Finding Regions Fast:
Single Entry Single Exit and
Control Regions in Linear Time*

Richard Johnson  
David Pearson  
Keshav Pingali  

TR 93-1365  
July 1993  

Department of Computer Science  
Cornell University  
Ithaca, NY  14853-7501  

*This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation. David Pearson is supported by a Fannie and John Hertz Fellowship.
Finding Regions Fast:  
Single Entry Single Exit and Control Regions in Linear Time

Richard Johnson  David Pearson  Keshav Pingali

Department of Computer Science
Cornell University, Ithaca, NY 14853
Report # CTC93TR141
July 1993

Abstract
Many compilation problems require computing the control dependence equivalence relation which divides nodes in a control flow graph into equivalence classes such that nodes are in the same class if and only if they have the same set of control dependences. In this paper, we show that this relation can be computed in $O(E)$ time by reducing it to a naturally stated graph problem: in a strongly connected component, divide nodes into equivalence classes such that every cycle passes through all or none of the nodes in an equivalence class. Our algorithm does not require the computation of the control dependence relation of the postdominator relation — in fact, it runs faster in practice than the best algorithms for either of these problems. We also show that our algorithm can be used to determine the single entry single exit regions of a control flow graph in $O(E)$ time.

1 Introduction

The notion of control dependence plays an important role in optimization and parallelization. Intuitively, a node $n$ is control dependent on a node $c$ if $c$ determines whether $n$ is executed. For example, in a conditional statement, statements on the true and false sides of the conditional are control dependent on the predicate. Since statements from different sides are control dependent on the predicate in a complementary sense, it is useful to associate a direction with control dependence; for example, we can say that statements on the true side of a conditional statement are control dependent on the predicate in the true direction. In the presence of nested control structures, multiway branches and unstructured flow of control, intuition is an unreliable guide and it is better to rely on a formal, graph theoretic notion of control dependence defined as follows.

Definition 1 A control flow graph $G$ is a graph with distinguished nodes START and END such that every node is reachable from START and END is reachable from every node in $G$. START has no predecessors and END has no successors.

To simplify the discussion, we will follow standard practice and assume that there is an edge from START directly to END in the control flow graph.

Definition 2 A node $p$ is said to postdominate a node $n$ if every path from $n$ to END contains $p$.

\footnote{This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation. David Pearson is supported by a Fannie and John Hertz Fellowship.}
By convention, a node postdominates itself. Control dependence can now be defined as follows [FOW87]:

**Definition 3** A node \( n \) is said to be control dependent on node \( c \) with direction \( l \) if there is a path from \( c \) to \( n \) beginning with edge \( l \) such that

1. \( n \) postdominates all nodes other than \( c \) on this path, and
2. if \( n \) and \( c \) are distinct nodes, \( n \) does not postdominate \( c \).

Although a control dependence is usually written as a tuple consisting of a node and a direction, the node is simply the source of the direction edge, so we will often leave the node implicit in our presentation. In the worst-case, the control dependence relationship can have size \( O(NE) \), where \( N \) and \( E \) are the number of nodes and edges respectively in the control flow graph. Figure 1 shows a small irreducible program and the control dependences of its nodes.

In many applications, the full control dependence relation is not needed — instead, we only want to know if two nodes have the same set of control dependences. One such application, discussed in more detail in Section 5, is the determination of single entry single exit (SESE) regions of a control flow graph, which in turn can be used to build the dependence flow graph or SSA form of a program [JP93]. These representations permit efficient optimization of programs. Another application of the control dependence equivalence relation is in global scheduling of instructions for pipelined machines [GS87]. Techniques for basic block scheduling can be extended immediately to a set of basic blocks with identical control dependences, since all instructions in such a set can be pooled together and scheduled subject only to data dependences. This increases the scope of the scheduling beyond basic blocks without introducing the complications of speculative execution.

Our goal then is to divide nodes into equivalence classes in which two nodes are in the same class if and only if they have the same set of control dependences. These equivalence classes have been called control dependence regions in the literature [FOW87]. In Figure 1, nodes \( c \) and \( f \) are in the same region, while all the other nodes are in regions containing only themselves. Note also that nodes \( c \) and \( f \) enclose a SESE region. We will use \( CD_\equiv \) to stand for the control dependence equivalence relation.

Cunam, Ferrante, and Sarkar [CFS90] gave an \( O(EN) \) time, \( O(E + N) \) space algorithm for computing \( CD_\equiv \). Briefly, this algorithm works by placing all nodes in a single equivalence class and then repeatedly refining the equivalence relation by considering the effect of each control dependence on the existing partition. In the worst-case, the algorithm performs \( O(N) \) work for each of \( O(E) \) control dependences. The problem with this approach is that \( CD_\equiv \) is defined in terms of the control dependence relation, which is \( O(EN) \) size in the worst-case.

In this paper, we show how to compute \( CD_\equiv \) in \( O(E) \) time. To accomplish this, we reduce the problem to one of computing a simple graph property that we call cycle equivalence: two nodes are cycle equivalent
in a strongly connected component iff for all cycles \( C \), either \( C \) contains both nodes or neither node. In this paper, we describe a simple linear time algorithm based on depth-first search for solving the cycle equivalence problem, thereby constructing the \( CD_\equiv \) relation in linear time\(^2\). Ball [Bal92] has also recognized the need to characterize \( CD_\equiv \) without using control dependence and has developed a linear time algorithm for computing \( CD_\equiv \). However, his algorithm works only for reducible graphs, and requires computation of both dominators and postdominators. We will show in Section 4 that computing just dominators or postdominators takes longer than computing \( CD_\equiv \) using our algorithm.

The rest of the paper is organized as follows. In Section 2, we reduce the problem of computing \( CD_\equiv \) to that of cycle equivalence. In Section 3, we reduce the cycle equivalence problem in directed graphs to an equivalent problem in undirected graphs. The main motivation for this reduction is that depth-first search algorithms in undirected graphs are simpler than those in directed graphs since there are no cross edges or forward edges to consider in the case of undirected graphs. In Section 4, we give a simple algorithm based on depth-first search for solving the cycle equivalence problem in undirected graphs. We also present experimental results that show that our algorithm for \( CD_\equiv \) is faster than computing postdominance, the first step of previous algorithms for control dependence and control dependence equivalence. In Section 5, we show how \( CD_\equiv \) can be used to determine the single entry single regions of a control flow graph in \( O(E) \) time. Section 6 concludes the paper with a list of open problems related to this work.

## 2 Control dependence equivalence is cycle equivalence

In this section, we define cycle equivalence formally and show that the problem of control dependence equivalence can be reduced to cycle equivalence.

**Definition 4** In a graph \( G \), two nodes are cycle equivalent if they occur in the same set of cycles.

Viewed as a binary relation on nodes, it is clear that the relation is reflexive, symmetric and transitive. We will call the equivalence relation \( CYC_\equiv \). Note that cycle equivalence is defined for both directed and undirected graphs. To reduce \( CD_\equiv \) to \( CYC_\equiv \), we need the following lemma.

**Lemma 1** Let \( a, b \) and \( p \) be nodes in \( G \) where \( b \) is distinct from \( a \) and \( p \). Suppose that \( a \) is control dependent on node \( p \) with direction \( l \) but \( b \) is not. Then there must be a path \( p \leftrightarrow a \) that does not contain \( b \).

**Proof:** Let \( f \) be the destination of edge \( l \). If \( f \) is \( a \) itself, the result is established.

Otherwise, \( f, a \) and \( b \) are distinct nodes. Suppose every path from \( f \) to \( a \) passes through \( b \). We show that this implies that \( b \) is control dependent on node \( p \) with direction \( l \), leading to a contradiction. Since \( a \) postdominates \( f \), every path \( f \downarrow END \) can be written as \( f \downarrow b \downarrow \downarrow a \downarrow END \), which in turn can be written as \( f \downarrow \downarrow b \downarrow \downarrow a \downarrow \downarrow END \) because every path from \( f \) to \( a \) passes through \( b \). Therefore, \( b \) postdominates \( f \). Moreover, this implies that \( a \) postdominates \( b \); since \( a \) does not postdominate \( p \), this means \( b \) does not postdominate \( p \). Therefore, \( b \) is control dependent on \( p \), which is incorrect. Therefore, there must be a path \( p \leftrightarrow a \) that does not pass through \( b \).

We now prove the key result that reduces \( CD_\equiv \) to \( CYC_\equiv \).

**Theorem 1** Let \( S \) be the strongly connected component constructed by adding an edge \( END \to START \) to a control flow graph \( G \). Nodes \( a, b \) in \( G \) have the same set of control dependences iff \( a, b \) are cycle equivalent in \( S \).

---

\(^2\)This algorithm was sketched in an earlier paper [JP93] but there was not enough room in the margin for a complete description and proof.
Proof: (by contradiction)

1. We show that if there is a cycle in $S$ containing $a$ but not $b$, then $a$ and $b$ do not have the same control dependences. This is done by showing that any such cycle must contain a node $p$ (possibly $a$ itself) such that $a$ is control dependent on $p$, but $b$ is not.

If $a$ postdominates all nodes on the cycle, then there must be a path $a \xrightarrow{\perp} \text{END}$ that does not pass through any of the nodes in the cycle other than $a$, and $a$ is control dependent on itself with some sense $l$ where $f$, the destination of edge $l$, is on the cycle. If $a$ does not postdominate all nodes on the cycle, there must be a node $p$ distinct from $a$ such that $a$ does not postdominate $p$ but $a$ postdominates all nodes in the cycle strictly between $p$ and $a$ (trace backwards from $a$ in the cycle to find such a node). Then $a$ is control dependent on $p$ with some sense $l$ where $f$, the destination of edge $l$, is on the cycle.

For $b$ to have the same control dependence, $b$ must postdominate $f$. The portion of the cycle $f \xrightarrow{\perp} p$ can be extended with any path $p \xrightarrow{\perp} \text{END}$ to give a path from $f$ to $\text{END}$. Since $b$ is not on the cycle, it follows that $b$ must occur on every path $p \xrightarrow{\perp} \text{END}$; so $b$ postdominates $p$ and cannot be control dependent on $p$.

We conclude that if $a$ and $b$ have the same control dependences in $G$, they are cycle equivalent in $S$.

2. We show that if $a$ and $b$ do not have the same set of control dependences, then there must be a cycle passing through one that does not pass through the other. Suppose $a$ is control dependent on $p$ with direction $l$ but $b$ is not. Let $f$ be the destination of edge $l$. There are four cases.

(a) $a$ dominates $p$, $b$ dominates $p$. Note that $b$ and $p$ or $a$ and $p$ may be the same node.

If there is some path (possibly empty) from $a$ to $p$ that does not contain $b$, $b$ is distinct from $a$ and $p$, and by Lemma 1, we have a cycle $a \xrightarrow{\perp} p \xrightarrow{\perp} A$ containing $a$ but not $b$. Otherwise, every path from $a$ to $p$ contains $b$. Therefore, $a$ dominates $b$ and there is a path $\text{START} \rightarrow a$ that does not pass through $b$. $b$ cannot postdominate $a$; otherwise, $b$ would be control dependent on $p$ with direction $l$. Therefore, there is a path $a \xrightarrow{\perp} \text{END}$ that does not contain $b$. Thus we can construct a $b$-free path $\text{START} \rightarrow a \rightarrow \text{END}$; together with backedge $\text{END} \rightarrow \text{START}$, this gives a cycle containing $a$ but not $b$.

(b) $a$ dominates $p$, $b$ does not dominate $p$. So $b$ and $p$ must be distinct nodes.

There must be a path $\text{START} \rightarrow a \rightarrow p$ that does not pass through $b$. From Lemma 1, there is a path $p \xrightarrow{\perp} a$ that does not pass through $b$. Therefore, we have a cycle $a \rightarrow p \rightarrow a$ that does not contain $b$.

(c) $a$ does not dominate $p$, $b$ dominates $p$.

There must a path $\text{START} \rightarrow p$ that contains $b$ since $b$ dominates $p$. Since $a$ is control dependent on $p$, there is a path from $p$ to $\text{END}$ that does not contain $a$. Therefore, there is a path from $\text{START}$ to $\text{END}$ that contains $b$ but not $a$.

(d) $a$ does not dominate $p$, $b$ does not dominate $p$. So $a$, $b$ and $p$ are distinct nodes.

Since $a$ is control dependent on $p$ with direction $l$ but $b$ is not, either (i) $b$ does not postdominate $f$, or (ii) $b$ postdominates $p$.

Suppose (i) is true. Since $b$ does not dominate $p$, there is a path $\text{START} \rightarrow p$ that does not contain $b$. There is a path $p \rightarrow f$ that does not contain $b$. Since $b$ does not postdominate $F$, there is a path $p \rightarrow \text{END}$ that does not contain $b$; however, this path contains $a$ since $a$ postdominates $f$. Therefore, there is a path $p \xrightarrow{\perp} \text{END}$ that does not contain $b$.

Suppose (ii) is true. There is a path $p \rightarrow b \rightarrow \text{END}$ that does not contain $a$. Since there is a path $\text{START} \rightarrow p$ that does not contain $a$, we have a path $p \xrightarrow{\perp} \text{END}$ that does not contain $a$.

$\square$
We leave it to the reader to verify that Theorem 1 holds for the program of Figure 1. We show how to compute $\text{CYC}_\text{m}$ next.

3 From directed to undirected graphs

Rather than compute $\text{CD}_\text{m}$ directly on a strongly connected component, we reduce the problem to that of finding cycle equivalence in an undirected graph. The motivation is that algorithms based on depth-first search of undirected graphs are simpler than those using directed graphs since cross edges and forward edges are eliminated.

Let $S$ be a strongly connected component. Create a corresponding undirected graph $U$ as follows.

1. Expand $S$ into $S'$: split each node $n$ into nodes $n_i, n'$, and $n_o$ such that inedges to $n$ now flow to $n_i$, outedges from $n$ flow from $n_o$, and there are directed edges $n_i \rightarrow n'$ and $n' \rightarrow n_o$.

2. Undirect edges in $S'$ to form $U$.

Figure 2(a) shows the node expansion step pictorially. To avoid cluttering the diagram, we have simply used triangles to represent new nodes introduced by node expansion. The directions of the triangles differentiate in-nodes from out-nodes. Figure 2(b) illustrates node expansion for the program of Figure 1. This figure also shows a DFS spanning tree for the undirected graph, with backedges shown as dashed lines.

In the actual implementation, node expansion and undirection of edges are not carried out literally; instead, a tuple holding information for the (conceptually) expanded nodes is associated with each node in the directed graph.

Theorem 2 Let $S$ be a strongly connected component, and let $U$ be the undirected graph formed from $S$ by the above transformation. Nodes $a$ and $b$ are cycle equivalent in $S$ if and only if their corresponding nodes $a'$ and $b'$ are cycle equivalent in $U$.

Proof:

[$\Rightarrow$] Suppose $a$ and $b$ are not cycle equivalent in $S$. Then there is some directed cycle in $S$ containing one of $a$ or $b$ but not both. The corresponding undirected cycle in $U$ contains one of $a'$ and $b'$ but not both, so $a'$ and $b'$ are not cycle equivalent in $U$.

[$\Leftarrow$] Without loss of generality, assume there is at least one cycle in $G$ containing $a'$ but not $b'$. Such a cycle also contains $a_i$ and $a_o$, and they are adjacent to $a'$. Each edge on such a cycle has an associated direction from the intermediate transformation step. Adjacent edges either have the same direction or opposing directions; if adjacent edges have opposing directions, we say there is a direction change at the node between the opposing edges.

Let $C'$ be a cycle of minimum direction changes containing $a'$ but not $b'$. If $C'$ has no direction changes, then there is a corresponding directed cycle in $S$ that contains $a$ but not $b$.

Otherwise, $C'$ has one or more direction changes. Since $C'$ begins with edge $a' \rightarrow a_o$ and ends with edge $a_i \rightarrow a'$, there is no direction change at $a'$. Furthermore, there must be an even number of direction changes that alternate between inward opposing and outward opposing edges; the first direction change occurs between inward opposing edges.

Now consider the first two direction changes on $C'$. The first clash occurs at an input node $c_i$ since the directions are inward opposing. The second clash occurs at an output node $d_o$ since the directions are outward opposing. Nodes $c'$ and $d'$ are adjacent to $c_i$ and $d_o$ and are not on $C'$. Since $S$ is strongly connected, there exists a directed path in $S$ from $c$ to $d$; let $E'$ be the corresponding path from $c_i$ to $d_o$ in $U$.

If neither $a'$ nor $b'$ occur on $E'$, consider the cycle obtained by replacing the portion of cycle $C'$ between $c_i$ and $d_o$ with path $E'$. This is a cycle containing $a'$ but not $b'$, and it has two less direction changes than $C'$, which contradicts the assumption that cycle $C'$ had the minimum number of direction changes.
(a) Expanding Nodes and Undirecting Edges

(b) Transformation and DFS of Program in Figure 1

Figure 2: The Transformation and an Example
Otherwise, \( a' \) and \( b' \) may occur (perhaps several times) on \( E' \). If the first occurrence on \( E' \) (after \( c_1 \)) is \( a' \), then juxtapose the path \( a' \to c_1 \) in \( C' \) with the path \( c_1 \to a' \) to the first occurrence of \( a' \) in \( E' \). This is a cycle containing \( a' \) but not \( b' \) and it has no direction changes. Similarly, if the last occurrence of either \( a' \) or \( b' \) on \( E' \) is an \( a' \), the cycle \( a' \to d_o \to a' \) gives a cycle containing \( a' \), but not \( b' \), with fewer direction changes than \( C' \) which is a contradiction.

Otherwise, both the first and the last occurrence of \( a' \) or \( b' \) on \( E' \) are of \( b' \). The path \( b' \to d_o \to c_1 \to b' \) gives a directed cycle containing \( b' \) but not \( a' \).

Therefore, we conclude that two nodes are cycle equivalent in \( S \) iff their representatives are cycle equivalent in \( U \). \( \square \)

In Figure 2(b), notice that nodes \( c' \) and \( f' \) occur in all the same cycles. Every other representative node has some cycle that separates it from other representatives. For example, node \( b' \) occurs in a cycle \( a_o \to b_i \to b' \to b_o \to c_i \to a_o \) which does not contain any other representative. Similarly, \( e' \) is separated from \( a' \) by the cycle containing \( c', d', f' \) and \( b' \).

4 Computing cycle equivalence in undirected graphs

To determine the set of cycle equivalent nodes in an undirected graph, we could enumerate all cycles in the graph but this would be too expensive. Fortunately, we can use depth-first search to recast the problem in terms of sets of backedges rather than sets of cycles.

4.1 Brackets of a node

Let \( U \) be an undirected graph formed from a control flow graph by

1. adding an edge \( \text{END} \to \text{START} \), and then
2. expanding each node and undirecting each edge as shown in Figure 2(a).

We perform a depth-first search of \( U \) starting from \( \text{START} \) and obeying the following rule:

Whenever DFS visits a node \( a_o \) or \( a_o \), prefer to visit \( a' \) before visiting any other neighbor.

This rule ensures that nodes \( a' \) corresponding to nodes in the original control flow graph are interior nodes in the DFS spanning tree. Thus, these interesting nodes are adjacent only to a tree parent and a tree child (nodes \( a_i, a_o \)) and have no incident backedges. After depth-first search, every edge is either a tree edge or a backedge, so every cycle in \( U \) consists of tree and backedges.

**Definition 5** In any depth-first traversal of \( U \), a bracket of a node \( n \) is a backedge connecting \( n \) or one of its descendants to an ancestor of \( n \) or to \( n \) itself.

Given our convention for depth-first search, a bracket of a representative node \( a' \) always connects a strict descendant of \( a' \) to a strict ancestor of \( a' \). We now show that the set of brackets of a representative node effectively identify the set of cycles through that node.

**Lemma 2** In any DFS walk of \( U \), if representative nodes \( a' \) and \( b' \) have any bracket in common then they are ordered by the ancestor relation in the DFS tree.

**Proof:** (by contradiction) Suppose \( a' \) and \( b' \) are not ordered by the ancestor relation. Then no descendant of \( a' \) is a descendant of \( b' \) and vice versa. Any bracket of \( a' \) connects a descendant of \( a' \) (say node \( x \)) to an ancestor of \( a' \) (say node \( y \)); since \( x \) is not a descendant of \( b' \), this cannot be a bracket of \( b' \). \( \square \)
Theorem 3 Representative nodes $a'$ and $b'$ in $U$ are cycle equivalent if and only if they have the same set of brackets in every DFS walk of $U$.

Proof: (a) We show that if two nodes do not have the same set of brackets in a DFS walk of $U$, they are not cycle equivalent. Suppose edge $e$ is a bracket of node $a'$ but not $b'$. Let $x$ and $y$ be the two end points of edge $e$. Consider the tree path between $x$ and $y$ by definition of brackets, $a'$ must occur on this path but $b'$ cannot. By concatenating edge $e$ with the tree path from $x$ to $y$, we get a cycle containing $a'$ but not $b'$.

(b) If $a'$ and $b'$ have the same set of brackets, Lemma 2 asserts that they are ordered by the ancestor relation in the DFS tree. Without loss of generality, assume that $a'$ is an ancestor of $b'$. Consider any cycle through $a'$. There are no cross edges or forward edges in the DFS tree; moreover, $a'$ has no brackets incident on it. Therefore, the cycle must contain at least one edge that connects a strict descendant of $a'$ to a strict ancestor of $a'$. Let $e = (x, y)$ be the first such edge in the cycle after $a'$, so that all the nodes between $a'$ and $x$ in the cycle are descendants of $a'$. Since $e$ is a bracket of $a'$, it must be a bracket of $b'$, so $x$ must be a strict descendant of $b'$. If the path from $a'$ to $x$ does not pass through $b'$, there must be some edge $(p, q)$ on the path where $p$ is a descendant of $a'$ but not of $b'$ and $q$ is a strict descendant of both $a'$ and $b'$. Since there are no cross edges in the tree, $(p, q)$ is a bracket of $b'$ but not of $a'$, which contradicts the assumption that $a'$ and $b'$ have the same set of brackets. Therefore, every cycle containing $a'$ must contain $b'$. The proof that every cycle containing $b'$ contains $a'$ is similar and is omitted. □

In Figure 2(b), $c'$ and $f'$ have the brackets $(b_c, a_c)$ and $(b_s, c_s)$. No other representative node has exactly this set of brackets; for example, note that node $d'$ has the bracket $(c_t, c_o)$.

We can use a straight-forward DFS algorithm to compute the set of brackets for each node; by comparing these sets, we can compute cycle equivalence. However, comparisons of sets are expensive, so we proceed as follows. Intuitively, the set of brackets of a node can be viewed as a name for the cycle equivalence class of the node. To speed the comparisons of names, we need a compact naming scheme for the equivalence classes. We describe one such scheme next.

4.2 Compact names for sets of brackets

Consider the graph shown in Figure 3(a) in which the DFS tree is simply a chain, and backedges correspond to 'structured' loops in which loops are either disjoint or are nested within each other. For simplicity,
only representative nodes are shown in the figure. For such graphs, it is easy to see that the set of brackets of a representative node is uniquely named by the innermost bracket of that node. To compute cycle equivalence for such graphs, we simply visit nodes in reverse depth-first order, keeping a stack of brackets - when we exit a node, we delete all brackets connecting the node to a descendent, and then push all brackets connecting the node to an ancestor. Each representative node is labeled with the name of the topmost bracket on the stack when the DFS visits it, so nodes with the same label have the same control dependences.

The next complication to consider is that backedges may partially overlap with each other. This situation is shown in Figure 3(b). The first problem is that in the reverse DFS walk, brackets are not deleted in stack order. Moreover, note that nodes $a'$ and $b'$ do not have the same set of brackets even though the topmost element of the bracket stack of both nodes is $z$. To fix the deletion problem, we represent the bracket set as a doubly-linked list rather than as a stack. Brackets will always be added to the same end of the list, but may be deleted from anywhere within the list. In this way, the most recently added bracket (the bracket whose lower endpoint is highest in the tree) will be at the head of the list. If the brackets are properly nested, the list will behave exactly like a stack. In addition, we maintain a counter that keeps track of the size of the bracket set. It is easy to see that the pair $<\text{topmost bracket, set size}>$ uniquely labels each equivalence class — for example, in Figure 3(b), nodes $a'$ and $b'$ will be placed in different equivalence classes, while nodes $a'$ and $c'$ are placed in the same equivalence class.

Finally, we must handle the case of general DFS trees, not just chains. Figure 3(c) gives an example. When we encounter a node that has more than one child, the bracket sets of the children must be merged. Unfortunately, the ‘most recently added’ bracket is no longer well defined. For example, at node $e$ in Figure 3(c), it is not clear whether the most recently added backedge should be edge $(f, d)$ or edge $(h, c)$. The resolution of this difficulty rests on the observation that only one of the sub-trees below node $e$ can contain any nodes cycle-equivalent to any ancestor of $e$. The reason is that nodes in the sub-trees can only have brackets that originate in the same sub-tree. Therefore any ancestor of $e$ having brackets from more than one of the branches cannot be cycle equivalent to a node in any of the branches. Therefore, nodes between $e$ and $b$ cannot be cycle equivalent to a node below $e$. However, nodes between $b$ and $a$ can be cycle equivalent to nodes between $h$ and $i$.

The solution therefore is to add an additional backedge whenever we need to merge two bracket sets. This backedge becomes the topmost bracket in the set, and the children’s bracket sets are then concatenated in arbitrary order. The new bracket originates from the vertex whose children are being merged, and extends up to the highest vertex whose brackets come from more than one of the branches. To add it thus requires keeping track, at each point in the tree, of the highest vertex reached by any backedge below this point. The destination of the new backedge is the second-highest of the children’s high backedges. This could be found by examination of the bracket sets, but the highest-ending backedge is not necessarily related to the first bracket in each set (the highest-originating) so a full search of the bracket set would be necessary. Fortunately, we can simply compute it independently in constant time. In Figure 3(c), we would add a dummy backedge from $e$ to $b$, as shown. Once this is done, the pair $<\text{topmost bracket, set size}>$ identifies the equivalence class as before.

### 4.3 The Algorithm

We can put these observations together into the following fast algorithm which makes use of an abstract data type called BracketList to maintain lists of brackets. The following operations are defined on this data type.

- **create ( )** : BracketList : Make an empty BracketList structure;
- **size ( bl:BracketList )** : integer : Number of elements in BracketList structure;
- **push ( bl:BracketList, e:bracket )** : BracketList : Push e on top of bl;
- **top ( bl:BracketList )** : bracket : Topmost bracket in bl;
- **delete ( bl:BracketList, e:bracket )** : BracketList : Delete e from bl;
- **concat ( bl1,bl2:BracketList )** : BracketList : Concatenate bl1 and bl2;

This abstract data type can be implemented as a record consisting of
1. a doubly-linked list of brackets,
2. a pointer to the last cell of the list, and
3. an integer representing the size of the list.

The doubly-linked list permits deletions anywhere in the list. The pointer to the last cell of the list permits fast concatenation of lists by in-place update to the cell. We leave it to the reader to verify that each of the operations of the abstract data type can be implemented in constant time using this concrete representation. The only subtlety is in delete. When an edge is pushed onto a bracket list, the edge data structure is updated so it has a pointer to the bracket list cell containing that edge; this permits constant time deletion of an edge from a bracket list.

We use integers to identify $CD_E$ classes. We assume a procedure \texttt{new-Clas-name()} which returns a new integer each time it is called. This can be implemented using a procedure with a \textit{static} variable that starts at 0 and that is incremented each time the procedure is called; the procedure simply returns the current value of the variable.

The following operations on nodes are used in the algorithm.

\begin{itemize}
\item \textbf{Number ( n:node ) : integer :} DFS number of node
\item \textbf{Class ( n:node ) : integer :} $CD_E$ class of node;
\item \textbf{BList ( n:node ) : BracketList :} List of brackets of node;
\item \textbf{Hi ( n:node ) : integer :} Highest destination node of any edge originating from a descendent of node;
\end{itemize}

Since an edge may be the topmost bracket of the bracket list of several representative nodes, the edge data structure caches the $CD_E$ class number and the size of the bracket list when the edge was most recently the topmost bracket of a node's bracket list. For example, in Figure 3(b), edge \( z \) is the topmost bracket for nodes \( c', a' \) and finally \( b' \). \( a' \) is given the same $CD_E$ class number as \( c' \) because the size of the bracket lists at \( a' \) is the same as the value cached on edge \( z \) at node \( c' \). In contrast, \( a' \) and \( b' \) are given different $CD_E$ class numbers. To access the values cached on brackets, the following operations are defined on edges.

\begin{itemize}
\item \textbf{RecentSize ( e:edge ) : integer :} Size of bracket set when \( e \) was most recently the topmost bracket for a representative node;
\item \textbf{RecentClass ( e:edge ) : integer :} $CD_E$ class number of representative node for which \( e \) was most recently the topmost bracket;
\end{itemize}

The edge and node data types can be implemented using records in the obvious way.

It is easy to see that during the DFS walk of the undirected graph, the amount of work performed at each node is some constant amount together with work proportional to the number of edges incident at the node. Since the numbers of nodes and edges in the undirected graph is the same, within a constant factor, as in the control flow graph, the complexity of the algorithm is $O(E)$ where $E$ is the number of edges in the original control flow graph.

### 4.4 Correctness

In proving the correctness of the algorithm, we will need the following lemma.

**Lemma 3** The backedges added by the algorithm do not affect the cycle-equivalence relation for representative vertices.

**Proof:** By Theorem 3, two representative vertices are cycle-equivalent if and only if they have the same set of brackets. Consider two representative vertices $x'$ and $y'$. If they have the same set of brackets with the added backedges, then clearly deleting the new backedges will still leave them with the same sets. If they have the same brackets from the original edge set, we must show that they will share the same set of new brackets.

Consider the example in Figure 3(c). Suppose node $x'$ is bracketed by the new backedge $(e, b)$. The origin of that backedge, $e$, is a node with at least two children: the highest-reaching branch...
Procedure cycle_equiv (G : CFG)
{
  /* Preprocessing */
  Add edge from END to START in G;
  Expand each node and undirect edges as described in Section 3;
  Perform depth-first search of undirected graph as described in
  Section 4.1, assigning preorder number to each node;

  /* Compute CD equivalence classes */
  for each node n, in reverse depth-first order, do
  {
    /* Compute Hi(n) */
    hi0 := min{ Number(t) | (t,n) is a backedge }; /* hi using backedges only */
    hi1 := min{ Hi(c) | c is a child of n }; /* how hi through children */
    hi-child := any child of n whose Hi value is equal to hi1; /* hi1 going child */
    hi2 := min{ Hi(c) | c is a child of n other than hi-child };
    Hi(n) := min{ hi0, hi1, hi2 };

    /* Compute BList(n) */
    BList (n) := create ();
    for each child c of n do
      BList (n) := concat (BList (n), BList (c)) od;
    for each backedge e from a descendant of n to n do
      BList (n) := delete (BList (n), e) od;
    for each backedge e from n to an ancestor of n do /* in any order! */
      { /* capping' backedge */
      BList (n) := push (BList (n), e) od;
      RecentSize (e) := -1; /* n is not a representative node */
      }
    if n has more than one child then
      push (BList (n), <n,hi2>); /* 'capping' backedge */
  } /* for each node */
} /* procedure */

Figure 4: The \textit{CYC}_{=} Algorithm
has a backedge to a point higher in the tree than $b$ (or to $b$ itself, if there was a tie), and the second-highest-reaching branch has a backedge to $b$. Now consider where the node $y'$ (which shares with $x'$ all brackets from the original set) can occur. It must share the bracket $(g, b)$ with $x'$, so it must be somewhere on the tree path from $g$ to $b$, but it also shares the bracket $(i, a)$, so must occur on the path from $i$ to $a$. The only region where these two paths intersect is in the path from $e$ to $b$. This means that $y'$ will also have the new bracket $(e, b)$.

**Theorem 4** The fast algorithm is correct.

**Proof:** We need to prove that two representative vertices will be in the same class if and only if they are cycle-equivalent. One direction is reasonably easy: if two vertices are cycle-equivalent, they will end up in the same class. By Theorem 3 two cycle-equivalent vertices will have the same bracket sets. By Lemma 3 the backedges added during the DFS will not affect the cycle-equivalence relation. Therefore the bracket sets, as computed by the algorithm, will have the same size and the same top bracket. They will therefore be grouped into the same class.

To complete the proof, we need to establish that if two representative vertices are not cycle-equivalent, then they will not be in the same class. Let $a'$ and $b'$ be two representative vertices that are not cycle-equivalent. By Theorem 3 they must have different bracket sets, including (by Lemma 3) the new backedges added by the algorithm. If these sets are different size, the algorithm clearly places them in different class, so let us suppose the bracket sets are the same size. By Lemma 2, if $a'$ and $b'$ are not ordered by the ancestor relation, then they have no brackets in common and are therefore placed in different classes. Otherwise, assume without loss of generality that $a'$ is an ancestor of $b'$. Since the sets are the same size, but not identical, $a'$ must have a bracket $(p, q)$ not shared by $b'$, and $b'$ must have a bracket $(r, s)$ not shared by $a'$. The node $p$ is a descendant of $a'$ — if it is also an ancestor of $b'$ then the edge $(p, q)$ must be linked on the bracket list ahead of $b'$'s top bracket. Either $(p, q)$ will be the top bracket, or there will be another still higher. This bracket cannot include $b'$, so $b'$ will have a different top bracket and will be in a different class.

Now assume that $p$ is *not* an ancestor of $b'$. In that case the paths from $a'$ to $p$ and to $b'$ diverge at some point. Call this node $d$. Since $d$ has multiple children, a backedge was added from $d$ to a point at which backedges from only one of the branches were still present. If that point is above $a'$, then the added backedge will bracket $a'$, and it or a still higher backedge will be the top bracket for $a'$, while it could not be the top bracket for $b'$. If the point is below $a'$, then all backedges from the branch $b'$ is on must have ended below $a'$, so the top bracket for $b'$, whatever it is, must have ended also and so cannot be a bracket of $a'$. In either case, $a'$ and $b'$ will have different top brackets, and so will be in different classes.

**4.5 Implementation**

Our algorithm for $CD_d$ is asymptotically optimal; in addition, the constant factor is small and the algorithm runs fast in practice. One detail of our implementation is worth noting: we avoid explicitly expanding nodes and undirecting edges. Instead, we use doubly-linked control flow edges (so that depth-first search can traverse edges in either direction), and we maintain a tuple of information at each control flow node, corresponding to the information that would be stored on the expanded nodes. The resulting code is slightly more complex, but the savings in space and time over working with the explicitly transformed graph are significant.

Figure 5 shows two graphs for which the control dependence relation is quadratic in the size of the program. We compared our implementation of the $CD_d$ algorithm to an implementation of Lengauer and Tarjan's fast postdominator algorithm, running on a Sun SPARC ELC. Figure 6 shows that the running time of our algorithm is linear in the size of the program, as expected. This figure also shows that our algorithm runs faster than the postdominator algorithm on all graphs. Times shown are the average over 25 consecutive runs.
We chose these examples because any straightforward algorithm for \( CD_{=\equiv} \) that partitions nodes into equivalence classes using the control dependence relation must have a running time that is quadratic in the size of the program. Moreover, finding postdominators is the first step in the standard algorithms for the computation of control dependence [FOW87] or for computing control dependence equivalence [CFS90] — therefore, we compute control dependence equivalence faster than the first step of either traditional algorithm.

5 Finding Single Entry Single Exit regions

An important application of cycle equivalence is the determination of single entry single exit (SESE) regions of a control flow graph. SESE regions are defined as follows.

**Definition 6** Two distinct nodes \( a \) and \( b \) in a control flow graph \( G \) enclose a single entry single exit region if

- \( a \) dominates \( b \)
- \( b \) postdominates \( a \)
- \( a \) and \( b \) are cycle equivalent in \( G \).

Every node in a control flow graph constitutes by itself what we might intuitively call a SESE region, but we exclude such trivial regions from consideration by requiring that \( a \) and \( b \) be distinct nodes. The first two conditions ensure that control reaches \( b \) whenever it reaches \( a \) and vice versa. The third condition ensures that whenever control reaches \( a \), it reaches \( b \) before reaching \( a \) again and vice versa. These conditions correspond to our intuitive notion of a single entry single exit region. We use an ordered pair \((a, b)\) to denote a SESE region where \( a \) is the entry node and \( b \) is the exit node. In practice, we have found it more useful to work with SESE regions defined by pairs of edges, rather than by pairs of nodes, satisfying these properties. It is straightforward to rephrase our discussion in terms of edges, but we will not do so here.

5.1 Finding Canonical SESE regions

The first problem we must solve is the determination of non-trivial SESE regions of a graph. If \((a, b)\) is a SESE region and \((b, c)\) is a SESE region, then \((a, c)\) is a SESE region as well. In other words, SESE regions have a certain transitivity property. A graph with \( N \) nodes can have \( O(N^2) \) SESE regions — each of the
Figure 6: Linear time behavior on difficult graphs
$N^2$ node pairs in a chain of $N$ nodes encloses a SESE region. However, we have never found any use for complete enumeration of all SESE regions of a graph. Instead, for each node $x$ in the graph, we want to find the smallest SESE regions, if they exist, for which $x$ is an entry node or an exit node. We will call these the canonical SESE regions associated with $x$; taking the ‘transitive closure’ of the canonical SESE regions of a graph generates all the SESE regions of that graph. We express this more formally as follows.

1. Given a node $x$, find a node $b$, if it exists, such that
   - $(x, b)$ is a SESE region, and
   - if $(x, b')$ is also a SESE region, then $b$ dominates $b'$.

2. Given a node $x$, find a node $a$, if it exists, such that
   - $(a, x)$ is a SESE region, and
   - if $(a', x)$ is also a SESE region, then $a$ postdominates $a'$.

The following result is the key to solving this problem for all nodes in the graph in $O(E)$ time.

**Theorem 5** Let $S$ be the strongly connected component constructed by adding an edge END $\rightarrow$ START to a control flow graph $G$. Nodes $a$ and $b$ in $G$ enclose a SESE region iff they are cycle equivalent in $S$.

**Proof:** Suppose $a$ and $b$ enclose a SESE region in $G$. Consider any cycle in $S$. If this cycle does not include the edge END $\rightarrow$ START, the cycle is present in $G$ and must contain either both or neither of $a$ and $b$. Suppose the cycle does contain the edge END $\rightarrow$ START. If the cycle contains $a$ but not $b$, then $a$ must occur on the portion of the cycle from START to END. Since this path contains $a$ but not $b$, $b$ does not postdominate $a$, which contradicts the definition of SESE regions. Therefore, if the cycle contains $a$ it must contain $b$. A similar argument shows the converse. Therefore, every cycle in $S$ contains both or neither of $a$ and $b$, which means $a$ and $b$ are cycle equivalent.

Suppose $a$ and $b$ are cycle equivalent in $S$. Every cycle in $G$ occurs in $S$; therefore, $a$ and $b$ must occur in all the same cycles in $G$. It remains to be shown that $a$ and $b$ are ordered by both dominance and postdominance. Every path from START to END contains both $a$ and $b$ or neither. Without loss of generality, suppose that there is a $b$-free path from START to $a$. Then, every path from $a$ to END must contain $b$, so $b$ postdominates $a$. Since $a$ and $b$ are distinct nodes $a$ cannot postdominate $b$, so there is an $a$-free path from $b$ to END. This means that $a$ must occur on every path from START to $b$; in other words, $a$ dominates $b$. Therefore, $a$ and $b$ are ordered by dominance and postdominance. $\Box$

We can compute cycle equivalent nodes in $O(E)$ time and space using the algorithm given earlier. To find canonical SESE regions, we simply order the nodes in each cycle equivalence class by dominance (alternatively, by postdominance). This is easy to do in a single DFS walk of the control flow graph because if $a$ dominates $b$, then $a$ must be an ancestor of $b$ in every DFS walk of the graph. Therefore, once each node has been assigned to a cycle equivalence class by the algorithm of Figure 4, we can order the nodes in each class by inserting them into each class in the order visited during a DFS walk of the graph.

To find the canonical SESE regions associated with a node $N$, we simply find, if they exist, the predecessor and the successor of $N$ in the ordered cycle equivalence class of $N$. This takes $O(E)$ time for all nodes in the graph.

### 5.2 Nesting structure of canonical SESE regions

The second problem which arises in many applications is the determination of the nesting structure of canonical SESE regions.

**Definition 7** A node $n$ in a graph $G$ is contained within a SESE region $(a, b)$ if $a$ dominates $n$ and $b$ postdominates $n$.
Intuitively, node \( n \) is 'between' \( a \) and \( b \) in the graph. This definition can be extended in the obvious way to containment of SESE regions. Theorem 6 describes how canonical SESE regions in a graph are related.

**Theorem 6** If \( R_1 \) and \( R_2 \) are two canonical SESE regions of a graph, one of the following statements applies.

1. \( R_1 \) and \( R_2 \) are node disjoint.
2. \( R_1 \) is contained within \( R_2 \) or vice versa.
3. The exit node of \( R_1 \) is the entry node of \( R_2 \) or vice versa.

In other words, canonical SESE regions cannot have any partial overlap — if two regions have any nodes in common, they are either nested or in tandem\(^3\). This is most obvious in the case of structured programs. For general control flow graphs, the required result may be proved as follows.

**Proof:**

Suppose distinct canonical SESE regions \((a, b)\) and \((c, d)\) both contain a node \( n \). Then \( a \) and \( c \) both dominate \( n \), and so they are ordered by dominance. Without loss of generality, assume that \( a \) dominates \( c \). Similarly, \( b \) and \( d \) both postdominate \( n \), and so they are ordered by postdominance.

If \( b \) postdominates \( d \), then \((c, d)\) is contained within \((a, b)\).

Otherwise, \( d \) postdominates \( b \). There are now three cases to consider.

1. Suppose \( b \) and \( c \) are the same node. Since \( b \) postdominates \( n \) and \( c \) dominates \( n \), it follows that \( n \), \( b \), and \( c \) must be the same node. Therefore, \( n \) is the only node in common and the two SESE regions are in tandem.

2. Suppose \( c \) dominates \( b \) and \( c \) and \( b \) are distinct nodes. We show that \( a \), \( b \), \( c \) and \( d \) are in the same control region and that \((a, b)\) cannot form a canonical SESE region, leading to a contradiction.

Since \( a \) dominates \( c \) and \( c \) dominates \( b \), it follows that every path from \( a \) to \( b \) must contain \( c \). Since \( b \) postdominates \( a \), this means that \( c \) postdominates \( a \) and \( b \) postdominates \( c \). Therefore, \( a \), \( b \), and \( c \) are ordered by dominance/postdominance; this argument can be extended to show that this is true for \( d \) as well, but this is not necessary for the proof.

Every cycle through \( a \) and \( b \) contains a path from \( a \) to \( b \), and hence contains \( c \). If there is a cycle through \( c \) that does not contain \( b \), then there is a path from \( c \) to \( d \) without \( b \); since \( d \) postdominates \( c \), this means that \( b \) does not postdominate \( c \), which is incorrect. Therefore, every cycle through \( c \) contains \( b \) and hence contains \( a \). This means that \( a \), \( c \), and \( b \) are in the same cycle equivalence class and \( c \) should be ordered between \( a \) and \( b \) in the equivalence class. This violates the assumption that the exit node of the canonical SESE region with entry \( a \) is \( b \).

Therefore, this case cannot occur.

3. Suppose \( c \) does not dominate \( b \). Then, there is a \( c \)-free path from \textsc{start} to \( b \). This means that \( c \) must postdominate \( b \); otherwise, we have a path \( \text{start} \xrightarrow{c} b \xrightarrow{c} \text{end} \) which passes through \( d \) (since \( d \) postdominates \( b \)) but not \( c \), violating the assumption that \( c \) dominates \( d \). Since \( b \) postdominates \( n \), this means that \( c \) postdominates \( n \). But \( c \) also dominates \( n \). Therefore, \( c \) and \( n \) must be identical. Therefore, \( b \) postdominates \( c \). Since we have shown that \( c \) postdominates \( b \), \( b \) and \( c \) are identical. But by assumption, \( c \) does not dominate \( b \), so \( c \) and \( b \) are distinct. We conclude that this case cannot occur.

\( \square \)

\(^3\)Notice that this property may not be true for SESE regions that are not canonical.
To determine the smallest SESE region containing a given SESE region, we can extend the DFS algorithm that determines canonical SESE regions. We maintain a stack during DFS that records the SESE regions containing the node being visited. Whenever DFS enters a SESE region through either the entry node or retreats into it through the exit node, we push the name of the SESE region on this stack; when exiting the region, we pop this name from the stack. From Theorem 6, it follows that the pushing and popping follows a stack discipline, which is the required behavior. The topmost SESE region on this stack when DFS reaches the entry node of a SESE region \( R_1 \) is the name of the smallest SESE region containing \( R_1 \).

### 5.3 Discussion

As a final remark, we note that a concept related to SESE regions called a **hammock** has been defined in the literature [Kas75]. Although it appears common practice to confuse hammocks and SESE regions, they are in fact different. The exit node of a hammock can be the target of edges outside the hammock; therefore, there are hammocks that are not SESE regions, as shown in Figure 7. Also, the entry node of a hammock cannot be the target of an edge from the exit of the hammock; therefore, there are SESE regions that are hammocks. For example, a repeat-until loop in a structured program is a SESE region, but it is not a hammock.

The standard algorithm for finding hammocks has complexity \( O(EN) \) [Kas75]. It may be possible to use the ideas of this paper to find hammocks in \( O(E) \) time, but we have not explored this further.

### 6 Extensions

We conclude this paper by describing three open problems related to this work.

A small program transformation can increase the number of SESE regions in the program. When a set of edges come together at some node in the control flow graph, it may be possible to introduce dummy nodes in the graph to bring the edges together in stages, rather than all at once. This process, called **merge factoring**, is shown in Figure 7. Nodes \( a \) and \( b \) in the original program do not form a SESE region. When the dummy node \( d \) is inserted, nodes \( a \) and \( d \) form a SESE region. Is there a fast algorithm for deciding where and how merge factoring should be performed? Note that the original program in Figure 7 is a hammock; therefore this problem may be related to hammock detection.

Secondly, it may be possible to use the techniques in this paper to represent the full control dependence relation in \( O(E) \) space such that the control dependences of a node can be determined in time proportional to the number of these dependences. This problem is of academic interest since we have never found a need to enumerate all the control dependences of a node. The intuitive idea is the following. The full control dependence relation has \( O(EN) \) size. However, it may be possible to **factor** this relation to represent it in \( O(E) \) space and to compute it in \( O(E) \) time. Notice that control dependence equivalence can be used to reduce the space required to represent the control dependence relation — for example, we can introduce a **region node** to represent an equivalence class, connect each node in the class to the region node, and direct
control dependences to the region node, not to every node in the equivalence class. If there are $P$ nodes in the equivalence class and they have $C$ control dependence edges each, the space required to represent this information goes down from $P \times C$ to $P + C$, without increasing the (asymptotic) time required to find all nodes with a given control dependence, or the time required to find all the control dependences of a node. We do not know if this idea can be extended to compute and to represent a factored control dependence relation in linear time and space$^4$.

Finally, is there a fast incremental algorithm for updating the control dependence equivalence relation when the control flow graph is changed? Such an algorithm would be useful in programming environments and text editors in which the structure of control flow may be modified incrementally by editing the text of the program. The theory of incremental algorithms for determining properties of directed graphs is still in its infancy. However, there is a well-established body of work on such incremental algorithms for undirected graphs [Rau92]. Since our algorithm for $CD\equiv$ ultimately solves a problem on undirected graphs, this technology may be applicable to the $CD\equiv$ problem for directed graphs.

References


$^4$In [JP93], we implied that we had a solution, but in writing this paper we realized that the problem remains open.