *Let $\mathcal{L}$ be a minimal loop nesting forest for G. Let (B_p, H_p) and (B_c, H_c) be two loops in $\mathcal{L}$ such that (B_p, H_p) is the parent of (B_c, H_c) in $\mathcal{L}$. If $H_c \subseteq \text{UnDominated}(B_p)$, then $\mathcal{L} - \{(B_p, H_p), (B_c, H_c)\} \cup \{(B_p, H_p \cup H_c)\}$ is another minimal loop nesting forest for G.*

PROOF. Straight forward. Note that the lemma says that if a loop's headers are not dominated by any vertex in it's parents loop body, then it can be "merged" with its parent (by eliminating the loop and adding its headers to it's parent's headers). □

Our goal is to construct a smaller version of Havlak's forest by eliminating some loops in the forest using the above lemma. However, rather than merge any two loops that satisfy the conditions of the above lemma we will only merge loops that share a common entry vertex.

Let $\cong$ denote the smallest equivalence relation on the loops of Havlak's forest such that $l_1 \cong l_2$ for any two loops l_1 and l_2 that have a common entry vertex. The union of two loops (B_1, H_1) and (B_2, H_2) is defined to be $(B_1 \cup B_2, H_1 \cup H_2)$. Let Γ denote the set of equivalence classes of Havlak loops. The Reduced Havlak Forest is defined to be $\{\bigcup_{l \in \gamma} l \mid \gamma \in \Gamma\}$.

THEOREM 5. *The Reduced Havlak Forest is a minimal loop nesting forest.*

PROOF. First note that if two loops l_1 and l_2 have a common entry vertex, then all loops that lie in-between l_1 and l_2 in the loop nesting forest must also have the same common entry vertex. Now, let (B_p, H_p) and (B_c, H_c) be two loops that share a common entry vertex u, and let (B_p, H_p) be the parent of (B_c, H_c). Then, none of the vertices in B_c are dominated by any of the vertices in $B_p - B_c$ (since the entry vertex u provides a path to vertices in B_c that does not pass through any vertex

in $B_p - B_c$). Consequently, the Reduced Havlak Forest can be generated from the Havlak Forest by repeatedly merging adjacent loops that satisfy the conditions of Lemma 1. The result follows from Lemma 1. □

The example in Figure 3(m)-(n) illustrates the above transformation. Havlak's loop nesting forest is shown in Figure 3(m), and the Reduced Havlak's forest is shown in Figure 3(n). In this example, all the three loops in Havlak's forest share a common entry vertex, namely v. Note that the collapsed forest contains the same set of loop bodies as the Sreedhar-Gao-Lee forest but has a different set of headers. (This does not hold true in general.)

The algorithm outlined by Ramalingam [1999] for generating Havlak's Forest already computes the equivalence relation $\cong$ and can be adapted to construct the Reduced Havlak Forest in almost linear time. Details of how this can be done appear in the appendix.

5. A GRAPH TRANSFORMATION

We now return to the question of how to generalize the algorithms described in Section 3 to handle irreducible graphs.

We first consider the problem of constructing the dominator tree. The difficulty in directly applying the algorithm described in Section 3 to irreducible graphs is that dropping the backedges of an irreducible graph does not yield an acyclic graph, even though it does yield a graph with the same dominator tree as the original graph. (Recall that a backedge is an edge $u \rightarrow v$ where v dominates u.)

We saw earlier (Theorem 2) that a loop nesting forest $\mathcal{L}$ of a graph G identifies a set of edges, the loopback edges, whose deletion does yield an acyclic graph $\mathcal{F}_{\mathcal{L}}(G)$. However, in the presence of irreducibility, the dominator tree for the resulting graph may not be the same as the dominator tree of the original graph.

Consider the example in Figure 4. The graph G in Figure 4(a) is the prototypical irreducible graph and has no backedges. Both Steensgaard's approach and the Sreedhar-Gao-Lee approach produce the same loop nesting forest in this simple example, which is shown in Figure 4(b). The forest consists of a single loop $\{v, w\}$ with two loop headers v and w. (Havlak's approach also identifies the same single loop body, but associates it with a single header, which depends on the depth-first search order.) The acyclic subgraph $\mathcal{F}_{\mathcal{L}}(G)$ obtained by eliminating the loopback edges is shown in Figure 4(c). Note that u dominates v in $\mathcal{F}_{\mathcal{L}}(G)$, but not in G. Thus, the dominator tree of $\mathcal{F}_{\mathcal{L}}(G)$ is different from the dominator tree of G. The same holds true even if we use Havlak's loop nesting forest. In general, the graph $\mathcal{F}_{\mathcal{L}}(G)$ may even contain vertices unreachable from the entry vertex of the graph, as illustrated by the example in Figure 3(j).

What we would like to have is a transformation that converts a possibly irreducible graph into an acyclic graph that has the same dominator tree as the original graph. We achieve this by deleting the loopback edges, while simultaneously adding some other edges that compensate for the deleted edges without creating cycles in the graph. Let us refer to a vertex outside a loop as a pre-entry vertex for that loop if it has a successor inside the loop. Let p be a pre-entry vertex of a loop L. When we delete a loopback edge $u \rightarrow h$ of the loop L, we potentially lose information about the presence of a path from p to the header h through the vertices in the

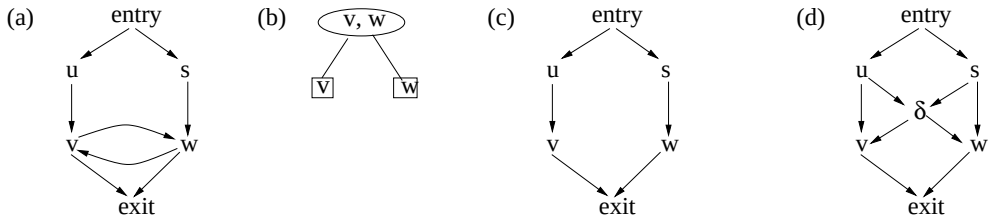

Fig. 4. (a) An irreducible graph G. (b) A loop nesting forest $\mathcal{L}$ for G. (c) The acyclic subgraph $\mathcal{F}_{\mathcal{L}}(G)$. (d) The transformed graph $\Psi_{\mathcal{L}}(G)$.

loop. We can compensate for this by adding edges from every pre-entry vertex of the loop to every header of the loop.

Such a transformation does work: if we apply the transformation to every loop in the graph, we do end up with an acyclic graph that has the same dominator tree as the original graph.

A direct implementation of this transformation can lead to a quadratic number of edges being added to the graph in the worst case, *e.g.* when a loop has $\theta(n)$ headers and $\theta(n)$ pre-entry vertices. To prevent such a potentially quadratic blowup in the number of edges, we use a modified transformation that achieves the same effect. We create a dummy header vertex δ_L for the loop, and add edges from every pre-entry vertex of the loop to the dummy header, and edges from the dummy header to every real header of the loop.

Consider the example shown in Figure 4. The headers of the only loop in this graph are v and w. The pre-entry vertices of this loop are u and s. Hence, our transformation deletes the loopback edges $w \to v$ and $v \to w$, but compensates for them by adding a new vertex δ and the new edges $u \to \delta$, $s \to \delta$, $\delta \to v$, and $\delta \to w$. This gives us the acyclic graph in Figure 4(d). It can be verified that this acyclic graph has the same dominator tree as the original graph in Figure 4(a).

The same transformation also helps in the computation of iterated dominance frontiers. Consider Theorem 1, which was the basis for the algorithm for computing iterated dominance frontiers in a reducible graphs. The theorem implies that dropping the backedges does not change the iterated dominance frontier of vertices that are not contained in any loop. This, however, is not true if we drop the loopback edges of an irreducible graph.

Consider the example shown in Figure 4(a) again. The iterated dominance frontier of the node u in the example of Figure 4(a) includes vertices v and w, thanks to the loopback edges $v \to w$ and $w \to v$. If we drop the loopback edges $w \to v$ and $v \to w$, we get the acyclic graph $\mathcal{F}_{\mathcal{L}}(G)$ shown in Figure 4(c). The iterated dominance frontier of u in this acyclic graph *does not include* v and w!

However, it can be verified that the iterated dominance frontier of u in the transformed acyclic graph in Figure 4(d). does include both v and w.

The transformation described above, for a single loop, is adequate for the problem of constructing the dominator tree. It is also adequate for computing iterated dominance frontiers, if we use, say, Steensgaard's loop nesting forest. However, if the loop contains header vertices that are not entry vertices of the loop (the Sreedhar-Gao-Lee loop nesting forest may contain such loops, for example), the

transformation will end up replacing all incoming edges of such headers by a single new edge from the new dummy vertex. This implies that such headers are not in the dominance frontier of the new dummy vertex, which is undesirable (as will be seen later). We ensure that such headers will be in the dominance frontier of the new dummy vertex by adding some more edges. For every loop header h that is not an entry vertex, we select some arbitrary pre-entry vertex p of the loop, and add the edge $p \rightarrow h$ to the graph.

More formally,

DEFINITION 5. *Given a loop L in a graph G, let* PreEntries(L) *denote the set of all pre-entry vertices of L, let* LoopBack(L) *denote the set of all loopback edges of L. For any non-empty set X, let* Arb(X) *denote an arbitrary element of X. Let* Entries(X) *denote the set of all entry vertices of X. We will denote the set of headers of L by $\mathcal{H}(L)$, abusing notation. We define the graph $\Psi_L(G)$ to be (V', E') where:*

$$\begin{aligned} V' = {} & V(G) \cup \{\delta_L\} \\ E' = {} & E(G) - \text{LoopBack}(L) \\ & \cup \{ p \rightarrow \delta_L \mid p \in \text{PreEntries}(L) \} \\ & \cup \{ \delta_L \rightarrow h \mid h \in \mathcal{H}(L) \} \\ & \cup \{ \text{Arb}(\text{PreEntries}(L)) \rightarrow h \mid h \in (\mathcal{H}(L) - \text{Entries}(L)) \} \end{aligned}$$

The above transformation can actually be simplified for reducible loops (loops with a single entry vertex): there is no need to add the new vertex δ_L or any of the other edges. We ignore such "optimizations" here for the sake of simplicity.

The above definition shows how we "transform away" a single loop L. We would like to apply this transformation to every loop in the graph, one after another. We now establish some results to ensure that this can be done in a meaningful way, using only the loops of the "original graph".

LEMMA 2. *Let L be a loop belonging to some minimal loop nesting forest of G. For any two vertices u and v in $V(G)$, v dominates u in G iff v dominates u in $\Psi_L(G)$.*

PROOF. We will establish that v does not dominate u in G iff v does not dominate u in $\Psi_L(G)$.

$\Rightarrow$. Assume that we have a path α in G from the entry vertex of the graph to u that does not contain the vertex v. If α does not contain any loopback edge of L, then α is a path in $\Psi_L(G)$ also, and the result follows. Otherwise, let α_p denote the longest prefix of path α that does not contain any vertex in L, and let α_s denote the longest suffix of path α that does not contain any loopback edge of L. Note that the last vertex of α_p must be a pre-entry vertex of L, while the first vertex of α_s must be a header of L. Hence, $\alpha_p \delta_L \alpha_s$ is a path in $\Psi_L(G)$ that does not contain vertex v. Hence, v does not dominate u in $\Psi_L(G)$.

$\Leftarrow$. Assume that we have a path α in $\Psi_L(G)$ from the entry vertex of the graph to u that does not contain the vertex v. If α does not contain any of the newly added edges, then α is a path in G too and the result follows. Otherwise, let α_s denote the shortest suffix of α that includes a header of L. (Note that α_s will not contain δ_L because any occurrence of δ_L in α will be followed by a header of L.) Let h denote the first vertex in α_s. Let α_p denote the shortest prefix of α that

includes some pre-entry vertex of L. (Note that α_p will not contain δ_L because any occurrence of δ_L in α will be preceded by a pre-entry vertex of L.) Let c denote the last vertex of α_p. Let h' denote some successor (in graph G) of c belonging to L. (c must have at least one successor belonging to L since c is a pre-entry vertex of L.) We consider two possible cases. First, consider the case where v is not in L. Since L is strongly connected, there exists a path β within L from h' to h that does not contain v. It follows that $\alpha_p\beta\alpha_s$ is a path in G that does not contain v. Next, consider the case where v belongs to L. Since a header of a loop is not dominated by any other vertex in the loop, we know that v does not dominate h. Hence, there exists some path β from the graph's entry vertex to h that does not contain v. It follows that $\beta\alpha_s$ yields a path in G from the graph's entry vertex to u. Hence, v does not dominate u in G either. □

Note that the above proof (the "⇒" direction) also establishes that every vertex in $\Psi_L(G)$ is reachable from the graph's entry vertex. The above lemma establishes that our transformation, when applied to a single loop L, preserves the domination relation (on the vertices of the original graph).

THEOREM 6. *If $\mathcal{L}$ is a minimal loop nesting forest for a graph G, and L is an outermost loop in $\mathcal{L}$, then $\mathcal{L} - \{ L \}$ is a minimal loop nesting forest for $\Psi_L(G)$.*

PROOF. (a) The other loops in the graph do not contain any of the headers of L (from the definition of minimality for loop nesting forests) and, hence, trivially continue to be strongly connected in $\Psi_L(G)$.

(b) We first show that a strongly connected set of $\Psi_L(G)$ can not contain any of the newly added edges. Note that any such set must include a pre-entry vertex p of L as well as a header h of L. This implies that G also contains a cycle which includes both p and h. But such a cycle can not be covered by any loop in $\mathcal{L}$ (since L is an outermost loop of $\mathcal{L}$) and this contradicts the fact that $\mathcal{L}$ is a loop nesting forest for G. Since a strongly connected set X of $\Psi_L(G)$ does not contain any of the newly added edges it must be a strongly connected set in G also. Further, X does not contain any of the headers of L (since we dropped all loopback edges of L). Hence, X must be covered by some loop in $\mathcal{L} - \{ L \}$ in both G and $\Psi_L(G)$.

(c) The proper nesting property obviously continues to hold for $\mathcal{L} - \{ L \}$.

(d) We know that none of the headers of loops in $\mathcal{L} - \{ L \}$ is dominated by any other vertex in the loop in G. It follows from Lemma 2 none of the headers of loops in $\mathcal{L} - \{ L \}$ is dominated by any other vertex in the loop $\Psi_L(G)$ also. □

DEFINITION 6. *Let $\mathcal{L}$ be a minimal loop nesting forest for G. Let $L_1, \cdots, L_k$ be an ordering of the loops in $\mathcal{L}$ where outer loops appear before inner loops. We define $\Psi_{\mathcal{L}}(G)$ to be the graph $\Psi_{L_k}(\cdots \Psi_{L_1}(G))$.*

THEOREM 7. *If $\mathcal{L}$ is a minimal loop nesting forest then $\Psi_{\mathcal{L}}(G)$ is an acyclic graph.*

PROOF. We can think of $\Psi_{\mathcal{L}}(G)$ as being generated from G in two steps. We first delete all loopback edges to obtain the graph $\mathcal{F}_{\mathcal{L}}(G)$. We then add all the edges from pre-entry vertices as in the definition above.

We observed earlier (Theorem 4) that the leaves of an "ordered forest" representation of $\mathcal{L}$, when enumerated in order, yields a topological sort ordering for $\mathcal{F}_{\mathcal{L}}(G)$

in which all vertices belonging to a loop are consecutive. In particular, all pre-entry vertices of a loop must appear before any of its headers in this ordering. This implies that the new edges we add to generate $\Psi_{\mathcal{L}}(G)$ does not create any cycle. In particular, if we label the internal node representing a loop L with the new node δ_L, then a pre-order enumeration of the "ordered forest" yields a topological sort order for the transformed graph $\Psi_{\mathcal{L}}(G)$. It follows that $\Psi_{\mathcal{L}}(G)$ is acyclic. □

The subsequent sections will show how the transformed graph can be used to solve the problems of computing the dominator tree and computing iterated dominance frontiers in the original graph. We now consider some complexity issues concerning the proposed graph transformation.

THEOREM 8. *Let $\mathcal{L}$ be a minimal loop nesting forest for G. If no vertex in the graph is the entry vertex of two or more loops in $\mathcal{L}$, then the size of the graph $\Psi_{\mathcal{L}}(G)$ is at most twice the size of G and $\Psi_{\mathcal{L}}(G)$ can be constructed in time linear in the size of$G$ (given $\mathcal{L}$).*

PROOF. Note that the number of loops in $\mathcal{L}$ can be no more than the number of vertices in G (since every loop has at least one header, and a vertex can not be the header of more than one loop in a minimal loop nesting forest). Since we add one new vertex per loop in the graph, the number of vertices in $\Psi_{\mathcal{L}}(G)$ is at most twice the number of vertices in G. Of the new edges added to the graph, we may ignore those directed to the headers of the loop. (We add at most two new edges directed towards a given header, and they are compensated for by the deletion of loopback edges anyway.) That leaves the new edges added from pre-entry vertices to δ_L. Let p be a pre-entry vertex of L. Then, the original graph G must have an edge $p \to e$ where e is an entry vertex of L. In loop nesting forests where a vertex is the entry vertex of at most one loop, every "entry edge" $p \to e$ results in the creation of at most one new edge $p \to \delta_L$. Hence, for these forests, the size of the graph $\Psi_{\mathcal{L}}(G)$ is guaranteed to be at most twice the size of the initial graph.

Let us now consider the complexity of the construction of $\Psi_{\mathcal{L}}(G)$. It follows from the above discussion that once the set of loopback edges and the set of pre-entry vertices of every loop L is known, the transformation can be done in time linear in the size of G. Given a loop nesting forest, it is easy to number all the vertices consecutively and to associate every loop with a range that identifies exactly the set of vertices belonging to that loop. Hence, checking if a vertex belongs to a loop can be done in constant time and checking an edge to see if it is a loopback edge is straightforward. As for identifying pre-entry vertices, our assumption that no vertex is the entry vertex of two or more loops implies that all the entry vertices of a loop appear as the children of the loop in the loop nesting forest. It is straightforward to identify the predecessors of these entry vertices that do not belong to the loop. □

Steensgaard's forest, the Sreedhar-Gao-Lee forest, as well as the Reduced Havlak forest satisfy the requirement that different loops never have a common entry vertex. Hence, the transformed graphs produced using these forests have a size at most twice the size of the initial graph. However, it is possible for a vertex to be the entry vertex of many loops in Havlak's forest. Hence, the size of $\Psi_{\mathcal{L}}(G)$ can be quadratic in the size of G if $\mathcal{L}$ is Havlak's forest. This is the motivation for the Reduced Havlak forest.

6. CONSTRUCTING THE DOMINATOR TREE OF AN IRREDUCIBLE GRAPH

We now show how the transformation described in the previous section can be used to construct the dominator tree of an irreducible graph.

6.1 The Algorithm

LEMMA 3. *Let $\mathcal{L}$ be a minimal loop nesting forest for G. For any two vertices u and v in $V(G)$, v dominates u in G iff v dominates u in $\Psi_{\mathcal{L}}(G)$.*

PROOF. This follows from Lemma 2, Theorem 6, and the definition of $\Psi_{\mathcal{L}}(G)$. □

LEMMA 4. *Let $\mathcal{L}$ be a minimal loop nesting forest for G and let L be a loop in $\mathcal{L}$. Vertex δ_L does not strictly dominate any vertex in $\Psi_{\mathcal{L}}(G)$.*

PROOF. Note that it is enough to show that δ_L does not dominate any of its successors. The successors of δ_L are the headers of L. Note that, by construction, every header of L has at least one other pre-entry vertex of L as a predecessor. But δ_L does not dominate any of the pre-entry vertices of L (since $\Psi_L(G)$ is acyclic and the pre-entry vertices of L are predecessors of δ_L.) Consequently, δ_L does not dominate any header of L. Hence, δ_L does not dominate any other vertex in $\Psi_L(G)$. □

It follows from the Lemma 4 that δ_L must be a leaf in the dominator tree of $\Psi_{\mathcal{L}}(G)$. It also follows from Lemma 3 that deleting δ_L from the dominator tree of $\Psi_L(G)$ yields the dominator tree of the original graph G. Given a tree T and a set of leaves L in T, let $T - L$ denote the subtree of T obtained by removing all the leaves in L. Let $\Delta_{\mathcal{L}}$ to denote the set $\{\delta_L \mid L \in \mathcal{L}\}$.

THEOREM 9. *Let $\mathcal{L}$ be a minimal loop nesting forest for G.*

$$\text{DomTree}(G) = \text{DomTree}(\Psi_{\mathcal{L}}(G)) - \Delta_{\mathcal{L}}$$

PROOF. Follows from Lemma 3 and Lemma 4. □

A note on the definition of the transformed graph $\Psi_L(G)$. If we do not add an edge from some (arbitrary) pre-entry vertex of loop L to any header of L that is not an entry vertex of the loop, δ_L will dominate such headers and not be a leaf in the dominator tree. We could, however, generalize the definition of $\text{DomTree}(\Psi_{\mathcal{L}}(G)) - \Delta_{\mathcal{L}}$ suitably (when we remove an internal node δ_L we need to make δ_L's children the children of δ_L's parent) so that the above theorem still holds. Thus, these additional edges are not critical to our dominator tree construction algorithm. We will see later, however, that they *are* critical for our algorithm for computing iterated dominance frontier.

It should now be clear how a minimal loop nesting forest of a graph can be used in conjunction with the algorithm outlined in Section 3.1.1 to construct the dominator tree of the graph.

Let us now consider the complexity of this algorithm. Once the acyclic graph $\Psi_{\mathcal{L}}(G)$ has been constructed, the dominator tree can be constructed in time linear in the size of $\Psi_{\mathcal{L}}(G)$. Hence, by using the Reduced Havlak forest, we obtain an almost linear time algorithm for constructing the dominator tree of a graph.

6.2 The Computation of Least Common Ancestors in a Dynamic Tree

The algorithms presented in this section for constructing the dominator tree require the computation of least common ancestors in a dynamic tree, namely the dominator tree as it is constructed. This requires the use of a "dynamic" tree data structure that supports the operations of adding a new leaf as well as the computation of least common ancestors.

Gabow [1990] presents a data structure and algorithms for a dynamic tree that allows both these operations to be done in constant time. Some believe Gabow's algorithm to be impractical, because it has large constant factors and is complex to implement. In practice, some of the other dynamic tree data structures [Mehlhorn and Tsakalidis], which are asymptotically nonoptimal, may be easier to implement and perform better.

In this section we outline a very simple, almost naive, data structure and algorithm for computing least common ancestors. We will analyze the complexity of this algorithm when used in the computation of dominator trees and argue that it should perform well in practice.

The data structure we use is very simple. The only algorithm we maintain for a vertex is its parent and its "depth". The operation "addLeaf(c,p)" adds a new vertex c as the child of an vertex p in the tree.

```
procedure addLeaf(vertex c, p)
  c.parent := p;
  c.depth := p.depth + 1;
```

Now we describe the algorithm for computing the least common ancestor of k vertices $p[1], \cdots, p[k]$. The essential idea is to compute the least common ancestor of two vertices u and v by walking up the tree from u and v until we reach their least common ancestor. The depth of the vertices is used to "synchronize" the two traversals (from u and v respectively): we walk up from the vertex at a greater depth, say u, to identify its ancestor w that is at the same depth as the other vertex v. We then walk up the tree from w and v lock step until we reach a common ancestor of w and v.

We compute the least common ancestor of k vertices $p[1], \cdots, p[k]$ by applying the above algorithm iteratively, with a simple optimization. The general iterative step is that we have computed the least common ancestor v of $p[i], \cdots, p[i-1]$ and need to compute the least common ancestor of v and $p[i]$. Now,as we walk up the tree from vertex $p[i]$, if we encounter a vertex that was already "visited" during the computation of the least common ancestor of $p[i], \cdots, p[i-1]$, then the current least common ancestor v is guaranteed to be an ancestor of $p[i]$ as well and we may skip $p[i]$ and proceed to the $i+1$-th iteration.

The algorithm is described in pseudocode below. ("Marking" visited vertices can be done using any standard technique.)

```
function lca(int k; vertex p[k])
  // Compute lca (p[1], p[2], ..., p[k]).
  // Assume that all vertices are marked unvisited initially.
  v := p[1];
  outerloop: for i = 2 to k do {
```

```
    // compute lca (v,p[i]) and assign it to v:
    // step 1: if either v or p[i] is at a greater depth than the other, identify
    // its ancestor that is at the same depth as the other vertex.
    u = p[i];
    while (v.depth > u.depth) do {
        mark v as visited;
        v := v.parent;
    }
    while (u.depth > v.depth) do {
        if (u is marked visited) then
            // u is a descendant of v; so, v == lca(u,v); just continue
            // with the next iteration of outerloop
            continue outerloop;
        mark u as visited;
        u := u.parent;
    }
    // Now u and v are at the same depth. Traverse up the tree from u
    // and v in lockstep until we reach the least-common-ancestor.
    while (u != v) do {
        mark u as visited; u := u.parent;
        mark v as visited; v := v.parent;
    }
}
mark all visited vertices as unvisited
return v
```

The worst case cost of the above computation is $O(|T|)$, where $|T|$ is the size of the current tree, which is not very good. However, let us now compute the total cost of all the least common ancestor operations performed during the construction of the dominator tree of an acyclic graph. Recall that for every vertex x with predecessors $p[1], \cdots, p[k]$ we compute $lca(k, p)$ to identify the immediate dominator of x. Consider any vertex w that is marked visited during the above computation. Clearly, x must be in the dominance frontier of w since w is an ancestor (in the dominator tree) of a predecessor of x and also a descendant of the immediate dominator of x. Further, w is visited only once during the computation. Hence, the cost of computing the immediate dominator of x is proportional to the number of predecesors of the vertex x + the number of vertices w such that x is in $DF(w)$.

Now, the sum of the cost of performing this operation for every vertex x in the graph is bounded by the size of the graph plus the sum of the sizes of the dominance frontier set of all vertices in the graph. This quantity, in principle, could be quadratic in the size of the graph. But as Cytron et al. show in their classical paper on computing the SSA form [Cytron et al. 1991], this quantity tends to be linear in the size of the given graph in practice. The argument that this quantity will tend to be linear in practice is actually stronger in the context of our application since we do not perform this computation over the control-flow graph, but on a transformed acyclic graph. For example, nested repeat-until loops lead to a quadratic blowup in the total size of the dominace frontiers, but are handled

in linear time in the above least-common-ancestor computation since we work only with the acyclic graph obtained by dropping the backedges of all the repeat-until loops.

7. COMPUTING THE ITERATED DOMINANCE FRONTIER IN AN IRREDUCIBLE GRAPH

We now consider how the algorithm described in Section 3.2 can be generalized to handle irreducible graphs. In particular, we will show that the iterated dominance frontier of any set of vertices X in a graph G can be found by computing the iterated dominance frontier of a corresponding set $X \cup HLC_{\mathcal{L}}(X)$ in the acyclic graph $\Psi_{\mathcal{L}}(G)$ using the following generalization of Theorem 1:

$$DF^{+}_{G}(X) = HLC_{\mathcal{L}}(X) \ \cup \ DF^{+}_{\Psi_{\mathcal{L}}(G)}(X \cup HLC_{\mathcal{L}}(X)).$$

In the sequel we will actually establish the above equality in a slightly different form, which utilizes the fact that the new vertex δ_L we add for any loop L serves as a representative of all the headers of L. Let us denote the loop body of L by $\mathcal{B}(L)$ and the set of headers of L by $\mathcal{H}(L)$. Define $\Delta_{\mathcal{L}}(X)$ to be $\{\delta_L \mid L \in \mathcal{L} \text{ and } X \cap \mathcal{B}(L) \neq \phi\}$. Thus, $\Delta_{\mathcal{L}}(X)$ is the set of dummy headers added for loops that overlap (that is, contain some vertex in) X. Define $\Delta_{\mathcal{L}}$ to be $\{\delta_L \mid L \in \mathcal{L}\}$. Thus, $\Delta_{\mathcal{L}}$ is the set of all dummy headers added. We will establish that

$$DF^{+}_{G}(X) = DF^{+}_{\Psi_{\mathcal{L}}(G)}(X \cup \Delta_{\mathcal{L}}(X)) - \Delta_{\mathcal{L}}.$$

In the following sequence of lemmas $\mathcal{L}$ denotes a minimal loop nesting forest for a graph G, and L denotes some loop in $\mathcal{L}$. We begin by establishing that any header h of any loop L containing a given vertex u is a join node of that vertex and the graph's entry vertex.

LEMMA 5. *For any $u \in \mathcal{B}(L)$, $J_G(u, entry(G)) \supseteq \mathcal{H}(L)$.*

PROOF. We will first establish a general result about join nodes: if there exists a non-empty path from u to v such that none of the vertices in the path strictly dominate v, then v must be a join node for u and $entry(G)$. This can be established via induction on path length. If the path consists of a single edge $u \rightarrow v$, then the claim is trivially true since there must also exist a path from $entry(G)$ to v that does not contain u (since u does not strictly dominate v). If the path consists of more than one edge, let w denote the successor of u in this path. Our inductive hypothesis implies that v is a join node for w and $entry(G)$. Hence, there exist paths $\alpha : entry(G) \rightarrow^{+} v$ $\beta : w \rightarrow^{+} v$ that converge on v. If path α does not contain vertex u, the result follows, since α and $u\beta$ form a converging pair. Otherwise, let α' denote the suffix of α that forms a path from u to v. Now, we have two vertex disjoint paths between u and v, namely α' and $u\beta$. Since u does not strictly dominate v, there must exist a path γ from $entry(G)$ to v that does not contain u. If γ does not intersect either α' or $u\beta$, then clearly v is a join node for u and $entry(G)$. Assume, without loss of generality, that γ intersects α' before it intersects $u\beta$. Let δ denote the path from $entry(G)$ to v that follows γ until it intersects α' and then follows α'. Clearly, δ and $u\beta$ form a converging pair, and the result follows.

For any $u \in \mathcal{B}(L)$ and $h \in \mathcal{H}(L)$, we know that there exists a path from u to h consisting only of vertices in L. Since none of the vertices in L strictly dominate h, it follows that h is a join node for u and *entry*(G). □

Once we delete the loopback edges of L, in generating the graph $\Psi_L(G)$, the headers of L may no longer be join nodes of u and *entry*(G). However, the headers of the loop *are* join nodes of the vertex δ_L and *entry*(G) in the transformed graph $\Psi_L(G)$, as shown below.

LEMMA 6. $J_{\Psi_L(G)}(\{entry(G), \delta_L\}) \supseteq \mathcal{H}(L)$.

PROOF. Let $h \in \mathcal{H}(L)$. Note that the graph $\Psi_L(G)$ contains the edge $\delta_L \rightarrow h$. We also know that δ_L does not dominate h in $\Psi_L(G)$ (Lemma 4). Hence, there exists a path from *entry*(G) to h that does not contain δ_L. The result follows. □

The reason we add an edge from some arbitrary pre-entry vertex of loop L to any header of the loop that is not an entry vertex in generating the graph $\Psi_L(G)$ is to ensure that Lemma 6 holds true. The addition of these edges is criticial for our algorithm for computing iterated dominance frontier, but not as important for our algorithm for constructing the dominator tree.

Let us now see how the iterated dominance frontier (or join set) of a set of vertices in the graph G can be computed in the transformed graph $\Psi_L(G)$. What we would like to show is that the iterated dominance frontier of a set Y is the same in both G and $\Psi_L(G)$ if Y does not contain any vertex belonging to loop L. If Y does contain vertices belonging to loop L, then the iterated dominance frontier of $Y \cup \{\ \delta_L\ \}$ in $\Psi_L(G)$ yields the iterated dominance frontier of Y in G.

We begin by considering a simple class of sets of vertices, namely those that contain the set of headers of L.

LEMMA 7. *Let L be a loop in a graph G, and let Y be a set of vertices such that $Y \supseteq \mathcal{H}(L) \cup \{entry(G)\}$. Then, $J_G(Y) = J_{\Psi_L(G)}(Y \cup \{\delta_L\}) - \{\delta_L\}$.*

PROOF. We will first show that $J_G(Y) \subseteq J_{\Psi_L(G)}(Y \cup \{\delta_L\}) - \{\delta_L\}$.

Consider any $w \in J_G(Y)$. We will show that $w \in J_{\Psi_L(G)}(Y \cup \delta_L)$. If $w \in \mathcal{H}(L)$, this is trivially true, since $\mathcal{H}(L) \subseteq J_{\Psi_L(G)}(\{\delta_L, entry(G)\})$. (from Lemma 6).

Otherwise, there exist two converging paths $\alpha : u \rightarrow^+ w$ and $\beta : v \rightarrow^+ w$ in G, where $u, v \in Y$. Define α' to be the longest suffix of α not containing any backedge of L. Define β' similarly. Clearly, α' and β' are converging paths in $\Psi_L(G)$. Further, these are not empty paths since $w \notin \mathcal{H}(L)$. Hence, w is a join node for Y in $\Psi_L(G)$ as well.

Now, we will show that $J_{\Psi_L(G)}(Y \cup \{\delta_L\}) - \{\delta_L\} \subseteq J_G(Y)$. Consider any $w \in J_{\Psi_L(G)}(Y \cup \delta_L) - \{\delta_L\}$. We will show that $w \in J_G(Y)$. If $w \in \mathcal{H}(L)$, this is trivially true (from Lemma 5).

Otherwise, there exists two converging simple paths $\alpha : u \rightarrow^+ w$ and $\beta : v \rightarrow^+ w$ in $\Psi_L(G)$, where $u, v \in Y \cup \delta_L$. Let α' and β' denote the longest suffixes of α and β, respectively, that do not contain any of the newly added edges. α' and β' are converging paths in G, and these are not empty paths since $w \notin \mathcal{H}(L)$. It follows that w is a join node for Y in G. □

The above result shows that the transformation Ψ_L "preserves" (in a loose sense) the join set of any set of vertices that includes all the headers of L. We now show that the above result, in fact, holds for any set of vertices that overlaps the loop L.

LEMMA 8. *Let L be a loop in a graph G, and let Y be a set of vertices such that entry$(G) \in Y$ and $Y \cap \mathcal{B}(L) \neq \phi$. Then, $J_G(Y) = J_{\Psi_L(G)}(Y \cup \{\delta_L\}) - \{\delta_L\}$.*

PROOF. First note that $J_G(Y) \supseteq \mathcal{H}(L)$ (from Lemma 5). Hence, $J_G(Y) = J_G(Y \cup \mathcal{H}(L))$. (This follows from the fact that the join set is equal to the iterated join set [Weiss 1992].) Also note that $J_{\Psi_L(G)}(Y \cup \{\delta_L\}) \supseteq \mathcal{H}(L)$. (from Lemma 6). Hence, $J_{\Psi_L(G)}(Y \cup \{\delta_L\}) = J_{\Psi_L(G)}(Y \cup \mathcal{H}(L) \cup \{\delta_L\})$ (again using the fact that the join set and the iterated join set are equal). But $J_G(Y \cup \mathcal{H}(L)) = J_{\Psi_L(G)}(Y \cup \mathcal{H}(L) \cup \{\delta_L\}) - \{\delta_L\}$ from Lemma 7. The result follows. □

Now that we have seen the effect of the transformation Ψ_L on the join set of any set that overlaps the loop L, we turn our attention to sets of vertices that are disjoint from the loop. Such sets fall into two classes, each of which we will consider separately. These two classes are based on a slight generalization of the concept of converging paths that was used to defined the concept of join set. We say that set Y of vertices converges on a set L of vertices iff there exist two non-null paths $\alpha : u \rightarrow^+ x$ and $\beta : v \rightarrow^+ y$ where $\{u, v\} \subseteq Y$, $\{x, y\} \subseteq L$, and the paths α and β are disjoint except possibly for the end vertices $\{x, y\}$. We will first establish that the transformation Ψ_L does not affect the join set of any set of vertices that does not converge on the loop.

LEMMA 9. *Let L be a loop in a graph G, and let Y be a set of vertices such that entry$(G) \in Y$ and $Y \cap \mathcal{B}(L) = \phi$. If Y does not converge on $\mathcal{B}(L)$ in G, then $J_G(Y) = J_{\Psi_L(G)}(Y)$.*

PROOF. We will first show that $J_G(Y) \subseteq J_{\Psi_L(G)}(Y)$. In other words, we need to show that if x is a join node for Y in G then x is a join node for Y in $\Psi_L(G)$ also. Consider any two converging disjoint paths $\alpha : u \rightarrow^+ x$ and $\beta : v \rightarrow^+ x$ in G, where $u, v \in Y$. If neither α nor β contain any backedge of L, then α and β are paths in $\Psi_L(G)$ too, and x is a join node for Y in $\Psi_L(G)$. Otherwise, either α or β contains a backedge of L. (Note that it is not possible for both α and β to contain a backedge of L, since that would imply that Y converges on $\mathcal{B}(L)$, contradicting our assumption.) Assume, without loss of generality, that α contains a backedge of L and that β does not. Let α be of the form $u_1 u_2 \cdots u_k$. Let u_i be the first pre-entry vertex of L in path α, and let u_j be the last header of L in path α. Define α' to be $u_1 \cdots u_i \delta_L u_j \cdots u_k$. Clearly, α' and β are proper, non-null paths in $\Psi_L(G)$. Further α' and β must be disjoint, except for x, because α' consists only of edges in α, which we know is disjoint from β, and the newly inserted edges involving vertex δ_L, which does not occur in β.

We will now show that $J_{\Psi_L(G)}(Y) \subseteq J_G(Y)$. Let $x \in J_{\Psi_L(G)}(Y)$. Consider any two converging paths $\alpha : u \rightarrow^+ x$ and $\beta : v \rightarrow^+ x$ in $\Psi_L(G)$, where $u, v \in Y$. We define path α' in G as follows. If α does not contain any of the newly added edges, then α' is defined to be α. Otherwise, let α_p denote the longest prefix of α_p that does not contain any of the newly added edges, and let α_s denote the longest suffix of α that does not contain any of the newly added edges. Note that α_p must end at some pre-entry vertex p of L, while α_s must start at some header h of L. There

must exist a path α_m in G from p to h consisting only of vertices in L, except for p itself (since L is strongly connected). Define α' to be $\alpha_p\alpha_m\alpha_s$. Define β' similarly. Note that α and β can not both contain vertices in loop L as that would imply that Y converges on $\mathcal{B}(L)$. Assume, without loss of generality that β contains no vertex from L. Then we must have $\beta = \beta'$. We have α' consisting of a subset of the edges in α and a subset of edges involving vertices inside L. Since we started with α and β that were disjoint and since β has no vertices from L α' and β' must be disjoint. Paths α' and β' converge at x in graph G. Hence, $x \in J_G(Y)$.

Hence, $J_G(X) = J_{\Psi_L(G)}(X)$. □

We will now consider sets of vertices that do converge on the loop L.

LEMMA 10. *Let L be a loop in a graph G, and let Y be a set of vertices such that entry$(G) \in Y$ and $Y \cap \mathcal{B}(L) = \phi$. If Y converges on $\mathcal{B}(L)$ in G, then $J_G(Y) = J_{\Psi_L(G)}(Y) - \{\delta_L\}$.*

PROOF. If Y converges on $\mathcal{B}(L)$, then we have converging paths $\alpha : u \rightarrow^+ x$ and $\beta : v \rightarrow^+ y$ where $\{u, v\} \subseteq Y$, and $\{x, y\} \subseteq \mathcal{B}(L)$. Assume, without loss of generality, that none of the vertices in α and β except their last vertices are in $\mathcal{B}(L)$. (If this is not true, we may simply choose the appropriate prefixes of α and β that satisfy this condition.) Since $\mathcal{B}(L)$ is strongly connected, it follows that path α can be extended into a path α' to vertex y and that y is a join node for Y. It follows that $J_G(Y) = J_G(Y \cup \{\ y\ \})$, using the fact that the join set and iterated join set are equal. It follows from Lemma 5 that $J_G(Y)$ includes $\mathcal{H}(L)$.

Hence, $J_G(Y) = J_G(Y \cup \mathcal{H}(L))$ using the fact that the join set and iterated join set are equal.

Similarly, paths α and β can be trivially modified (by replacing each of their last edge $s \rightarrow t$ by a corresponding edge $s \rightarrow \delta_L$) into paths in $\Psi_L(G)$ that converge on δ_L. It follows that δ_L is a join node for Y in $\Psi_L(G)$. It follows from Lemma 6 that $J_{\Psi_L(G)}(Y)$ includes $\mathcal{H}(L) \cup \{\delta_L\}$.

Hence, $J_{\Psi_L(G)}(Y) = J_{\Psi_L(G)}(Y \cup \mathcal{H}(L) \cup \{\delta_L\})$, using the fact that the join set and iterated join set are equal.

The result follows from Lemma 7. □

We now combine the previous two results.

LEMMA 11. *Let L be a loop in a graph G, and let Y be a set of vertices such that entry$(G) \in Y$ and $Y \cap \mathcal{B}(L) = \phi$. Then, $J_G(Y) = J_{\Psi_L(G)}(Y) - \{\delta_L\}$.*

PROOF. This follows directly from Lemmas 9 and 10. □

We are now ready to combine the various previous lemmas to show how the iterated dominance frontier of any set X in the graph G can be computed in the transformed graph $\Psi_L(G)$. Given a loop L, and a set of vertices X, define $\Delta_L(X)$ to be the empty set if X and $\mathcal{B}(L)$ are disjoint and $\{\delta_L\}$ otherwise.

LEMMA 12. *Let L be a loop in a graph G and let X be any set of vertices in G.*

$$DF^+_G(X) = DF^+_{\Psi_L(G)}(X \cup \Delta_L(X)) - \{\delta_L\}.$$

PROOF. Follows from Lemma 8 and Lemma 11. Recall that for any set X of vertices in any graph H, $DF^+_H(X) = J_H(X \cup \{entry(H)\})$. □

Recall that $\Delta_{\mathcal{L}}(X)$ is defined to be $\{\delta_L \mid L \in \mathcal{L} \text{ and } \mathcal{B}(L) \cap X \neq \phi\}$ and that $\Delta_{\mathcal{L}}$ is defined to be $\{\delta_L \mid L \in \mathcal{L}\}$.

THEOREM 10. *Let $\mathcal{L}$ be a minimal loop nesting forest for a graph G and let X be any set of vertices in G.*

$$DF_G^+(X) = DF^+_{\Psi_{\mathcal{L}}(G)}(X \cup \Delta_{\mathcal{L}}(X)) - \Delta_{\mathcal{L}}$$

PROOF. Follows from Theorem 6 and iterated applications of Lemma 12. □

Theorem 10 shows how the algorithm of Section 3.2.1 can be adapted to work for arbitrary graphs using loop nesting forests. Given any set X of vertices in a graph G, we can walk up any loop nesting forest $\mathcal{L}$ of G to identify the set $\Delta_{\mathcal{L}}(X)$. We can then apply the algorithm of Section 3.2.1 to the set of vertices $X \cup \Delta_X(\mathcal{L})$ in the acyclic graph $\Psi_{\mathcal{L}}(G)$ to identify the iterated dominance frontier of X in G.

If we use the Reduced Havlak loop nesting forest, the whole algorithm runs in almost linear time. (The construction of the loop nesting forest runs in almost linear time, and the subsequent steps run in linear time.)

8. RELATED WORK

The classical algorithm for identifying loops is Tarjan's interval finding algorithm [Tarjan 1974], which is restricted to reducible graphs. Recently, several algorithms have been proposed for identifying loops in arbitrary graphs. This includes the algorithms described by Steensgaard [1993], Sreedhar *et. al.* [1996], and Havlak [1997]. Ramalingam [1999] improves upon the efficiency of these different algorithms.

The concepts of domination and dominator tree have many applications in program analysis and program optimization. The most well known and widely used algorithm for computing dominators is the almost linear time algorithm of Lengauer and Tarjan [1979]. Harel [1985] claimed a linear time algorithm for constructing the dominator tree, but Alstrup *et al.* [1996;] showed that Harel's algorithm was, in fact, non-linear, and presented a modified algorithm that runs in linear time. More recently, Buchsbaum *et al.* [1998] have presented a simpler linear time algorithm for constructing the dominator tree. They also present experimental results which show that, in practice, the simpler algorithm due to Lengauer and Tarjan is probably the algorithm of choice.

Reif and Lewis [1977] show how a problem called the P-graph completion problem can be reduced in linear time to the problem of computing a dominator tree. The P-graph completion problem is the same as that of computing the iterated dominance frontier of a set of vertices that includes the graph's entry vertex, but the reduction due to Reif and Lewis seems to have been forgotten in the recent literature. The concept of the iterated dominance frontier was brought to prominence by the work of Cytron *et al.* [1991] on computing the Static Single Assignment form, a data structure that has found numerous applications in program optimization. The algorithm presented by Cytron *et al.* for computing the iterated dominance frontier of a set of vertices takes quadratic time in the worst case, but appears to be competitive with other linear time algorithms [Pingali and Bilardi 1997; Sreedhar and Gao 1995] for this problem in practice. In fact, in applications such as constructing the SSA form, which require computing the iterated dominance frontier of a number of different sets of vertices, the Cytron *et al.* algorithm has been shown

to perform better than the Sreedhar *et al.* linear time algorithm because of the way the Cytron *et al.* algorithm amortizes some of the work over the multiple iterated dominance frontier computation [Sreedhar 1995].

9. CONCLUSION

In this paper, we have presented new, almost linear time, algorithms for two graph-theoretic problems, that of constructing the dominator tree of a graph and that of computing the iterated dominance frontier of a set of vertices in a graph. We do not claim that the new algorithms will be more efficient than or as efficient as the standard algorithms used to solve these problems. Our primary goal in presenting the new algorithms is to illustrate an approach we believe may be of interest in solving other graph-theoretic problems in the context of compilation.

In particular, we have utilized the above two applications as a vehicle for understanding the concepts of loops and loop nesting forests in graphs. We have shown how three previously defined loop nesting forests can be generated as instances of a single parametric definition, and studied the properties common to the family of loop nesting forests that can be generated from this parametric definition. We have shown how these loop nesting forests can be used (in our two applications) to transform arbitrary problem instances (including those based on irreducible graphs) into equivalent problem instances based on acyclic graphs. We have also introduced a new loop nesting forest, which too is an instance of our parametric definition. This new loop nesting forest, which can be constructed in almost linear time, turns out to be better suited for the two applications we consider. We believe that our problem reduction strategy as well as the new loop nesting forest may be of use in various other applications as well.

ACKNOWLEDGMENTS

I would like to thank the anonymous referees, John Field, Deepak Goyal, V. C. Sreedhar, and Mark Wegman for their comments which helped improve this paper.

REFERENCES

Alstrup, S., Harel, D., Lauridsen, P. W., and Thorup, M. Dominators in linear time. *SIAM J. Comput.*. To appear.

Alstrup, S., Lauridsen, P. W., and Thorup, M. 1996. Dominators in linear time. Tech. Rep. 96/35, Department of Computer Science, University of Copenhagen.

Alstrup, S. and Lauridsen, P. W. 1996. A simple and optimal algorithm for finding immediate dominators in reducible graphs. Tech. Rep. D-261/96, TOPPS Bibliography.

Buchsman, A. L., Kaplan, H., Rogers, A., and Westbrook, J. 1998. A new, simpler linear-time dominators algorithm. *ACM Trans. Program. Lang. Syst. 20,* 6, 1265–1296.

Choi, J.-D., Cytron, R., and Ferrante, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*. 55–66.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 452–490.

Cytron, R., Lowry, A., and Zadeck, F. K. 1986. Code motion of control structures in high-level languages. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*. 70–85.

Gabow, H. N. 1990. Data structure for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*. 434–443.

Harel, D. 1985. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of 17th ACM Symposium on Theory of Computing*. 185–194.

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

Lengauer, T. and Tarjan, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 121–141.

Mehlhorn, K. and Tsakalidis, A. Data structures. In *Handbook of Theoretical Computer Science: Volume A (Algorithms and Complexity)*, J. van Leeuwen, Ed. MIT Press, 328–329.

Morgan, R. 1998. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, MA.

Pingali, K. and Bilardi, G. 1997. Optimal control dependence computation and the roman chariots problem. *ACM Trans. Program. Lang. Syst. 19,* 3 (May), 462–491.

Ramalingam, G. and Reps, T. 1994. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*. 287–298.

Ramalingam, G. 1999. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst. 21,* 2, 175–188.

Reif, J. H. and Lewis, H. R. 1977. Symbolic evaluation and the global value graph. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*. Los Angeles, California, 104–118.

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.

Sreedhar, V. C. and Gao, G. 1995. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*. 62–73.

Sreedhar, V. C. 1995. Efficient program analysis using dj graphs. Ph.D. thesis, School of Computer Science, McGill University, Montréal, Québec, Canada.

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.

Weiss, M. 1992. The transitive closure of control dependence: The iterated join. *ACM Lett. Program. Lang. Syst. 1,* 2, 178–190.

Received Nov 18 1999; Revised Apr 16 2001; Accepted Sept 14 2001

Appendix

This appendix presents the various previous definitions of loop nesting forests and relates them to the axiomatic definition of this paper. It also shows how the Reduced Havlak forest introduced in this paper can be construced in almost linear time.

Steensgaard's Loop Nesting Forest

Given a set of vertices X, a vertex belonging to X said to be an *entry* vertex of X if it has a predecessor outside X. Let Entries(X) denote the set of all entry vertices of X.

Thus, Entries is the function that maps a given set of vertices to the set of its entry vertices. Steensgaard's definition can be obtained as a special case of our definition, by identifying the headers of a loop to be its entry vertices. (Steensgaard's original definition takes a slightly different form, but is completely equivalent. In particular, consider the i-th iteration in the iterative identification of loops. This step consists of identifying the non-trivial strongly connected components of $\mathcal{F}_{\mathcal{H}}^{i-1}(G)$ as loops. Steensgaard refers to a vertex u in a loop X identified in this step as a header of X if u has a predecessor outside X *in the graph* $\mathcal{F}_{\mathcal{H}}^{i-1}(G)$. Our definition, however, identifies a vertex u in the loop X as a header of X if u has a predecessor outside X *in the original graph* G. Both definitions are equivalent, since the construction process ensures that if an incoming edge of a vertex x is dropped, then x will never be contained in any loop (strongly connected component) subsequently identified.)

DEFINITION 7. $\text{SteensgaardsForest}(G) = \text{LoopForest}_{\text{Entries}}(G)$

THEOREM 11. $\text{SteensgaardsForest}(G)$ *is a minimal loop nesting forest.*

PROOF. It follows from Theorem 3 that we just need to prove that for every $X \in \text{Loops}_{\text{Entries}}(G)$, $\phi \neq \text{Entries}(X) \subseteq \text{UnDominated}(X)$. Consider any $X \in \text{Loops}_{\text{Entries}}(G)$. Assume that X is a maximal strongly connected component of $\mathcal{F}_{\mathcal{H}}^{i}(G)$.

Recall that we assume that the entry vertex of the graph does not have any predecessor and that every vertex in the graph is reachable from the entry vertex of the graph. Hence, it follows that X does not include the entry vertex of the graph and that $\text{Entries}(X)$ must be nonempty.

Let us now show that $\text{Entries}(X) \subseteq \text{UnDominated}(X)$. Consider any $u \in \text{Entries}(X)$. We need to show that u is not strictly dominated by any vertex in $\text{Entries}(X)$. The proof is by contradiction. Assume that u is strictly dominated by some vertex w in $\text{Entries}(X)$. By definition, u has some predecessor $v \notin \text{Entries}(X)$. Now, w must dominate v and, hence, v must be reachable from w in $\mathcal{F}_{\mathcal{H}}^{i}(G)$. (We can establish, using induction, that all vertices in the graph $\mathcal{F}_{\mathcal{H}}^{i}(G)$ continue to be reachable from the graph's entry vertex.) These together imply that v must be in the same strongly connected component as u and w, which contradicts our assumption. The result follows. □

Havlak's Loop Nesting Forest

We will now consider the loop nesting forest defined by Havlak [Havlak 1997]. Given a set of vertices X, let $\text{First}_{DFST}(X)$ denote the singleton set consisting of the vertex in X that is visited first during depth first search (DFS). The Havlak forest is defined to be the forest generated by the scheme described in Section 4.2 using First_{DFST} as the "header" function.

DEFINITION 8. $\text{SimpleHavlaksForest}(G, DFST) = \text{LoopForest}_{\text{First}_{DFST}}(G)$

THEOREM 12. $\text{SimpleHavlaksForest}(G, DFST)$ *is a minimal loop nesting forest for any depth first search tree* $DFST$ *of* G.

PROOF. The function First_{DFST} trivially satisfies the requirements of Theorem 3 and the result follows. □

For a reducible graph, the set of loops in SimpleHavlaksForest$(G, DFST)$ is independent of the depth first search tree and corresponds to the set of loops (intervals) identified by Tarjan's algorithm. For an irreducible graph, however, SimpleHavlaksForest$(G, DFST)$ may depend on the tree $DFST$.

Havlak also outlines a simple extension to his algorithm that constructs a more "refined" loop nesting forest, *i.e.*, a forest containing more loops. For any vertex u, let Dom(u) denote the set of vertices dominated by u. The set of loops identified by the refined algorithm is given by the following definition. (Havlak's construction differs slightly from the following definition, as it *duplicates* vertices that are headers of multiple loops to ensure that a vertex is a header of atmost one loop.)

DEFINITION 9.

$$(a)\text{RefinedHavlakLoops}(G, DFST) = \text{Loops}_{\text{First}_{DFST}}(G) \ \cup \bigcup_{u \in V(G)} SCC(\langle \text{Dom}(u) \rangle)$$

$$(b)\text{RefinedHavlaksForest}(G, DFST) = \langle \text{RefinedHavlakLoops}(G, DFST), \text{First}_{DFST} \rangle$$

THEOREM 13. RefinedHavlaksForest$(G, DFST)$ *is a loop nesting forest, though not necessarily a minimal one, for any depth first search tree* $DFST$ *of* G.

PROOF. It is obvious that RefinedHavlaksForest$(G, DFST)$ satisfies conditions (a), (b), and (d) of Definition 1. We just need to establish (c), the proper nesting property: Consider any loop X with header x and any other loop Y with header y. If neither x nor y is an ancestor of the other in the DFS tree, then X and Y must be disjoint. Otherwise, assume, without loss of generality, that x is an ancestor of y in the DFS tree. It is straightforward to show that if X and Y have a nonempty intersection then X must contain y, and that if X contains y then X must contain Y. The result follows. □

The Sreedhar-Gao-Lee Loop Nesting Forest

We will now consider the algorithm outlined by Sreedhar, Gao and Lee [Sreedhar et al. 1996] for constructing a loop nesting forest. This algorithm is based on a bottom up traversal of the dominator tree. Let *level*(u) denote the depth of a node u in the dominator tree, with the root being at level 0. Let GL_i denote the subgraph of G induced by vertices at a level greater than or equal to i:

$$GL_i = \langle \{u \in V \mid \mathit{level}(u) \geq i\} \rangle$$

Let Dom(u) denote the set of vertices dominated by u.

The set of loops identified by the Sreedhar-Gao-Lee algorithm is given by:

DEFINITION 10.

$$\text{RefinedSGLLoops}(G) = \bigcup_{i>0} SCC(GL_i) \ \cup \bigcup_{u \in V(G)} SCC(\langle \text{Dom}(u) \rangle)$$

The loops in $\bigcup_{u \in V(G)} SCC(\langle \text{Dom}(u) \rangle)$ are referred to as reducible loops, while the loops that are in $\bigcup_{i>0} SCC(GL_i)$ but not in $\bigcup_{u \in V(G)} SCC(\langle \text{Dom}(u) \rangle)$ are referred to as irreducible loops.

A simple modification of the Sreedhar-Gao-Lee algorithm (obtained by simply removing the step for explicitly identifying reducible loops) identifies a smaller set of loops that is of more interest to us.

```
procedure collapse(loopBody, loopHeader)
    for every z ∈ loopBody do
        loop-parent(z) := loopHeader;
        LoopTree.union(z, loopHeader); // Use loopHeader as representative of merged set
    end for

procedure findloop(potentialHeader)
    loopBody = {};
    worklist = { LoopTree.find(y) | y → potentialHeader is a backedge } − {potentialHeader};
    while (worklist is not empty) do
        remove an arbitrary element y from worklist;
        add y to loopBody;
        for every predecessor z of y do
            if (LoopTree.find(z) ∉ (loopBody ∪ {potentialHeader} ∪ worklist)) then
                add LoopTree.find(z) to worklist;
            end if
        end for
    end while
    if (loopBody is not empty) then
        collapse (loopBody, potentialHeader);
        MergedLoops.add(potentialHeader);
    end if

procedure mergeLoopsWithEntryVertex(z)
    t := loop-parent( if MergedLoop.contains(z) then MergedLoops.find(z) else z );
    while (t ≠ NULL) do
        u = MergedLoops.find(t);
        t := loop-parent(u);
        if (t ≠ NULL) then MergedLoops.union (u, t); end if
    end while

procedure processCrossFwdEdges(x)
    for every edge y → z in crossFwdEdges[x] do
        add edge find(y) → find(z) to the graph;
        mergeLoopsWithEntryVertex(z);
    end for

procedure ConstructReducedHavlakForest (G)
    for every vertex x of G do
        loop-parent(x) := NULL; crossFwdEdges[x] := {};
        LoopTree.add(x);
    end for
    for every forward edge and cross edge y → x of G do
        remove y → x from G and add it to crossFwdEdges[least-common-ancestor(y,x)];
    end for
    for every vertex x of G in reverse-DFS-order do
        processCrossFwdEdges(x);
        findloop(x);
    end for
```

Fig. 5. An algorithm for constructing the Reduced Havlak loop nesting forest of a graph. For a union find data structure UF, the function UF.add(z) initially places z in an equivalence class by itself. The function UF.union(u, v) merges u's and v's classes into one, using the representative of the class to which v belongs as the representative element for the merged class. The function UF.find(z) returns the representative element of z's equivalence class. The function UF.contains(z) returns true iff z has been added to UF via the operation UF.add(z).

DEFINITION 11. SimpleSGLLoops$(G) = \bigcup_{i>0} SCC(GL_i)$

It turns out that this simpler loop nesting forest is also one that can be generated using the scheme of Section 4.2. Recall that UnDominated is the function that maps every set of vertices X to the set of vertices in X that are not dominated by any other vertex in X. We can show that

THEOREM 14. SimpleSGLLoops(G) = Loops$_{\text{UnDominated}}(G)$.

PROOF. Straightforward. □

DEFINITION 12. SimpleSGLForest$(G) = \langle$SimpleSGLLoops(G), UnDominated$\rangle$

DEFINITION 13. RefinedSGLForest$(G) = \langle$RefinedSGLLoops(G), UnDominated$\rangle$

THEOREM 15. SimpleSGLForest(G) *is a minimal loop nesting forest for* G.

PROOF. The function UnDominated trivially satisfies the requirements of Theorem 3 and it follows that SimpleSGLForest(G) is a minimal loop nesting forest for G.. □

THEOREM 16. RefinedSGLForest(G) *is a loop nesting forest for* G*,though not necessarily a minimal one.*

PROOF. It is clear that RefinedSGLForest(G) satisfies conditions (a), (b), and (d) of Definition 1. Condition (c), the proper nesting property, follows since the loops identified are maximal strongly connected components of subgraphs that have the proper nesting property. □

An Algorithm for Constructing the Reduced Havlak Forest

In this section we present an almost linear time algorithm for constructing the Reduced Havlak forest. This algorithm is a simple adaptation of the algorithm outlined in [Ramalingam 1999] for constructing Havlak's forest and appears in pseudo-code in Fig. 5. The pseudo-code does not explicitly create the forest data structure defined earlier, but all the relevant information is computed, as explained below.

The algorithm makes use of two union-find data structures, (see Tarjan [Tarjan 1983] or Cormen et al. [Cormen et al. 1990]), each of which is used to maintain a collection of disjoint sets of vertices. Every set S in the collection maintained by the union-find data structure MergedLoops represents a loop in the ReducedHavlak forest, and the elements of the set are the loop's headers. A single element of the set S is used as a representative of the loop, and the operation MergedLoops.find(h) on any element of S returns this representative.

For any non-header vertex z, loop-parent(z) is the representative element of the smallest loop containing z. For any header vertex h of a loop L, the function loop-parent(MergedLoops.find(h)) yields the representative of the smallest loop that properly contains L.

It is straightforward to extend the algorithm to create an explicit representation of the forest.

The second union-find data structure used by the algorithm, LoopTree, maintains the outermost loops that have been identified at any stage. In particular, the find operation on any vertex (using the LoopTree data structure) returns the

(representative) header of the outermost loop identified so far that contains the vertex. This data structure is constructed and used exactly as in Tarjan's original algorithm [Tarjan 1974].

We refer the reader to [Ramalingam 1999] for more details about the algorithm. We note that this algorithm requires the computation of least-common-ancestors over the depth-first search tree. Since the tree is a static (unchanging) tree, this computation can be done efficiently using a simple algorithm (see Tarjan [Tarjan 1979] or Cormen et al. [Cormen et al. 1990, problem 22-3]).