On Loops, Dominators, and Dominance Frontiers

G. Ramalingam
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, 10598, USA
E-mail: `{rama}@watson.ibm.com`

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 [7] and Sparse Evaluation Graphs [5]. The second is a new algorithm for constructing the dominator tree [10] 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.

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 [11]. However, while there is a well accepted notion of what the loops in a reducible graph are [19], there is less agreement about how the loop nesting forest should be defined for arbitrary graphs. For instance, Steensgaard [18], Sreedhar *et. al.* [17], and Havlak [9] each provide a different definition.

Each of these definitions has its advantages and disadvantages. Rather than work with 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 [14], 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 [14], 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). The new algorithms provide compiler implementors more choices. For instance, the ability to construct the SSA form without constructing the dominator tree may provide programming convenience in addition to improving compile-time space requirements by eliminating an intermediate data structure (a possibly significant factor for dynamic and just-in-time compilers).

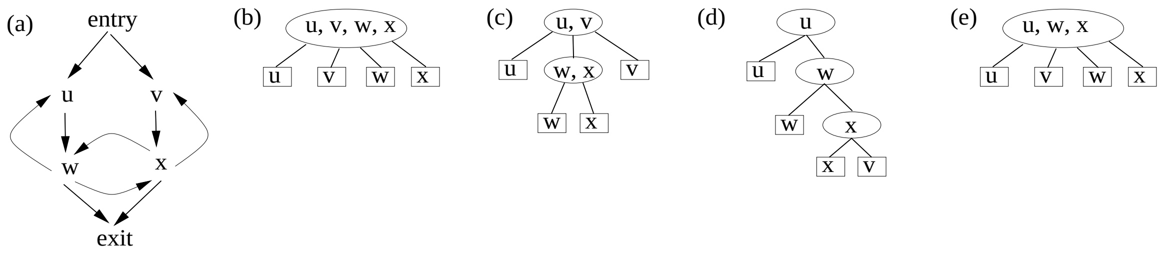

Figure 1: An example illustrating different loop nesting forests for a single control-flow graph, shown in (a). The internal vertices of the forest, shown as ellipses, denote loops, with the loop's headers shown inside the ellipse. The leaves of the forest, shown as rectangles, identify vertices in the control-flow graph. The ancestor-descendant relation in this forest captures the loop containment relation. (b) The Sreedhar-Gao-Lee forest. (c) Steensgaard's forest. (d) Havlak's forest. (e) Reduced Havlak's forest.

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 *entry*(G) has no predecessors and that every vertex in G is reachable from *entry*(G). If Y is a set of vertices in a graph G, then $\langle Y \rangle_G$, the subgraph of G induced by Y, 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.

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. A strongly connected component is said to *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 $x \neq y$. The domination relation can be concisely represented by a tree, called the dominator tree, such that x strictly dominates y iff x is an 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.

3 What's in a Loop

This paper is centered around the use of *loop nesting forests* to solve two problems. Given a structured program, the loops in the program and the nesting relation between them can be easily identified. Given an unstructured program, however, it is no longer obvious what the loops in the program are. Arbitrary programs, including unstructured ones, can be represented by control-flow graphs. Loop identification is a (partial) attempt at identifying the "structure" of a program, given its control-flow graph representation.

The classical algorithm for identifying loops is Tarjan's interval finding algorithm [19], 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 Fig. 1(a). In this example, the Sreedhar-Gao-Lee algorithm [17] identifies a single loop $\{ u, v, w, x \}$, while Steensgaard's algorithm [18] identifies two loops $\{ u, v, w, x \}$ and $\{ w, x \}$, while Havlak's algorithm [9] 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.)

Which of these definitions is the "right" one? It turns out that each of these definitions have their own advantages and disadvantages. The best definition may well depend on the intended application. We will start off by presenting an axiomatic definition of a loop nesting forest that is satisfied by the forest constructed by each of these three algorithms. The algorithms we present later will work correctly with any loop nesting forest that satisfies these axioms.

3.1 Loop Nesting Forests: An Axiomatic Characterization

What we seek is an abstract treatment of loops that will enable results that are applicable to several different loop nesting forests. The key to this is to consider a loop in a graph G to be not a set of vertices, but 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 $2^{V(G)} \times 2^{V(G)}$, where $2^{V(G)}$ denotes the powerset of $V(G)$. A loop nesting forest is a *set of loops*, *i.e.* a subset of $2^{V(G)} \times 2^{V(G)}$. We will sometimes 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. Thus, $\langle \mathcal{B}, \mathcal{H} \rangle$ is 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.

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

The second property we assume of a 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 disjoint or the loop body of one is contained within the loop body of the other. 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 *set of loops* satisfying the various properties described here, but occasionally we will use the term to denote the *forest* that represents such a set (as explained above). In particular, it should be clear what it means for one loop to be the "parent" of another loop.

We also expect the set of loops to be "complete" in some sense. 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. We will soon see how this condition implies "completeness" (see Theorem 1.)

The final property we assume concerns loop headers. We assume that a vertex dominated by some other vertex in the loop cannot be a header of the loop. Given a set of vertices X, let Undom(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 $H \subseteq$ Undom(B).

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: $\forall B \in \mathcal{B}.\ \phi \neq \mathcal{H}(B) \subseteq$ Undom(B).

Let $\mathcal{L}$ be a loop nesting forest for a graph G. We refer to an edge from a vertex in a loop (body) to one of its headers as a *loopback edge* of the loop and the graph. Let $\mathcal{F}_{\mathcal{L}}(G)$ denote the graph obtained by removing all the loopback edges of G.

Theorem 1 *If $\mathcal{L}$ is a loop nesting forest for G then, $\mathcal{F}_{\mathcal{L}}(G)$ is an acyclic graph.*

Proof Omitted. $\Box$

Thus, a loop nesting forest lets us decompose a graph into an acyclic graph and a set of loopback edges. The above theorem illustrates 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 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.

The above definition was partially guided by the applications we present later. The definition is general enough to include various specific forests that have been previously defined, while strong enough for our intended application and to establish several interesting properties.

3.2 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 [18] and Havlak [9], by abstracting the header function.

Consider any function $\mathcal{H} : 2^{V(G)} \rightarrow 2^{V(G)}$ such that for every non empty X, $\phi \neq \mathcal{H}(X) \subseteq \text{Undom}(X)$. We will show that this identifies a collection of loops (with $\mathcal{H}$ as its header function) satisfying Definition 1.

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

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

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

Proof Omitted. □

Conversely, every minimal loop nesting forest can be generated using the above scheme. Note that the above scheme can be used even if $\mathcal{H}$ is a partial function, as long as $\mathcal{H}$ is defined for the elements of $\text{Loops}_{\mathcal{H}}(G)$. It can be shown that if $\mathcal{L} = \langle \mathcal{B}', \mathcal{H}' \rangle$ is a minimal loop nesting forest for G, then $\mathcal{B}'$ must be $\text{Loops}_{\mathcal{H}'}(G)$.

3.3 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 definition by choosing 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. Let $\mathcal{E}(X)$ denote the set of all entry vertices of X. Steensgaard's forest [18] is the forest $\text{LoopForest}_{\mathcal{E}}(G)$ obtained as a special case of our definition, by identifying the headers of a loop to be its entry vertices. It follows from Theorem 2 that Steensgaard's loop nesting forest is a minimal loop nesting forest.

The loops identified by Havlak's algorithm [9] depend on the order in which vertices are visited during DFS (depth-first search). Given a DFS tree $DFST$ and 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 DFS. The Havlak forest is defined to be $\text{LoopForest}_{\text{First}_{DFST}}(G)$ Clearly, Havlak's forest is also a minimal loop nesting forest. (Havlak also outlines a simple extension to his algorithm that constructs a more "refined" forest with more loops. It can be shown that this refined forest is a loop nesting forest according to our definition, though not a minimal one.)

The definition presented in Section 3.2 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 [9, 14].

Sreedhar, Gao and Lee [17] outline an algorithm, based on a bottom up traversal of the dominator tree, for identifying the loops in a graph. We can show that the algorithm constructs a loop nesting forest (satisfying Definition 1) but not a minimal one. However, the set of loops constructed by this algorithm consist of a primary set of loops as well as an additional set of "reducible" loops. It turns out that the set of primary loops identified by the Sreedhar, Gao, and Lee algorithm is, in fact, equal to the minimal loop nesting forest $\text{LoopForest}_{\text{Undom}}(G)$, the forest generated by the scheme of Section 3.2 with Undom as the header function. The Sreedhar, Gao, and Lee algorithm has a worst case quadratic complexity but can be improved to run in almost linear time [14].

The appendix illustrates how the header functions described above along with the construction of Section 3.2 yields the different loop forests for the example in Fig. 1.

3.4 A New Loop Nesting Forest

Our goal is to explore the use of loop nesting forests in solving the two problems of computing iterated dominance frontiers and constructing the dominator tree. The approach we take is to generalize simple algorithms that work for *acyclic* graphs by showing how a problem instance involving an *arbitrary* graph can be transformed into equivalent problem instance over an *acyclic* graph, given *any* minimal loop nesting forest for the graph.

Though our approach can be used with any minimal loop nesting forest in principle, not all minimal loop nesting forests are equally appropriate for use. In fact, all the three forests we have looked at so far turn out to be unattractive for the applications we consider.

Steensgaard's 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. Unfortunately, the size of the transformed acyclic graph generated by our approach can be quadratic in the size of the initial graph if we use Havlak's forest in 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 forest without such disadvantages, by merging loops that have a common entry vertex.

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 $H_c \subseteq \text{Undom}(B_p)$, 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 Omitted. □

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 3 *The Reduced Havlak forest is a minimal loop nesting forest.*

Proof Follows from repeated application of Lemma 1. □

The algorithm outlined by Ramalingam [14] for generating Havlak's forest can be adapted to construct the Reduced Havlak forest in almost linear time. (Details omitted due to lack of space.) Fig. 1(e) illustrates the Reduced Havlak forest of the graph in Fig. 1(a).

4 Constructing the Dominator Tree

We now present our first application of loop nesting forests, an algorithm for constructing the dominator tree of a control-flow graph.

4.1 Acyclic Graphs

Consider an acyclic graph. Let u be a vertex with predecessors $v_1, \cdots, v_k$. 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

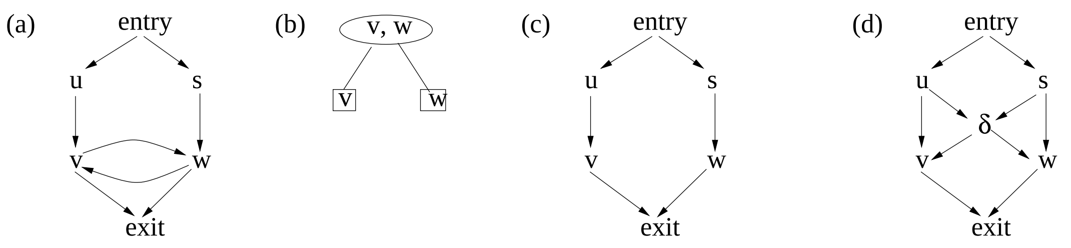

Figure 2: (a) An irreducible graph G. (b) A loop nesting forest $\mathcal{L}$ for G. (c) The acyclic subgraph $\mathcal{F}_{\mathcal{L}}(G)$. (d) The transformed graph $\Psi_{\mathcal{L}}(G)$.

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 of v_1 through v_k in the dominator tree. This suggests the following algorithm for constructing the dominator tree of an acyclic graph incrementally by processing the vertices in topological sort order.

```
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 the predecessors of $u$;
        add $u$ as a child of $z$ in DomTree ;
    end for
```

Gabow [8] describes a tree data structure in which the operations of adding a new vertex to the tree and finding the least common ancestor of two vertices can both be done in constant time. The above algorithm runs in linear time if we utilize Gabow's data structure to implement `DomTree`. The algorithm described above appears in [15] and [2].

4.2 Reducible Graphs

An edge $u \to v$ is said to be a *backedge* if v dominates u. Removing the backedges of a graph does not change the 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. The backedges of a reducible graph can be identified easily (in linear time) using depth first search. Consequently, the dominator tree algorithm described above for acyclic graphs can be trivially adapted to handle reducible control-flow graphs[2].

4.3 Irreducible Graphs

Our goal is to show how loop nesting forests can be used to extend the above algorithm to handle cyclic, possibly irreducible, graphs. As we have seen earlier (Theorem 1) a loop nesting forest $\mathcal{L}$ of a graph G identifies a set of edges, the loopback edges, whose deletion yields an acyclic graph $\mathcal{F}_{\mathcal{L}}(G)$. The trouble is that, 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 irreducible graph G in Fig. 2. Both Steensgaard's forest and the Sreedhar-Gao-Lee forest of this graph are the same, which is shown in Fig. 2(b). The forest consists of a single loop $\{v, w\}$ with two loop headers v and w. The acyclic subgraph $\mathcal{F}_{\mathcal{L}}(G)$ obtained by eliminating the loopback edges is shown in Fig. 2(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 if we use Havlak's forest. In general, the graph $\mathcal{F}_{\mathcal{L}}(G)$ may even contain vertices unreachable from $entry(G)$. (The Sreedhar-Gao-Lee forest of the graph in Fig. 1(a) illustrates this phenomenon.)

The solution is that we delete the loopback edges, but add some other edges that compensate for the deleted edges without creating cycles in the graph. A vertex outside a loop is said to be 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 $u \to h$ 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 loop. We can compensate for this by adding edges from every pre-entry vertex of the loop to every header of the loop. If we apply the transformation to every loop in the graph, we end up with an acyclic graph that has the same dominator tree as the original graph. Unfortunately, this transformation can add a quadratic number of edges 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 blowup in the number of edges, we use a modified transformation that achieves the same effect, as follows: 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 in Fig. 2. The headers of the loop in this graph are v and w. The pre-entry vertices of this loop are u and s. Our transformation deletes the loopback edges $w \rightarrow v$ and $v \rightarrow w$, but compensates for them by adding a vertex δ and the edges $u \rightarrow \delta$, $s \rightarrow \delta$, $\delta \rightarrow v$, and $\delta \rightarrow w$. This gives us the acyclic graph in Fig. 2(d).

We actually use a slightly more general transformation than the above. 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 ends up being undesirable in the context of our second application (Section 5). So, in addition to the edges involving the dummy header δ_L, we need to add 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. Recall that $\mathcal{E}(X)$ 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:*

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

Let $\mathcal{L}$ be a minimal loop nesting forest for G. We define $\Psi_{\mathcal{L}}(G)$ to be the graph obtained by applying the above transformation to every loop L in $\mathcal{L}$, where outer loops are transformed before inner loops. (Actually, the above transformation can 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 improvements here for the sake of simplicity.) Note that if L is an outermost loop, then the transformation Ψ_L does not modify the set of entry or pre-entry vertices of any other loop in the graph. The above definition of $\Psi_{\mathcal{L}}$ is not the simplest possible, but the interpretation of $\Psi_{\mathcal{L}}$ as the sequential composition of a series of transformations of the form Ψ_L simplifies proving properties of $\Psi_{\mathcal{L}}$ by proving them for the simpler single loop transformations of the form Ψ_L.

What is the size of the transformed graph $\Psi_{\mathcal{L}}(G)$? If every vertex is the entry vertex of at most one loop in the forest $\mathcal{L}$ (this is true of Steensgaard's forest, the Sreedhar-Gao-Lee forest, as well as the Reduced Havlak forest), then the size of the graph $\Psi_{\mathcal{L}}(G)$ cab be shown to be 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.

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 $\Delta_{\mathcal{L}}$ denote the set $\{\delta_L \mid L \in \mathcal{L}\}$.

Theorem 4 *If $\mathcal{L}$ is a minimal loop nesting forest for G, then* DomTree$(G) =$ DomTree$(\Psi_{\mathcal{L}}(G)) - \Delta_{\mathcal{L}}$.

Proof Omitted. □

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 4.1 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.

5 Computing the Iterated Dominance Frontier

We now present a second application of loop nesting forests, an algorithm for constructing the iterated dominance frontier of a set of vertices.

5.1 The Problem

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 $\alpha : u \to^* w$ and $\beta : v \to^* w$ such that the only vertex common to the two paths is the vertex 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. 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 $J_G^+(X)$. The subscript G will be occasionally omitted.

For any set X that includes the entry vertex of the graph, Cytron *et al.* [7] show that $J^+(X) = DF^+(X)$. Weiss [20] establishes the stronger result that $J(X) = DF^+(X)$. The iterated dominance frontier is directly useful in computing sparse evaluation graphs and the SSA form. We refer the reader to [7] for further details.

5.2 Acyclic Graphs

Consider a set of vertices X in an acyclic graph G. Let us refer to elements of $X \cup \{\ \mathit{entry}(G)\ \} \cup J_G^+(X \cup \{\ \mathit{entry}(G)\ \})$ as "definitions". A definition d is said to "reach" another node u if there is a path from d to u not containing any definition other than d or u. If at least two definitions reach u, then u must be in $J_G^+(X \cup \{\ \mathit{entry}(G)\ \})$. In other words, vertices not in $J_G^+(X \cup \{\ \mathit{entry}(G)\ \})$ must have exactly one reaching definition. (Recall that every vertex in the graph is reachable from *entry*(G).) 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 resulting algorithm runs in linear time, if appropriate implementations of the various sets is used. This algorithm is not new; it appears in [6] and [13].

```
function ComputeIteratedDominanceFrontier(G: Acyclic Graph, $X$ : Subset of V(G))
    ItDomFr := {};
    for every vertex $u$ of G in topological sort order do
        reachingDefs := {};
        for every predecessor $v$ of $u$ do
            if $v \in (\mathtt{ItDomFr} \cup X \cup \{entry(G)\})$
                then add $v$ to reachingDefs;
                else add uniqueReachingDef($v$) to reachingDefs;
        end for
        if |reachingDefs| = 1
            then uniqueReachingDef($u$) := the only element of reachingDefs;
            elsif ($u \neq entry(G)$) then add $u$ to ItDomFr;
    end for
    return ItDomFr;
```

5.3 Reducible Flow Graphs

How can the above algorithm be extended to handle reducible graphs? A reducible graph G can be decomposed into an acyclic graph $\mathcal{F}(G)$ and a set of backedges. The contribution of backedges to the iterated dominance frontier of a set of vertices can be identified using the graph's loop nesting forest. Consider the following observation. If a vertex x is contained in a loop, then the iterated dominance frontier of x will include the entry vertex of the loop. 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 X includes $HLC(X)$. In fact, it turns out that

Theorem 5 $DF_G^+(X) = HLC(X) \cup DF_{\mathcal{F}(G)}^+(X \cup HLC(X))$.

This shows how the algorithm outlined earlier for acyclic graphs can be utilized to compute $DF_G^+(X)$ using the loop nesting forest. In particular, the set of loops containing any given vertex is given by the ancestors of the vertex in

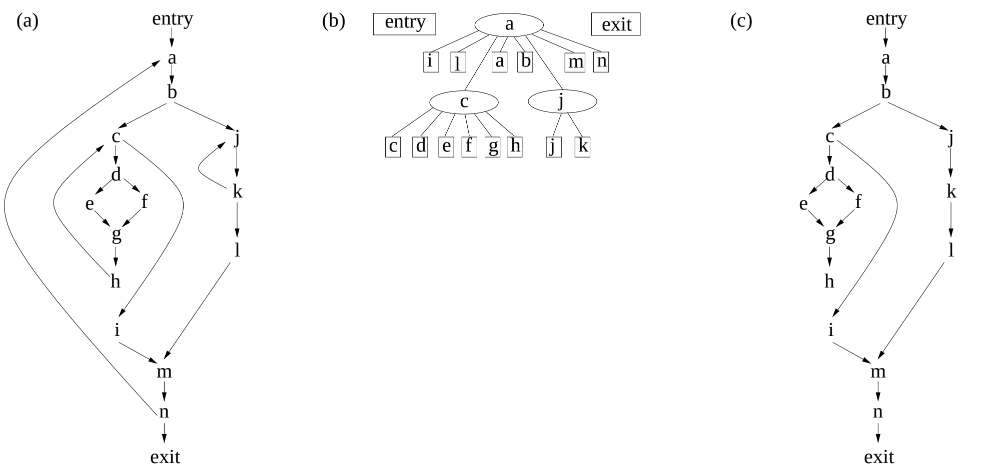

Figure 3: (a) A reducible control-flow graph G. (b) Its loop nesting forest. (c) The acyclic graph obtained by dropping the backedges of G.

the loop nesting forest. 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 earlier. Theorem 5 then yields us the iterated dominance frontier of X in G.

For example, assume we want to compute the iterated dominance frontier of vertex e in the graph in Fig. 3(a). The loop nesting forest of the graph is shown in Fig. 3(b). $HLC(e)$, which is obtained by "walking up" this forest from e, is $\{c, a\}$. Fig. 3(c) shows the acyclic graph $\mathcal{F}(G)$ obtained by dropping all backedges of G. To identify the iterated dominance frontier of e in G, we first identify the iterated dominance frontier of $\{e\} \cup HLC(e) = \{e, a, c\}$ in $\mathcal{F}(G)$. By applying our earlier algorithm to the acyclic graph, we determine that the iterated dominance frontier of $\{e, a, c\}$ in $\mathcal{F}(G)$ is $\{g, m\}$. Hence, the iterated dominance frontier of 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 [19], this gives us a simple almost linear time for computing the iterated dominance frontier of a set of vertices in a reducible graph. This approach for reducible graphs is used by Cytron *et al.* [6].

5.4 Irreducible Graphs

We will now see how the ideas described so far can be extended to deal with irreducible graphs. A loop nesting forest $\mathcal{L}$ for a graph G identifies a set of loopback edges whose deletion yields an acyclic graph $\mathcal{F}_{\mathcal{L}}(G)$. Generalizing our earlier definition of $HLC(X)$, we define $HLC_{\mathcal{L}}(X)$ to be the set of *headers* of *loops containing* some vertex in X.

However, just replacing $\mathcal{F}(G)$ by $\mathcal{F}_{\mathcal{L}}(G)$ and $HLC(X)$ by $HLC_{\mathcal{L}}(X)$ in Theorem 5 does not yield a valid theorem. Consider the example in Fig. 2(a). The iterated dominance frontier of u in the graph of Fig. 2(a) includes vertices v and w, thanks to the loopback edges $v \to w$ and $w \to v$. The contribution of these edges to the iterated dominance frontier of u is, however, *not* captured by the loop containment relation. In particular, u is not contained in any loop, and $HLC(u)$ is empty. 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 Fig. 2(c). The iterated dominance frontier of u in this acyclic graph *does not include* v and w! Thus, Theorem. 5 fails to hold true in irreducible graphs if we just replace $\mathcal{F}(G)$ by $\mathcal{F}_{\mathcal{L}}(G)$ and $HLC(X)$ by $HLC_{\mathcal{L}}(X)$.

The solution is the same one we used in Section 4.3: when we delete the loopback edges, we add other edges that compensate for the deleted edges without creating cycles in the graph. Consider the example in Fig. 2(a). The headers of the loop in this graph are v and w. The pre-entry vertices of this loop are u and s. Our transformation deletes the 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 Fig. 2(d). It can be verified that the iterated dominance frontier of u in the transformed graph does include both v and w.

More generally, we can 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 $\Psi_{\mathcal{L}}(G)$, (Recall the definition of $\Psi_{\mathcal{L}}(G)$ from Section 4.3).

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

In fact, the above equation can be simplified. 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 $\{\delta_L \mid L \in \mathcal{L}$ and $X \cap \mathcal{B}(L) \neq \phi\}$. Thus, $\Delta_{\mathcal{L}}(X)$ is the set of dummy headers added for loops that overlap X. Define $\Delta_{\mathcal{L}}$ to be $\{\delta_L \mid L \in \mathcal{L}\}$.

Theorem 6 *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 Omitted. □

Theorem 6 shows how the algorithm of Section 5.2 can be adapted to work for arbitrary graphs using loop nesting forests. Given any set X of vertices in a graph G, we walk up any loop nesting forest $\mathcal{L}$ of G to identify $\Delta_{\mathcal{L}}(X)$. We then apply the algorithm of Section 5.2 to the set $X \cup \Delta_X(\mathcal{L})$ 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.)

6 Related Work

Loops play a fundamental role in several loop optimizations and transformations [11]. The classical algorithm for identifying loops is Tarjan's interval finding algorithm [19], 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 [18], Sreedhar *et. al.* [17], and Havlak [9]. Ramalingam [14] improves upon the efficiency of these different algorithms.

The concepts of domination and dominator tree have many uses in program analysis and optimization. The standard algorithm for computing dominators is Lengauer and Tarjan's [10] almost linear time algorithm. Alstrup *et al.* [3, 1] and Buchsbaum *et al.* [4] present linear time algorithms for this problem.

The concept of the iterated dominance frontier was brought to prominence by the work of Cytron *et al.* [7] on the SSA form, a data structure with numerous applications in program optimization. The Cytron *et al.* algorithm for computing the iterated dominance frontier of a set of vertices takes quadratic time in the worst case, but is competitive with other linear time algorithms [12, 16] for this problem in practice.

7 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. Though linear time algorithms are already known for these problems we believe that these new algorithms are interesting because of the approach we take.

In particular, we have utilized these two applications as a vehicle for understanding the concepts of loops and loop nesting forests. 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 definition. We have shown how these 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 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.

Acknowledgements

I would like to thank John Field, V. C. Sreedhar and Mark Wegman for their feedback on this paper.

References

[1] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM J. Comput.* To appear.

[2] Stephen Alstrup and Peter W. Lauridsen. A simple and optimal algorithm for finding immediate dominators in reducible graphs. Technical Report D-261/96, TOPPS Bibliography, 1996.

[3] Stephen Alstrup, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. Technical Report 96/35, Department of Computer Science, University of Copenhagen, 1996.

[4] Adam L. Buchsman, Haim Kaplan, Anne Rogers, and Jeffrey Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296, 1998.

[5] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 55–66, 1991.

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

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):452–490, October 1991.

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

[9] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[10] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[11] Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, MA, 1998.

[12] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[13] G. Ramalingam. On sparse evaluation representations. In *Fourth International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1997.

[14] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, 1999.

[15] G. Ramalingam and Thomas Reps. 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*, pages 287–298, 1994.

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

[17] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, November 1996.

[18] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, Wash., October 1993.

[19] Robert E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9:355–365, 1974.

[20] Michael Weiss. The transitive closure of control dependence: The iterated join. *ACM Letters on Programming Languages and Systems*, 1(2):178–190, 1992.

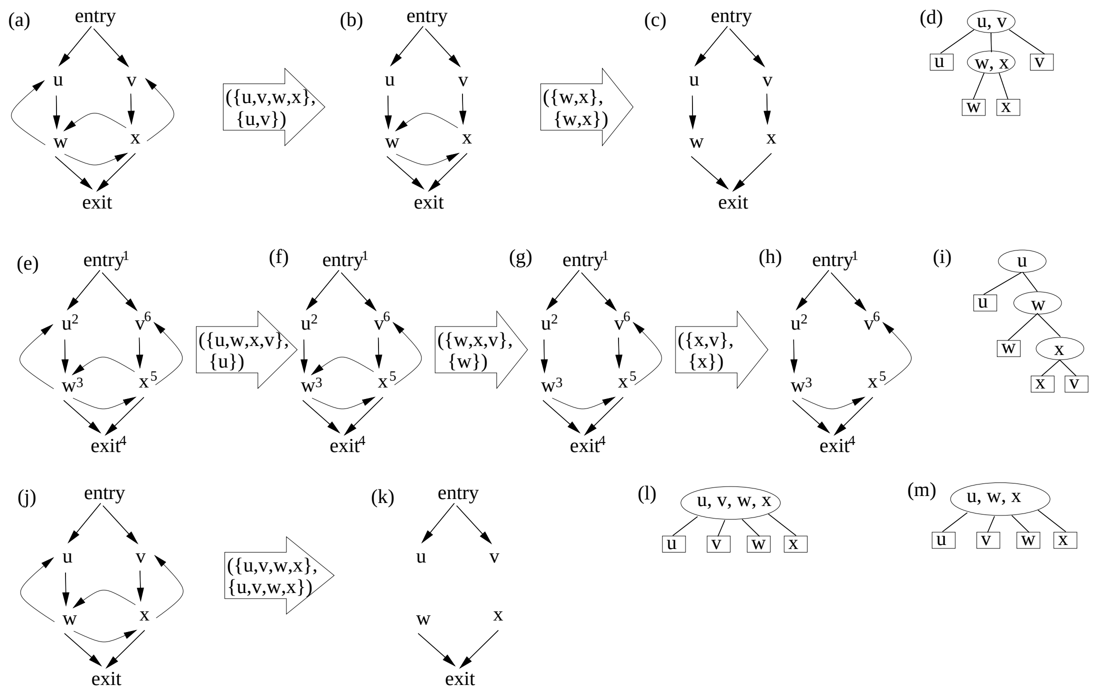

Figure 4: An example illustrating the different loop nesting forests. (a)-(d) Steensgaard's forest. (e)-(i) Havlak's forest. (j)-(l) The Sreedhar-Gao-Lee forest. (m) The Reduced Havlak forest.

Appendix: An Example of the Different Loop Nesting Forests

The appendix illustrates using an example how the different loop nesting forests are obtained.

Steensgaard's forest is obtained as a special case of the scheme described in Section 3.2 by selecting the entry vertices of a loop to be its headers. Consider the graph in Fig. 4(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 Fig. 4(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 Fig. 4(c). As this graph is acyclic, the process halts, yielding the loop nesting forest in Fig. 4(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.

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 DFS. The Havlak forest is the forest generated by the scheme of Section 3.2 using First_{DFST} as the "header" function. Consider the graph in Fig. 4(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 Fig. 4(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 Fig. 4(i).

We define the Sreedhar-Gao-Lee forest [17] to be the forest generated by the scheme of Section 3.2 using Undom as the "header" function. (The algorithm presented by Sreedhar, Gao, and Lee actually identifies an additional set of "reducible" loops as well; adding these loops to the forest yields a loop nesting forest that is not minimal). The Sreedhar-Gao-Lee forest of the graph in Fig. 4(j) consists of only one loop, namely $\{u, v, w, x\}$, and is shown in Fig. 4(l).

Finally, Fig. 4(m) illustrates our new loop nesting forest (the Reduced Havlak forest), obtained by merging together the loops in Havlak's forest (shown in Fig. 4(i)) that have a common entry vertex. In this example, all the three loops in Havlak's forest share a common entry vertex, namely v.