RC 21245(94831)29JUL98 Computer Science/Mathematics

Research Report

On Sparse Evaluation Representations

G. Ramalingam
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
rama@watson.ibm.com

IBM Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Abstract

The Sparse Evaluation Graph has emerged over the past several years as an intermediate representation that captures the dataflow information in a program compactly and helps perform dataflow analysis efficiently. The contributions of this paper[1] are three-fold:

- We present a linear time algorithm for constructing a variant of the Sparse Evaluation Graph for any dataflow analysis problem. Our algorithm has two advantages over previous algorithms for constructing Sparse Evaluation Graphs. First, it is simpler to understand and implement. Second, our algorithm generates a more compact representation than the one generated by previous algorithms. (Our algorithm is also as efficient as the most efficient known algorithm for the problem.)
- We present a formal definition of an *equivalent flow graph*, which attempts to capture the goals of sparse evaluation. We present a quadratic algorithm for constructing an equivalent flow graph consisting of the minimum number of vertices possible. We show that the problem of constructing an equivalent flow graph consisting of the minimum number of vertices and edges is NP-hard.
- We generalize the notion of an equivalent flow graph to that of a *partially equivalent flow graph*, an even more compact representation, utilizing the fact that the dataflow solution is not required at every node of the control-flow graph. We also present an efficient linear time algorithm for constructing a partially equivalent flow graph.

Keywords: sparse evaluation graphs, static single assignment forms, dataflow analysis, graph transformations, quick propagation graphs, equivalent flow graphs, partially equivalent flow-graphs.

1 Introduction

The technique of sparse evaluation has emerged, over the past several years, as an efficient way of performing program analysis. Sparse evaluation is based on the simple observation that for any given analysis problem, a number of the "statements" in a given program may be "irrelevant" with respect to the analysis problem. As a simple example, consider any version of the "pointer analysis" problem (*e.g.*, see [13, 1]), where the goal is to identify relations that may exist between pointer-valued variables. An assignment to an integer-valued variable, such as "`i := 10;`", will typically be irrelevant to the problem and may be ignored. The goal of sparse evaluation is very simply to construct a "smaller" program whose analysis is *sufficient* to produce results for the original program. This not only enables the analysis to run faster, but also reduces the space required to perform the analysis. (For some complex analyses like pointer analysis, for which space is often a bottleneck, sparse evaluation can make the difference between being able to complete the analysis and not.)

The idea of sparse evaluation was born in the context of the work done on the *Static Single Assignment* (SSA) form [5, 6], which showed how the SSA form could help solve various analysis problems, such as constant propagation and redundancy elimination, more efficiently. Choi *et al.* [2] generalized the idea and showed how it could be used for an arbitrary dataflow analysis problem expressed in Kildall's framework [12]. The idea is, in fact, applicable to analysis problems expressed in various different frameworks, and more generally, to the problem of computing extremal fixed points of a collection of equations of certain form. However, for the sake of concreteness, we will also deal with analysis problems expressed in Kildall's framework.

In the context of Kildall's framework, we are interested in solving some dataflow analysis problem over a control-flow graph G. The idea behind sparse evaluation is to construct a smaller graph H, which we will refer to as an *equivalent flow graph*, from whose dataflow solution the solution to the original graph G can be trivially recovered. More detailed discussions of equivalent flow graphs and their use can be found in [2, 11, 16].

Choi *et al.* [2] define a particular equivalent flow graph called the *Sparse Evaluation Graph* (SEG) and present an algorithm for constructing it. Johnson *et al.* [11, 10] define a different equivalent flow graph called

[1] A preliminary version of this paper appeared in the proceedings of the *Fourth International Static Analysis Symposium*, volume 1302 of *Lecture Notes inComputer Science*, pages 1–15.

the *Quick Propagation Graph* (QPG) and present a linear time algorithm for constructing it. In general, the Quick Propagation Graph is not as compact as the Sparse Evaluation Graph. Cytron and Ferrante [4], Sreedhar and Gao [16], and Pingali and Bilardi [14, 15] improve upon the efficiency of the original Choi *et al.* algorithm for constructing the Sparse Evaluation Graph. Duesterwald *et al.* [8] show how a congruence partitioning technique can be used to construct an equivalent flow graph, which we believe is the same as the standard Sparse Evaluation Graph. The Pingali-Bilardi algorithm and the Sreedhar-Gao algorithm, which are both linear, have the best worst-case complexity among the various algorithms for constructing the Sparse Evaluation Graph.

The contributions of this paper are as follows.

- We define a new equivalent flow graph, the *Compact Evaluation Graph* (CEG), and present a linear time algorithm for constructing the Compact Evaluation Graph for any (monotonic) dataflow analysis problem. Our algorithm has two advantages over previous algorithms.
  - *Simplicity.* Our algorithm is particularly simple to understand and implement. It is conceptually simple because it is based on two graph transformations, whose correctness is transparently obvious. In implementation terms, its simplicity derives from the fact that it does not require the computation of the dominator tree. It utilizes just the well-known strongly connected components algorithm and the topological sort algorithm.
  - *Compactness.* In general, the Compact Evaluation Graph is smaller than both the Sparse Evaluation Graph and the Quick Propagation Graph. In particular, we show that both SEG and QPG can also, in principle, be generated utilizing the two graph transformations mentioned above. However, while the Compact Evaluation Graph is in normal form with respect to these transformations, SEG and QPG are not necessarily so. Since these transformations make the graph smaller and since these transformations are Church-Rosser, it follows that the Compact Evaluation Graph is at least as small as both SEG and QPG.
- For a reasonable definition of equivalent flow graph, we present a quadratic algorithm for constructing a equivalent flow graph consisting of the *minimum number of vertices.* We also show that the problem of constructing equivalent flow graph consisting of the *minimum number of vertices and edges* is NP-hard.
- We show how we can utilize the fact that the dataflow solution is not required at every node of the control-flow graph to construct an even more compact representation, which we call a *partially equivalent flow graph.* We present an efficient algorithm, also based on simple graph transformations, to produce a partially equivalent flow graph.

The rest of the paper is organized as follows. Section 2 presents the notation and terminology we use. Section 3 describes the Compact Evaluation Graph and an algorithm for constructing it. Section 4 discusses the graph transformations used to construct the Compact Evaluation Graph. Section 5 compares the Compact Evaluation Graph to previously proposed equivalent flow graphs. Section 6 presents our results concerning equivalent flow graphs of minimum size. Section 7 introduces the concept of a partially equivalent flow graph and presents an algorithm for constructing one. Section 8 briefly discusses how these concepts apply in the case of interprocedural analysis. Section 9 presents a comparison of our work with previous work, and Section 10 presents our conclusions.

2 Notation and Terminology

Dataflow analysis problems come in various different flavors, but many of the differences are cosmetic. In this paper we will focus on "forward" dataflow analysis problems, but our results are applicable to "backward" dataflow analysis problems as well. We will also assume that the "transfer functions" are associated with the vertices of the control-flow graph rather than the edges and that we are interested in identifying the dataflow solution that holds at exit from nodes. In particular, when we talk about the dataflow solution at a node u, we mean the solution that holds at exit from u.

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

Formally, a dataflow analysis problem instance is a tuple $\mathcal{F} = (\mathcal{L}, G, M, c)$, where:

- $\mathcal{L} = (L, \sqcap)$ is a semilattice
- G is a control-flow graph
- $M : V \rightarrow (L \rightarrow L)$ is a map from G's vertices to dataflow functions
- $c \in L$ is the "dataflow fact" associated with the entry vertex.

We will refer to $M(u)$ as the transfer function associated with vertex u. The function M can be extended to map every path in the graph to a function from L to L: if p is a path $[v_1, v_2, ..., v_k]$ then $M(p)$ is defined to be $M(v_k) \circ M(v_{k-1}) \cdots M(v_1)$. (It is convenient to generalize the above definition to any sequence of vertices, even if the sequence is not a path in the graph.) The meet-over-all-paths solution $MOP_{\mathcal{F}}$ to the problem instance $\mathcal{F} = (\mathcal{L}, G, M, c)$ is defined as follows:

$$MOP_{\mathcal{F}}(u) =_{def} \textstyle\bigsqcap_{p \in Paths(entry(G),u)} M(p)(c)$$

Here $Paths(entry(G), u)$ denotes the set of all paths p from $entry(G)$ to u. The maximal fixed point solution of the problem, denoted $MFP_{\mathcal{F}}$, is the maximal fixed point of the following collection of equations over the set of variables $\{x_u \mid u \in V\}$:

$$\begin{aligned} x_u &= M(u)(\bigsqcap_{v \rightarrow u} x_v) \quad \text{for every } u \in (V - \{entry(G)\}) \\ x_{entry(G)} &= M(entry(G))(c) \end{aligned}$$

Most of the results in this paper apply regardless of whether one is interested in the maximal fixed point solution or the meet-over-all-paths solution. Whenever we simply refer to the "dataflow solution", the statement applies to both solutions.

Assume that we are interested in solving some dataflow analysis problem over a control-flow graph G. The idea behind the sparse evaluation technique is to construct a (usually smaller) graph H, along with a function f from the set of vertices of G to the set of vertices of H such that the dataflow solution at a node u in graph G is the same as the dataflow solution at node $f(u)$ in graph H. This implies that it is sufficient to perform the dataflow analysis over graph H. Furthermore, the graph H and the mapping f are to be constructed knowing only the set of nodes I in G that have the identity transfer function with respect to the given dataflow analysis problem. (In other words, the reduction should be valid for any dataflow problem instance over the graph G that associates the identity transfer function with vertices in I.) Though this description is somewhat incomplete, it will suffice for now. We will later present a formal definition of an equivalent flow graph.

3 Compact Evaluation Graphs: An Overview

The goal of this section is to explain what the Compact Evaluation Graph is and our algorithm for constructing this graph in simple terms, without any distracting formalisms. Formal details will be presented in latter sections.

Let S be a set of nodes in G that includes the entry node of G as well as any node that has a non-identity transfer function with respect to the given dataflow analysis problem. We will refer to nodes in S (whose execution may *modify* the abstract program state) as m-nodes, and to other nodes (whose execution will *preserve* the abstract program state) as p-nodes.

We are given the graph G and the set S, and the idea is to construct a smaller graph that is equivalent to G, as explained earlier. Our approach is to generate an equivalent flow graph by applying a sequence of elementary transformations, very much like the T1-T2 style elimination dataflow analysis algorithms [18, 9]. We use two elementary transformations T2 and T4 (named so to relate them to the T1, T2, and T3 transformations of [18, 9]).

The Basic Transformations

Transformation T2: Transformation T2 is applicable to a node u iff (i) u is a p-node and, (ii) u has only one predecessor. Let v denote the unique predecessor of a node u to which T2 is applicable. The graph $T2(u,v)(G)$ is obtained from G by merging u with v: that is, we remove the node u and the edge $v \rightarrow u$ from the graph G, and replace every outgoing edge of u, say $u \rightarrow w$, by a corresponding edge $v \rightarrow w$. (Our T2 transformation is essentially the same as the one outlined by Ullman[18], but we apply it only to p-nodes.)

Note that the dataflow solution for graph G can be obtained trivially from the dataflow solution for graph $T2(u,v)(G)$. In particular, the dataflow solution for node u in G is given by the dataflow solution for node v in $T2(u,v)(G)$. The dataflow solution for every other node is the same in both graphs.

Transformation T4: The T4 transformation is applicable to any strongly connected set of p-nodes. (A set X of vertices is said to be strongly connected if there exists a path between any two vertices of X, where the path itself contains only vertices from X). If X is a strongly connected set of p-nodes in graph G, the graph $T4(X)(G)$ is obtained from G by collapsing X into a single p-node: in other words, we replace the set X of vertices by a new p-node, say w, and replace any edge of the form $u \rightarrow v$, where $u \notin X$ and $v \in X$ by the edge $u \rightarrow w$, and replace any edge of the form $u \rightarrow v$, where $u \in X$ and $v \notin X$ by the edge $w \rightarrow v$, and delete any edges of the form $u \rightarrow v$, where $u \in X$ and $v \in X$.

Note that the dataflow solution for graph G can be obtained trivially from the dataflow solution for graph $T4(X)(G)$. In particular, the solution for any node u in G is given by the solution for the (new) node w in $T4(X)(G)$ if $u \in X$, and by the solution for node u in $T4(X)(G)$ if $u \notin X$.

Our algorithm constructs an equivalent flow graph by taking the initial graph and repeatedly applying the T2 and T4 transformations to it until no more transformations are applicable. We will show latter that the final graph produced is independent of the order in which these transformations are applied. Let $(T2+T4)^*(G)$ denote the final graph produced. Every vertex u in the final graph $(T2+T4)^*(G)$ corresponds to a set S_u of vertices in the original graph G. Further, either S_u contains no m-nodes, in which case the transfer function associated with u in graph $(T2+T4)^*(G)$ is the identity function, or S_u contains exactly one m-node v (and zero or more p-nodes), in which case u's transfer function in graph $(T2+T4)^*(G)$ is the same as the transfer function of v in G. The dataflow solution to any vertex in S_u in G is given by the dataflow solution to the vertex u in the $(T2+T4)^*(G)$. We refer to $(T2+T4)^*(G)$ as the Compact Evaluation Graph of G.

Computing the Normal Form

We now present our algorithm for constructing the normal form of a graph with respect to the T2 and T4 transformations.

Step 1: Let G denote the initial control flow graph. Let G_p denote the restriction of G to the set of p-nodes in G. (That is, G_p is the graph obtained from G by removing all m-nodes and edges incident on them.) Identify the maximal strongly connected components of G_p using any of the standard algorithms. Let X_1, X_2, $\cdots$, X_k denote the strongly connected components of G_1 in topological sort order. (The topological sort order implies that if there is an edge from a vertex in X_i to a vertex in X_j, then $i \leq j$.)

Step 2: Apply the T4 transformation to each X_i in G. (That is, "collapse" each X_i to a single vertex w_i.) Let us denote the resulting graph G_1. (Note that the graph G_p is used only to identify the sets X_1 through X_k. The transformations themselves are applied starting with the graph G.)

Step 3: Visit the vertices w_1 to w_k of G_1 in that order. When vertex w_i is visited, check if the T2 transformation is applicable to it, and if so, apply the transformation. Let us denote the final graph produced (after w_k has been visited) by $\omega(G)$.

We will show later that $\omega(G)$ is $(T2+T4)^*(G)$. It is obvious that the complexity of the basic algorithm is linear in the size of the graph. As is the case with such algorithms, the actual complexity depends on implementation details, especially details such as how sets are implemented. It is straightforward to implement the algorithm so that it runs in linear time. Also note that the simple algorithm for identifying strongly connected components described in [3], due to Kosaraju and Sharir, directly generates the components in topological sort order.

Example

The example in Figure 1 illustrates our algorithm. Assume that we are interested in identifying the reaching definitions of the variable x for the graph G shown in Figure 1(i). For this problem, a vertex in the control-flow graph is a m-node iff it is the entry node or it contains a definition of the variable x. Let us assume that the nodes r, c, and g (shown as bold circles) are the m-nodes in G, and that the remaining nodes (shown as regular circles) are p-nodes.

Step 1: The first step of our algorithm is to identify the maximal strongly connected components of the subgraph of G consisting only of the p-nodes. In this example, G_p has only one non-trivial maximal strongly connected component, namely $\{a, d, e\}$. Each of the remaining p-nodes forms a strongly connected component consisting of a single vertex.

Step 2: The next step is to apply the T4 transformation to each of the strongly connected components identified in the previous step. The T4 transformation applied to a strongly connected component consisting of a single vertex (without any self loop) is the identity transformation, and, hence, we need to apply the T4 transformation only to $\{a, d, e\}$. Reducing this set of vertices to a new vertex w gives us the graph in Figure 1(ii). (In this and later figures, a vertex generated by merging a set X of vertices of the original graph is shown as a polygon enclosing the subgraph induced by the set X; this subgraph, shown using dashed edges and italic fonts, is not part of the transformed graph, but is shown only as an aid to the reader.)

Step 3: The next step is to visit the (possibly transformed) strongly connected components — that is, the set of vertices w, b, f, i, h, and j — in topological sort order, and try to apply the T2 transformation to each of them.

We first visit node w. Node w has two predecessors, namely r and c, and the T2 transformation is not applicable to w. We then visit b, which has only one predecessor, namely w. Hence, we apply the T2 transformation to b and obtain the graph shown in Figure 1(iii). We similarly apply the T2 transformation to nodes f and i (one after another), merging them both with w, and to node h, merging it with g. The T2 transformation is not applicable to the last node visited, j.

The graph in Figure 1(vi) is the normal form of G with respect to the T2 and T4 transformations.

4 On T2 and T4 Transformations

In this section we show that if we apply T2 and T4 transformations to a graph, in any order whatsoever, until no more applicable transformations exist, the resulting graph is unique. We also establish that our algorithm produces this unique "normal form". In what follows, a T transformation denotes either a T2 or T4 transformation.

Theorem 1 *Let τ_1 and τ_2 be two T transformations applicable to a graph G. Then there exists a T transformation τ_1' applicable to $\tau_2(G)$ and a T transformation τ_2' applicable to $\tau_1(G)$ such that $\tau_1'(\tau_2(G)) = \tau_2'(\tau_1(G))$.*

Proof

In what follows, we denote the vertex obtained by collapsing a set X of vertices by v_X.

If τ_1 and τ_2 involve disjoint transformations, this follows in a straightforward way. We just choose τ_1' to be τ_1 and τ_2' to be τ_2.

Let us now consider two overlapping T2 transformations. If τ_1 is $T2(u, v)$ and τ_2 is $T2(v, w)$, then we choose τ_1' to be $T2(u, v)$ (the same as τ_1) and τ_2' to be $T2(u, w)$ (the same as τ_2, but "renamed" to handle the merging of v with u).

Let us now consider two overlapping T4 transformations. If τ_1 is $T4(X)$ and τ_2 is $T4(Y)$, we choose τ_1' to be $T4(X - Y + v_Y)$ and τ_2' to be $T4(Y - X + v_X)$.

Let us now consider overlapping T2 and T4 transformations. If τ_1 is $T2(u, v)$ and τ_2 is $T4(Y)$, where $\{u, v\} \subseteq Y$, then we let τ_1' be the identity transformation, and τ_2' to be $T4(Y - X + v_X)$, where $X = \{u, v\}$.

If τ_1 is $T2(u, v)$ and τ_2 is $T4(Y)$, where $u \in Y$, then we let τ_1' be the $T2(v_Y, v)$, and τ_2' to be $T4(Y)$. □

It follows from the above theorem that T2 and T4 transformations form a finite Church Rosser system. Hence, every graph has a unique normal form with respect to these transformations. We now show that the graph $\omega(G)$ produced by our algorithm is this normal form.

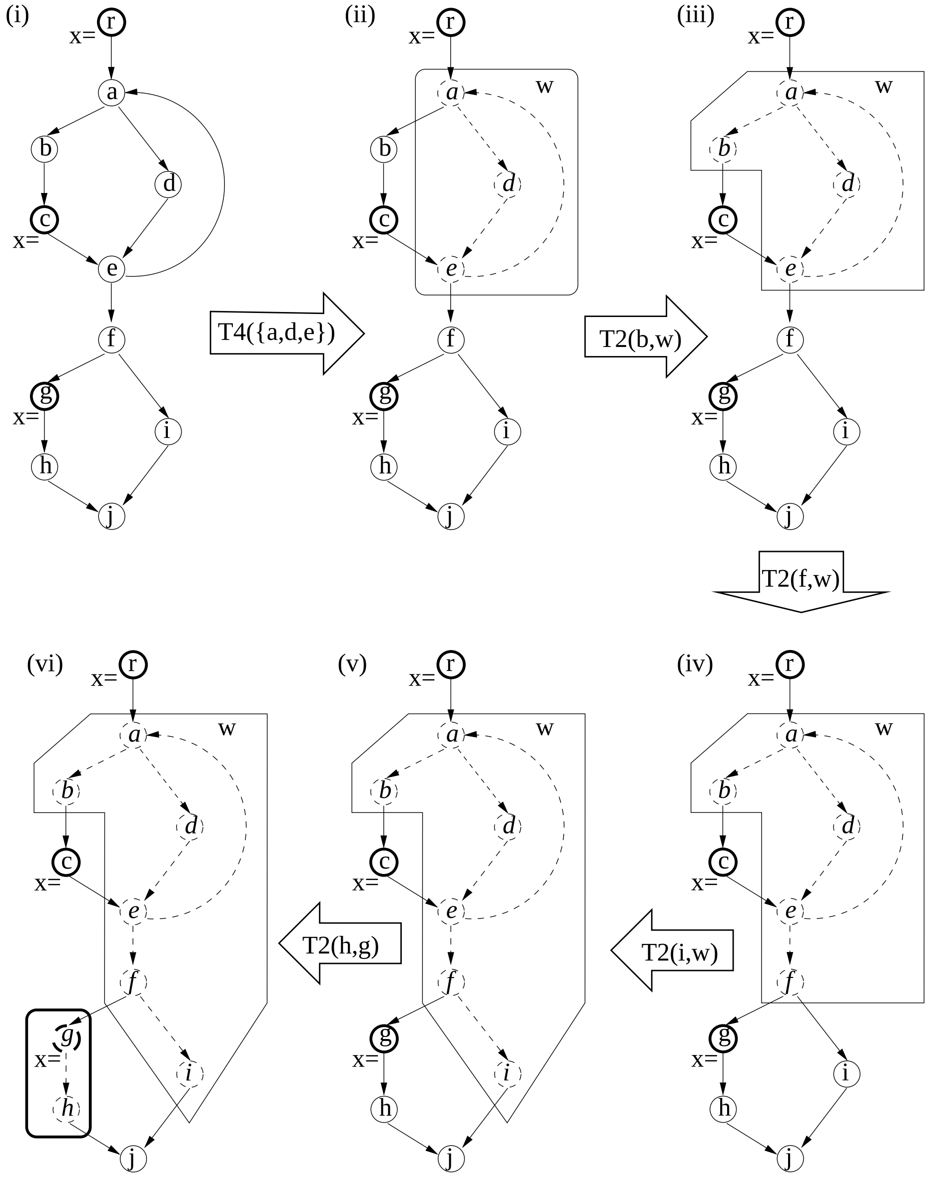

Figure 1: An example illustrating our algorithm for constructing the Compact Evaluation Graph.

Theorem 2 *No T2 or T4 transformations are applicable to $\omega(G)$.*

Proof

First observe that no (nontrivial) strongly connected set of p-nodes exists in the graph G_1. Hence, no T4 transformation is applicable to graph G_1. Clearly, the application of one or more T2 transformations to G_1 cannot create a nontrivial strongly connected set of p-nodes. Hence, no T4 transformation is applicable to the final graph $\omega(G)$ either.

Now, consider the construction of $\omega(G)$ from G_1. Assume that we find that the T2 transformation is not applicable to a p-node w_i when we visit node w_i. In other words, w_i has at least two predecessors when we visit it. Clearly any predecessor of w_i must be either a m-node or a node w_j where $j < i$. Subsequent T2 transformations can only eliminate a node of the form w_j where $j > i$. Hence, none of w_i's predecessors will be eliminated subsequently. Hence, the T2 transformation is not applicable to w_i in $\omega(G)$ either. □

5 A Comparison With Previous Equivalent Flow Graphs

In this section we compare CEG, the equivalent flowgraph produced by our algorithm to two previously proposed equivalent flow graphs, namely the Sparse Evaluation Graph (SEG) [2] and the Quick Propagation Graph (QPG) [11, 10]. We will show that both the Sparse Evaluation Graph and the Compact Evaluation Graph can be generated from the original graph by applying an appropriate sequence of T2 and T4 transformations. (Our goal is not to present algorithms to generate SEG or QPG; rather, it is to show that SEG and QPG are just two of the many equivalent flowgraphs that can be generated via T2-T4 transformations.) It follows that CEG is at least as small as SEG and QPG.

We begin by defining the Sparse Evaluation Graph. 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 dominance frontier of a vertex x, denoted $DF(x)$, is the set of all y such that x dominates some predecessor 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 $IDF_i(X)$ to be $DF(X)$ if $i = 1$ and $DF(X \cup IDF_{i-1}(X))$ if $i > 1$. The limit of this sequence is called the iterated dominance frontier of X, denoted $IDF(X)$.

Given a graph G and a set of vertices S, the Sparse Evaluation Graph consists[2] of the set of vertices V_{IDF} and the set of edges E_{IDF} defined as follows:

$$\begin{aligned} V_{IDF} &= S \cup IDF(S) \\ E_{IDF} &= \{x \rightarrow y \mid x, y \in V_{IDF}, \text{there exists a path from } x \text{ to } y \text{ in } G \text{ none of} \\ &\qquad \text{whose internal vertices are in } V_{IDF}\} \end{aligned}$$

For any vertex $u \in V_{IDF}$, define the set $partition(u)$ as follows:

$$\begin{aligned} partition(u) &= \{v \in (V - V_{IDF}) \mid \text{there exists a path from } u \text{ to } v \text{ in } G \text{ none of} \\ &\qquad \text{whose internal vertices are in } V_{IDF}\} \end{aligned}$$

Let $partition^+(u)$ denote the set $\{u\} \cup partition(u)$.

Lemma 1 . *Let $u \in V_{IDF}$. Then, u dominates every vertex in $partition(u)$.*

Proof

Let $u \rightarrow v_1 \cdots \rightarrow v_k$ be a path in G such that none of the v_i is in V_{IDF}. We will show that u dominates v_i, for $1 \leq i \leq k$ by induction on i.

Consider $i = 1$. If u does not dominate v_1, then v_1 is in $DF(u)$, by definition. Hence, v_1 must be in V_{IDF}, contradicting our assumption. (This is because $DF(IDF(X)) \subseteq IDF(X)$.)

[2] Choi *et al.* also discuss a couple of simple optimizations that can be further applied to the SEG, which we ignore here. We will discuss these later, in Section 7.

Now consider any $i > 1$. We know from the inductive hypothesis that u dominates v_{i-1}. If u does not dominate v_i, then v_i is in $DF(u)$, by definition. Hence, v_i must be in V_{IDF}, contradicting our assumption.

The result follows. □

Lemma 2 *Let u and v be two different vertices in V_{IDF}. Then, partition(u) and partition(v) are disjoint.*

Proof
Since domination is an antisymmetric relation, either u does not dominate v or v does not dominate u. Assume without loss of generality that u does not dominate v. This implies that there exists a path α from the entry vertex to v that does not contain u.

Consider any w in *partition*(v). By definition, there exists a path β from v to w none of whose internal vertices are in V_{IDF}. In particular, β does not contain u.

The concatenation of α and β is a path from the entry vertex to w that does not contain u. Hence, u does not dominate w. It follows from Lemma 1 that w is not an element of *partition*(u). The result follows. □

Lemma 3 *If v is in partition(u), then any predecessor w of v must be in partition$^+$(u).*

Proof
Recall that we assume that every vertex in the control-flow graph is reachable from the entry vertex. Consider any path α from the entry vertex to w and let z be the last vertex in α that belongs to V_{IDF}. This implies that w belongs to *partition*$^+(z)$. But this also implies that v is in *partition*$^+(z)$, by definition of *partition*(z). Hence, z must be u (from Lemma 2). □

Theorem 3 *The Sparse Evaluation Graph can be produced from the original control-flow graph by applying an appropriate sequence of T2 and T4 transformations.*

Proof
We first show that for any vertex u in V_{IDF}, the whole of *partition*(u) can be merged into u through a sequence of T2 and T4 transformations.

Consider the subgraph induced by *partition*(u). Let $C_1 \cdots C_k$ denote the strongly connected components of this subgraph, in topological sort order. Reduce every C_i to a vertex w_i using a T4 transformation. Let H_u denote the resulting graph.

Now apply the T2 transformation to vertices w_1 to w_k in that order. The T2 transformation will be applicable to each w_i for the following reason.

Note that Lemma 3(b) implies that any predecessor of w_i, in the graph H_u, must be either u or some w_j where $j < i$. Hence, once we apply the T2 transformation to vertices w_1 to w_{i-1}, w_i will have u as its unique predecessor. Hence, we can apply the T2 transformation to w_i as well and merge it with u.

It is clear that at the end of this process every vertex in *partition*(u) has been merged into u. If we repeat this process for every vertex u in V_{IDF}, clearly the resulting graph is the same as the Sparse Evaluation Graph. □

Corollary 1 *The Compact Evaluation Graph can be generated from the Sparse Evaluation Graph by applying an appropriate sequence of T2 and T4 transformations.*

Note that the application of either T2 or T4 can only make the graph smaller. (Both transformations reduce the number of nodes and the number of edges in the graph.) Hence, the above corollary implies that the representation produced by our algorithm is at least as sparse as the one produced by Choi et al.'s algorithm.

Figure 2 shows the difference between the Compact Evaluation Graph and the Sparse Evaluation Graph for the example graph G presented in Figure 1. As can be seen, the Compact Evaluation Graph can be generated from the Sparse Evaluation Graph by applying the transformation $T4(\{x, y\})$.

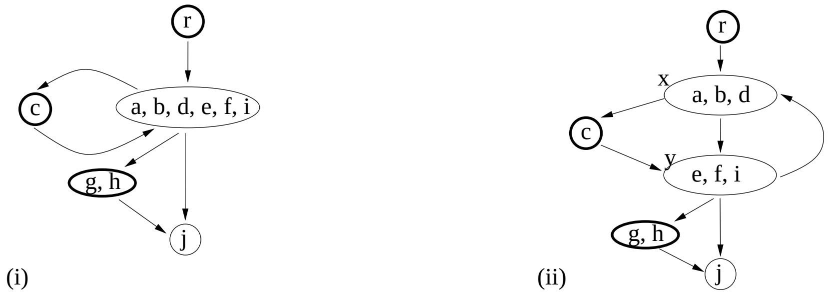

Figure 2: (i) The Compact Evaluation Graph, produced by our algorithm. (ii) The Sparse Evaluation Graph, produced by previous algorithms.

We can also establish results analogous to the above for the Quick Propagation Graph defined in [10]. The Quick Propagation Graph is based on the concept of single-entry single-exit regions. Every single-entry single-exit R has a unique entry edge $u \rightarrow v$ such that it is the only edge from a vertex outside R to a vertex inside R. Let us refer to the vertex u as the entry vertex of R. We can show, just as in the proof of Theorem 3, that any single-entry single-exit region R consisting only of p-nodes can be merged with its entry vertex using T2 and T4 transformations. Since the Quick Propagation Graph is constructed precisely by merging single-entry single-exit regions consisting only of p-nodes with their entry vertices, we have:

Theorem 4 *The Quick Propagation Graph can be produced from the original control-flow graph by applying an appropriate sequence of T2 and T4 transformations.*

6 On Minimum Size Equivalent Flow Graphs

We have now seen three different graphs, namely SEG, QPG, and CEG, that can all serve as "equivalent flow graphs", *i.e.* help speed up dataflow analysis through sparse evaluation techniques. This raises the question: what, exactly, is an equivalent flow graph? In particular, is it possible to construct a equivalent flow graph of minimum size efficiently? In this section, we attempt to address these questions by presenting one possible definition of equivalent flow graphs.

Let S be a set of vertices in a graph G. Given a path $\alpha = [x_1, \cdots, x_m]$, we define the S-projection of α, denoted $project_S(\alpha)$, to be the subsequence $[x_{i_1}, \cdots, x_{i_k}]$ of α consisting of only vertices in S. Let σ be some arbitrary sequence of elements of S. We say that σ is an S-path between vertices x and y iff there exists a path α between vertices x and y whose S-projection is σ. We will use the notation $x[s_1, \cdots, s_k]^S y$ to denote the fact that there is an S-path $[s_1, \cdots, s_k]$ from x to y, usually omitting the superscript S as it will be obvious from the context.

Definition 1 *Let f be a function from the set of vertices of G to the set of vertices of another graph H. Let $f(S)$ denote the set $\{f(x) \mid x \in S\}$. We say that the $\langle f, H\rangle$ preserves S-paths if*

(i) $f(entry(G)) = entry(H)$.
(ii) f is one-to-one with respect to S: $\forall x, y \in S.(x \neq y \Rightarrow f(x) \neq f(y))$, and
(iii) for any vertex y in graph G, $[s_1, \cdots, s_k]$ is a S-path between $entry(G)$ and y in G iff $[f(s_1), \cdots, f(s_k)]$ is a $f(S)$-path between $entry(H)$ and $f(y)$ in H.

We say that a dataflow analysis problem instance over a graph G is S-restricted if the transfer function associated with any vertex not in S is the identity function.

With the above definition, one can show that if $\langle f, H\rangle$ preserves S-paths, then for any S-restricted dataflow analysis problem instance over graph G, the MOP or MFP solution for G can be recovered from the MOP or MFP solution for H.

Theorem 5 *Let G be a control-flow graph and S a set of vertices in G that includes the entry vertex of G. Let $\langle f, H\rangle$ preserve S-paths. Let $\mathcal{F} = (\mathcal{L}, G, M, c)$ be an S-restricted dataflow analysis problem instance over*

G. Define $\mathcal{F}'$ to be $(\mathcal{L}, H, M', c)$ where $M'(u)$ is defined to be $M(x)$ if $u = f(x)$ for some $x \in S$, and the identity function otherwise. Then, for every vertex u in G,

$$\begin{aligned} MOP_{\mathcal{F}}(u) &= MOP_{\mathcal{F}'}(f(u)) \\ MFP_{\mathcal{F}}(u) &= MFP_{\mathcal{F}'}(f(u)) \end{aligned}$$

Proof

Note that for any path α in G, $M(\alpha) = M(project_S(\alpha))$, since the transfer functions associated with vertices not in S is the identity function. Let r denote the entry vertex of G. Let S-Paths(r, u) denote the set of all S-paths from r to u. Obviously,

$$\begin{aligned} MOP_{\mathcal{F}}(u) &= \textstyle\bigsqcap_{p \in Paths(entry(G), u)} M(p)(c) \\ &= \textstyle\bigsqcap_{p \in \text{S-Paths}(r,u)} M(p)(c) \end{aligned}$$

Since $\langle f, H \rangle$ preserve S-paths, it follows trivially that $MOP_{\mathcal{F}}(u) = MOP_{\mathcal{F}'}(f(u))$.

Let us now consider the dataflow equations induced by $\mathcal{F}$. Let u be a vertex not in S. Since the transfer function associated with u is the identity function, the equation associated with u

$$x_u = M(u)(\bigsqcap_{v \rightarrow u} x_v)$$

reduces to

$$x_u = \bigsqcap_{v \rightarrow u} x_v$$

Let us now eliminate from the right hand side of all equations any variable x_u associated with a vertex not in S. The elimination is slightly complicated if there are cycles involving vertices not in S. But if we have a cycle consisting only of vertices in S, the equations for the vertices in the cycle together imply that $x_u = x_v$ for any two vertices u and v in the cycle. Hence, mutually recursive equations induced by vertices not in S can be converted into self-recursive equations, and then the self-recursion can be eliminated. The elimination process finally transforms the equation associated with any vertex w into

$$x_w = M(w)(\bigsqcap_{v \in \{s \mid s[sw]w \text{ or } s[s]w\}} x_v)$$

Note that the meet is over the set of all vertices s in S that can reach w through a path consisting of no vertices in S other than its endpoints. Since we assume that every vertex in the graph is reachable from the entry vertex, it is clear that $s[sw]w$ iff $entry(G)[\alpha sw]w$ for some path α. It is clear that the dataflow equations of both $\mathcal{F}$ and $\mathcal{F}'$ are isomorphic when reduced to this simple form. Hence, the maximal fixed point solution of both $\mathcal{F}$ and $\mathcal{F}'$ are the same.

$\square$

The above theorem shows that the conditions of Definition 1 are sufficient to ensure that the dataflow solution for G can be recovered from the dataflow solution for H. It can also be argued that these conditions are necessary, in fact, for a theorem like the above to hold. Obviously, we need condition (i) of Definition 1. Further, for any two vertices in S, it is trivial to construct an S-restricted dataflow analysis problem instance over G such that the solution at the two vertices are different. Hence, clearly condition (ii) is also necessary. Similarly, if for any vertex y in G the set of S-paths between $entry(G)$ and y don't correspond to the set of $f(S)$-paths between $entry(H)$ and $f(y)$, it is again trivial to construct a S-restricted dataflow analysis problem instance over G such that the solution at y differs from the solution at $f(y)$. Hence, we may define:

Definition 2 *Given a graph G, and a set S of vertices in G, we say that $\langle f, H \rangle$ is an equivalent flow graph of G with respect to S iff $\langle f, H \rangle$ preserves S-paths.*

6.1 An Algorithm For Constructing Vertex Minimal Equivalent Flow Graphs

We now present a simple algorithm for constructing an equivalent flow graph consisting of the minimum number of vertices possible. The algorithm runs in $O(|S|(|V|+|E|))$ time, where $|S|$ denotes the number of m-nodes in the graph, while $|V|$ and $|E|$ denote the number of vertices and edges in the graph.

We begin with some notation that will be helpful in relating algorithms based on collapsing multiple vertices into a single vertex, such as our earlier algorithm based on T2 and T4 transformations, to the notion of equivalent flow graphs introduced above. Let $\cong$ be an equivalence relation on the vertices of a graph G. Let $\mathcal{C}_{\cong}$ denote the set of equivalence classes of $\cong$. Let $[u]_{\cong}$ denote the equivalence class to which vertex u belongs. Let $[]_{\cong}$ denote the function from $V(G)$ to $\mathcal{C}_{\cong}$ that maps every vertex to its equivalence class. We may occasionally omit the subscript $\cong$ to reduce notational clutter. We now define the *quotient graph* obtained by collapsing every equivalence class into a single vertex. The following definition depends on the set S of m-nodes and is not the most obvious or natural definition, but the reason for the definition will become apparent soon.

The graph $G/_S\cong$ is the graph H whose vertex set and edge set are as below:

$$\begin{aligned}
V(H) &= \mathcal{C}_{\cong} \\
E(H) &= \{[u] \to [v] \mid u \to v \in E(G) \text{ and } ((v \in S) \text{ or } ((\not\exists s \in S. v \cong s) \text{ and } (u \not\cong v)))\} \\
\mathit{entry}(H) &= [\mathit{entry}(G)]
\end{aligned}$$

The definition of $E(H)$ deserves some explanation. The basic idea is that an edge $u \to v$ of G will become the edge $[u] \to [v]$ in the collapsed graph. However, the above definition ensures that certain edges are eliminated completely. In particular, if $u \cong v$, the corresponding edge $[u] \to [v]$ will be a self loop, and is retained only if $v \in S$. Similarly, if $v \cong s$ for some $s \in S$, then an edge directed to v is projected only if $v \in S$. Otherwise, it is eliminated.

Note that T2 and T4 transformations can be viewed as simple quotient graph constructions. In particular, $T2(u, v)$ corresponds to the equivalence relation that places u and v into the same equivalence class and every other vertex in an equivalence class by itself. Similarly, $T4(X)$ corresponds to an equivalence relation in which all vertices in X are equivalent to each other, while every vertex not in X is in an equivalence class by itself. A sequence of such transformations corresponds to an equivalence relation too, namely the transitive closure of the union of the equivalence relations associated with the individual transformations in the sequence. Hence, the compact evaluation graph itself is the quotient graph with respect to an appropriate equivalence relation.

We now define a particular equivalence relation $\cong_S$ induced by set S. Define $\mathit{pred}_S(u)$ to be the set of vertices $s \in S$ such that there exists an S-path $[s]$ from s to u. Note that for any $s \in S$, $\mathit{pred}_S(s) = \{s\}$. We say that $u \cong_S v$ iff $\mathit{pred}_S(u) = \mathit{pred}_S(v)$. Our algorithm identifies the equivalence classes of the above equivalence relation, and collapses each equivalence class to a single vertex. More formally, our algorithm produces the equivalent flow graph $\langle []_{\cong_S}, G/_S\cong_S \rangle$.

We will first establish the minimality claim.

Theorem 6 *If $\langle f, H \rangle$ preserves S-paths, then*

$$f(x) = f(y) \Rightarrow x \cong_S y$$

Proof

Let $\langle f, H \rangle$ preserve S-paths and assume that $f(x) = f(y)$. Then,

$$\begin{array}{rll}
w \in \mathit{pred}_S(x) \Leftrightarrow & [w] \text{ is a } S\text{-path from } w \text{ to } x & \text{definition of } \mathit{pred}_S(x) \\
\Leftrightarrow & [f(w)] \text{ is a } f(S)\text{-path from } f(w) \text{ to } f(x) & (\text{since } \langle f, H \rangle \text{ preserve } S\text{-paths}) \\
\Leftrightarrow & [f(w)] \text{ is a } f(S)\text{-path from } f(w) \text{ to } f(y) & (\text{since } f(x) = f(y)) \\
\Leftrightarrow & [w] \text{ is a } S\text{-path from } w \text{ to } y & (\text{since } \langle f, H \rangle \text{ preserve } S\text{-paths}) \\
\Leftrightarrow & w \in \mathit{pred}_S(y) & \text{definition of } \mathit{pred}_S(y)
\end{array}$$

Hence $\mathit{pred}_S(x) = \mathit{pred}_S(y)$ and $x \cong_S y$. □

It follows from the above theorem that we cannot construct an equivalent flow graph with fewer vertices than $\langle []_{\cong_S}, G/_S{\cong_S}\rangle$. We now just need to show that $\langle []_{\cong_S}, G/_S{\cong_S}\rangle$ *is* an equivalent flow graph. The following theorem establishes a more general result, namely that for any equivalence relation $\cong$ that approximates (refines) $\cong_S$, the quotient graph with respect to $\cong$ is an equivalent flow graph.

Theorem 7 *$\langle []_{\cong}, G/_S{\cong}\rangle$ preserves S-paths iff*

$$x \cong y \Rightarrow x \cong_S y$$

Proof
Let f denote $[]_{\cong}$ and let H denote $G/_S{\cong}$. The forward implication of the theorem follows directly from Theorem 6. Consider the reverse implication. The first two conditions (of Definition 1) follow trivially, and we need to show that the third condition holds too.

Recall that $x[s_1, \cdots, s_k]^S y$ denotes the fact that there is an S-path $[s_1, \cdots, s_k]$ from x to y. Let r denote the entry vertex of G. We need to show that

$$r[s_1, \cdots, s_k]y \Leftrightarrow f(r)[f(s_1), \cdots, f(s_k)]f(y)$$

$\Rightarrow$ We will establish the forward implication by induction on the length of the path from r to y. The base case is trivial since we have $f(r)[f(r)]f(r)$. For the inductive step assume that we have a path P from r to y, consisting of n edges, whose S-projection is $[s_1, \cdots, s_k]$.

First consider the case where $y \cong s_k$ and $y \neq s_k$. Consider the prefix of path P ending at vertex s_k. This is a path from r to s_k consisting of less than n edges whose S-projection is $[s_1, \cdots, s_k]$. It follows from the inductive hypothesis that $f(r)[f(s_1), \cdots, f(s_k)]f(s_k)$. Since $f(y) = f(s_k)$, it follows that $f(r)[f(s_1), \cdots, f(s_k)]f(y)$.

Now consider the remaining case, where $y = s_k$ or $y \not\cong s_k$. Note that the presence of path P implies that $s_k[s_k]y$. Hence, if $y \not\cong s_k$, then $\nexists s \in S.s \cong y$. If $x \rightarrow y$ is the last edge of path P, we have $r[s_1, \cdots, s_k]x$ via a path of $n-1$ edges. It follows from the inductive hypothesis that $f(r)[f(s_1), \cdots, f(s_k)]f(x)$. If $x \cong y$, then $f(x) = f(y)$ and we are done. Otherwise, H includes the edge $f(x) \rightarrow f(y)$, and it follows that $f(r)[f(s_1), \cdots, f(s_k)]f(y)$ also.

$\Leftarrow$ We will establish the reverse implication by induction on the length of the path from $f(r)$ to $f(y)$ in H. Again, the base case is trivial since we have $r[r]r$. For the inductive step assume that we have a path P from $f(r)$ to $f(y)$ of length n edges whose $f(S)$ projection is $[f(s_1), \cdots, f(s_k)]$. We will establish that $r[s_1, \cdots, s_k]y$ in two steps.

Proof that $r[s_1, \cdots, s_k]s_k$. First consider the case that $y \cong s_k$. The last edge of P must be of the form $f(x) \rightarrow f(s_k)$ where $x \rightarrow s_k$ is an edge in G. Since we have a path of less than $n-1$ edges from $f(r)$ to $f(x)$ whose S-projection is $[f(s_1), \cdots, f(s_{k-1})]$, it follows from the inductive hypothesis that $r[s_1, \cdots, s_{k-1}]x$, which together with the edge $x \rightarrow s_k$ implies that $r[s_1, \cdots, s_k]s_k$. Now consider the case that $y \not\cong s_k$. Then, we have a path of less than $n-1$ edges from $f(r)$ to $f(s_k)$ whose S-projection is $[f(s_1), \cdots, f(s_k)]$. It follows from the inductive hypothesis that $r[s_1, \cdots, s_k]s_k$.

Proof that $s_k[s_k]y$. Consider the suffix of path P from $f(s_k)$ to $f(y)$. This suffix can be written in the form $f(u_1) \rightarrow f(u_2) \cdots f(u_j)$ where $s_k \cong u_1$, $u_1 \rightarrow u'_2$, $u'_2 \cong u_2$, $\cdots$, $u'_j \cong u_j$, and $u_j = y$. It immediately follows that $s_k[s_k]u_i$ for $1 \leq i \leq j$. In particular, this implies that there is a path Q from s_k to y in G whose S-projection is $[s_k]$.

It follows that $r[s_1, \cdots, s_k]y$.

□

It is easy to verify that the equivalence relations corresponding to T2 and T4 transformations are approximations of $\cong_S$. Hence, the above theorem shows that $\langle []_{\cong_S}, G/_S{\cong_S}\rangle$, the Compact Evaluation Graph, the Sparse Evaluation Graph, and the Quick Propagation Graph are all valid equivalent flow graphs.

Identifying the equivalence classes of $\cong_S$

We now present an efficient algorithm for identifying the equivalence classes of $\cong_S$. The algorithm is a partitioning algorithm similar to Hopcroft's algorithm for minimizing finite automata.

We initially start out with a partition in which all nodes are in a single equivalence class. We then refine the partition by considering every node in S, one by one. For every node m in S, we first perform a traversal of the graph to identify R_m, the set of all nodes reachable from m without going through another node in S. These are the nodes u such that $pred_S(u)$ contains m. Then, every equivalence class X is refined into two equivalence classes $X \cap R_m$ and $X - R_m$, if both these sets are nonempty. This refinement ensures that for any two vertices x and y left in the same equivalence class, $m \in pred_S(x)$ iff $m \in pred_S(y)$. Hence, once the refinement has been done with respect to every vertex in S, $pred_S(x) = pred_S(y)$ for any two vertices x and y left in the same equivalence class.

The refinement of the partition, for a given node m, can be done in linear time, if appropriate data structures are used. (For example, by maintaining each equivalence class as a doubly linked list, so that an element can be removed from an equivalence class in constant time.) Consequently, the final partition can be constructed in time $O(|S|(|V| + |E|))$.

In practice, it might be more efficient to first construct the compact evaluation graph, using the linear time algorithm, and to then apply the above quadratic algorithm to the smaller compact evaluation graph.

This algorithm is similar in spirit to the work of Duesterwald *et al.* [8], who present both an $O(|V| \log |V|)$ algorithm and an $O(|V|^2 \log V)$ partitioning based algorithm for constructing equivalence graphs. (This complexity measure is based on the assumption that the number of edges incident on a vertex is bounded by a constant.) Both Duesterwald *et al.*'s algorithm and Hopcroft's algorithm utilize the edges of the graph to refine partitions, while our algorithm uses paths in the graph consisting only of p-nodesto do the refinement step. This guarantees that the graph produced by our algorithm has the minimum number of vertices possible, which is not the case with Duesterwald *et al.*'s algorithm.

6.2 Constructing Edge Minimal Equivalent Flow Graphs is NP-hard

We now show that the problem of constructing the smallest equivalent flow graph becomes much more difficult if one counts the number of edges in the graph as well. Define the size of $\langle f, H\rangle$ to be the sum of the number of nodes and the number of edges in H.

Theorem 8 *The problem of finding an equivalent flow graph of minimum size is NP-hard.*

Proof

(Reduction from the set-covering problem.) The set-covering problem ([3]) is the following: Given a finite $(X, \mathcal{F})$, where $X = \bigcup_{S\in\mathcal{F}} S$, find a minimum-size subset C of $\mathcal{F}$ such that $X = \bigcup_{S\in C} S$. The set-covering problem is known to be NP-hard. We now show that given an instance $(X, \mathcal{F})$ of the set-covering problem, we can construct in polynomial time a graph G such that a minimum-size cover for $(X, \mathcal{F})$ can be generated (in polynomial time) from a minimum-size equivalent flow graph for G. We assume that the input instance is such that $X \notin \mathcal{F}$. Otherwise, $\{X\}$ is trivially the minimum-size cover for $(X, \mathcal{F})$.

The graph G consists of a m-node r (the entry vertex), a m-node m_x for every $x \in X$, a p-node p_S for every $S \in \mathcal{F}$, and a p-node *exit*. The graph consists of an edge from r to every m_x, an edge from m_x to p_S iff $x \in S$, and an edge from every p_S to *exit*.

Let $\langle f, H\rangle$ be a minimum size equivalent flow graph for G. Assume that every predecessor of $f(exit)$ in H is some vertex of the form $f(p_{S_w})$. If this is not the case, then H can be trivially modified as follows, without increasing its size, to ensure this. Consider the vertex $f(exit)$ in H. Let w be any predecessor of $f(exit)$ in H. We claim that there must be some $f(p_{S_w})$ reachable from w.

> **Claim**: There exists some $f(p_{S_w})$ reachable from w.
> **Proof**: Consider the following cases:
> *Case 1: w is $f(r)$.* This is not possible, since $\langle f, H\rangle$ preserves S-paths.
> *Case 2: w is $f(m_x)$, for some x.* Clearly, x must be in some set $S \in \mathcal{F}$. Hence, there must exist an edge from m_x to p_S in G. Since $\langle f, H\rangle$ preserves S-paths, there must exist some path from $f(m_x)$ to $f(p_S)$ in H.

Case 3: w is $f(p_S)$, for some S. The result trivially follows.
Case 4: w is $f(exit)$. This is not possible, since we can drop the edge from $f(exit)$ to itself to get a smaller equivalent flow graph.
Case 5: w is not $f(u)$, for any u. If no $f(p_S)$ is reachable from w, then we could simply merge w with $f(exit)$ to generate a smaller equivalent flow graph. Hence, there must exist a $f(p_S)$ reachable from w.

Now, let us replace every predecessor w of $f(exit)$ by $f(p_{S_w})$. This gives a minimum size equivalent flow graph in which all predecessors of $f(exit)$ are of the form $f(p_S)$.

It can be shown that the set $\{S \in \mathcal{F} \mid f(p_S) \rightarrow f(exit) \text{ is an edge in } H\}$ is a minimum size cover for $(X, \mathcal{F})$. Clearly, the conditions for S-path preservation imply that this set must be cover for $(X, \mathcal{F})$. If it is not a minimum size cover, let $C \subseteq \mathcal{F}$ be a minimum size cover for $(X, \mathcal{F})$. Replace the predecessors of $f(exit)$ by the set $\{f(p_S) | S \in C\}$. This will give us a smaller equivalent flow graph, contradicting our assumption that $\langle f, H \rangle$ is a minimum size equivalent flow graph. □

6.3 Discussion

Let us look at the results we have seen so far from a slightly different perspective. We have seen that cycles involving p-nodes are irrelevant and may be eliminated (*e.g.*, via T4 transformations). Once such cycles are eliminated, the problem of constructing equivalent flow graphs becomes similar to a well-known problem: minimizing the computation required to evaluate a set of expressions over a set of variables. The one additional factor we need to consider is that the only operator allowed in the expressions is the meet operator, which is commutative, associative, and idempotent. For example, the problem may be viewed as that of minimizing a boolean circuit consisting only of, say, the boolean-and operator.

Our algorithm for constructing the vertex minimal equivalent flow graph essentially eliminates common subexpressions. From the point of view of performing dataflow analysis, this achieves the "best possible" space reduction one might hope for (since iterative algorithms typically maintain one "solution" for every vertex in the graph). This also reduces the number of "meet operations" the iterative algorithm needs to perform in order to compute the final solution, but not to the least number necessary. Eliminating further "unnecessary" edges from the graph can reduce the number of meet operations performed by the analysis algorithm, though it will not provide further space savings.

This helps to place the above NP-hardness result in perspective, indicating what can be achieved efficiently and what cannot.

There is yet another question concerning the significance of the above NP-hardness result. Our feeling is that it may be simpler to generate the minimum size equivalent flow graph for control-flow graphs generated from structured programs than to do it for graphs generated from unstructured programs and that the NP-hardness result might not hold if we restrict attention to structured control-flow graphs. Consider the example in Figure 3(i). As before, m-nodes are shown as bold circles. This graph is already in normal form with respect to both T2 and T4 transformations. Hence, the compact evaluation graph of this graph is itself. However, a smaller equivalent flow graph exists for this input graph, as shown in Figure 3(ii).

Note, however, that the graph in Figure 3(i) *cannot* be generated using structured programming constructs. In contrast, consider the graph shown in Figure 3(iii). This graph can be generated using only structured constructs such as `CASE` statements and `If-Then-Else` statements. The nodes e, f, and g of this graph have the same solution as the corresponding nodes in Figure 3(i). In this case, however, our linear time T2-T4 based algorithm *will* be able to reduce this graph to the normal form shown in Figure 3(ii) !

An interesting question that arises is whether it is simpler to generate the minimum size equivalent flow graph for control-flow graphs of structured programs. In particular, does the NP-hardness result hold if we restrict attention to structured control-flow graphs?

7 Partially Equivalent Flow Graphs

Note that the equivalent flow graphs we have considered so far permit the dataflow solution for any vertex in the original graph to be recovered from the sparse graph. In general, we may not require the dataflow

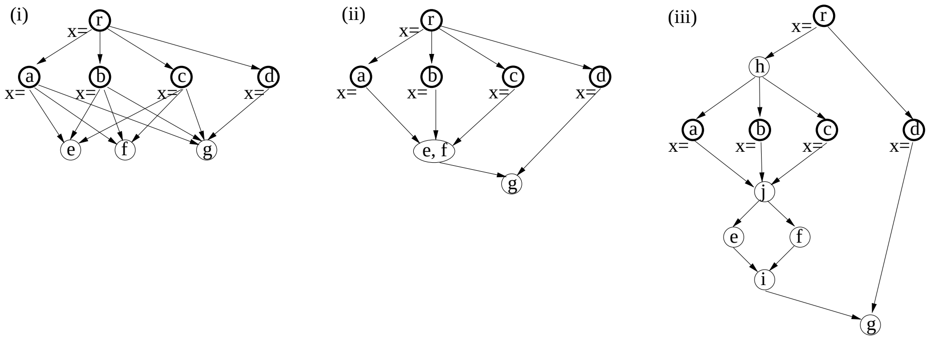

Figure 3: An example illustrating the kind of factoring that our algorithm does not attempt to achieve

solution at every vertex. For example, if we are solving the reaching definitions problem for a variable x, the solution will usually be necessary only at nodes that contain a use of the variable x. One can use this fact to construct graphs that are even more compact than the equivalent flow graphs. We will refer to such generalized graphs that allow us to recover the dataflow solution for a specified set of vertices in the original graph as *Partially Equivalent Flow Graphs.*

Let us refer to a node where the dataflow solution is required as a r-node and to a node where the dataflow solution is not required as a u-node. Let us refer to a node that is both a p-node and a u-node as a up-node. We now define some transformations that can be used in the construction of a Partially Equivalent Flow Graph.

More Transformations

Transformation T5: The T5 transformation is applicable to a node u if (i) u is an up-node, and (ii) u has a unique successor. The T5 transformation is structurally the same as a T2 transformation. It simply merges the node u, to which it is applicable, with u's unique successor. Let v denote u's successor. The graph $T5(u,v)(G)$ is obtained by removing the node u and the edge $u \to v$ from the graph G and by replacing every incoming edge $w \to u$ of u by a corresponding edge $w \to v$.

Note that the dataflow solution for node u in graph G cannot be, in general, obtained from the dataflow solution for any node in $T5(u,v)(G)$. However, this is okay since the dataflow solution at u is not required.

Transformation T6: The T6 transformation is applicable to any set of u-nodes that has no successor. (A set X of nodes is said to have no successor if there exists no edge from a node in X to a node outside X.) If X is a set of u-nodes that has no successor in G, then the graph $T6(X)(G)$ is obtained from G by deleting all nodes in X as well as any edges incident on them.

The T6 transformation is rather simple: it says that a node can be deleted if that node and all nodes reachable from that node are u-nodes. This is similar to the pruning of dead ϕ-nodes discussed in [19, 2] but more general.

We now outline a transformation that essentially captures an optimization described by Choi *et al* [2]. This optimization, however, requires us to relax our earlier condition that the Partially Equivalent Flow Graph is to be constructed knowing nothing about the transfer functions associated with the m-nodes. Assume that we further know whether the transfer function associated with a m-node is a constant-valued function or not. (For example, in the problem of identifying the reaching definitions of a variable x, every m-node has a constant-valued transfer function, since it *generates* the single definition of x contained in that node and *kills* all other definitions of x.) Let us refer to a m-node as a c-node if the transfer function associated with that node is a constant-valued function.

Transformation T7: The T7 transformation is applicable to any c-node that has one or more incoming edges, and the transformation simply deletes these incoming edges.

The T7 transformation may not preserve the meet-over-all-paths solution since it creates vertices unreachable from the entry vertex. However, it does preserve the maximal fixed point solution.

Theorem 9 *The T2, T4, T5, T6, and T7 transformations form a finite Church-Rosser system.*

Proof
Tedious, but straightforward. □

The Algorithm

Luckily, the transformations do not significantly interact with each other. Let us denote the normal form of a graph G with respect to the set of all T2, T4, T5, T6, and T7 transformations by $(T2{+}T4{+}T5{+}T6{+}T7)^*(G)$. Let $T5^*(G)$ denote the normal form of G with respect to the set of all T5 transformations. $T6^*(G)$, $T7^*(G)$, $T2^*(G)$, and $T4^*(G)$ are similarly defined. We can show that:

Theorem 10 $(T2 + T4 + T5 + T6 + T7)^*(G) = T5^*(T6^*(T7^*(T2^*(T4^*(G)))))$

Proof
We will sketch the outline of a proof and omit details.

Assume that a graph is in normal form with respect to T4 transformations. In other words, it does not have any nontrivial (that is, of size > 1) strongly connected set of p-nodes. Clearly, the application of a T5 transformation will not create any nontrivial strongly connected set of p-nodes. Hence, the graph will continue to be in normal form with respect to T4 transformation even after the application of a T5 transformation. Similarly, the graph will continue to be in normal form with respect to T4 transformations even after the application of a T6 or a T7 or a T2 transformation.

Now assume that a graph is in normal form with respect to T2 transformations. One can show that the graph will continue to be in normal form with respect to T2 transformations even after the application of a T5 or T6 or T7 transformation.

Similarly, a graph in normal form with respect to T7 transformations will continue to be so even after the application of a T6 or T5 transformation. And a graph in normal form with respect to T6 transformations will continue to be so after the application of a T5 transformation.

This establishes that $T5^*(T6^*(T7^*(T2^*(T4^*(G)))))$ is in normal form with respect to all the transformations. □

We now present our algorithm for constructing a partially equivalent flow graph for a given graph G.

Step 1. Compute $G_1 = T4^*(G)$ (as outlined earlier).

Step 2. Compute $G_2 = T2^*(G_1)$ (as outlined earlier).

Step 3. Compute $G_3 = T7^*(G_2)$, by simply deleting all incoming edges of every c-node in G_2.

Step 4. Compute $G_4 = T6^*(G_3)$ as follows: perform a simple backward graph traversal from every r-node to identify the set X of nodes from which a r-node is reachable. Delete all other nodes and edges incident upon them.

Step 5. Compute $G_5 = T5^*(G_4)$ as follows: Let $w_1, \cdots, w_k$ be the set of up-nodes of G_4 in topological sort order. Since G_4 is in normal form with respect to T4 transformations, it cannot have any cycle of p-nodes, and hence a topological sort ordering of the up-nodes must exist. Visit vertices w_k to w_1, in that order, applying the T5 transformation to any w_i that has only one successor.

The graph G_5 can be shown to be in normal form with respect to all the transformations described earlier.

Example

Figure 4 illustrates the construction of a partially equivalent flow graph using our algorithm. Assume that all applicable T2 and T4 transformations have been applied to the initial graph using the algorithm outlined earlier, and that the resulting graph is as shown in Figure 4(i). Assume that we are interested in the dataflow solution only at nodes e and i (shown as square vertices in the figure). All the remaining nodes (shown as circles) are u-nodes. We also assume that all the m-nodes have a constant transfer function.

The next step in computing the partially equivalent flow graph is applying all possible T7 transformations. This produces the graph shown in Figure 4(ii). We then apply all feasible T6 transformations, which produces the graph in Figure 4(iii).

We then examine all remaining up-nodes in reverse topological sort order, applying the T5 transformations where possible. It turns out that the T5 transformation is applicable to both f and d, and applying these transformations produces the normal form in Figure 4(v).

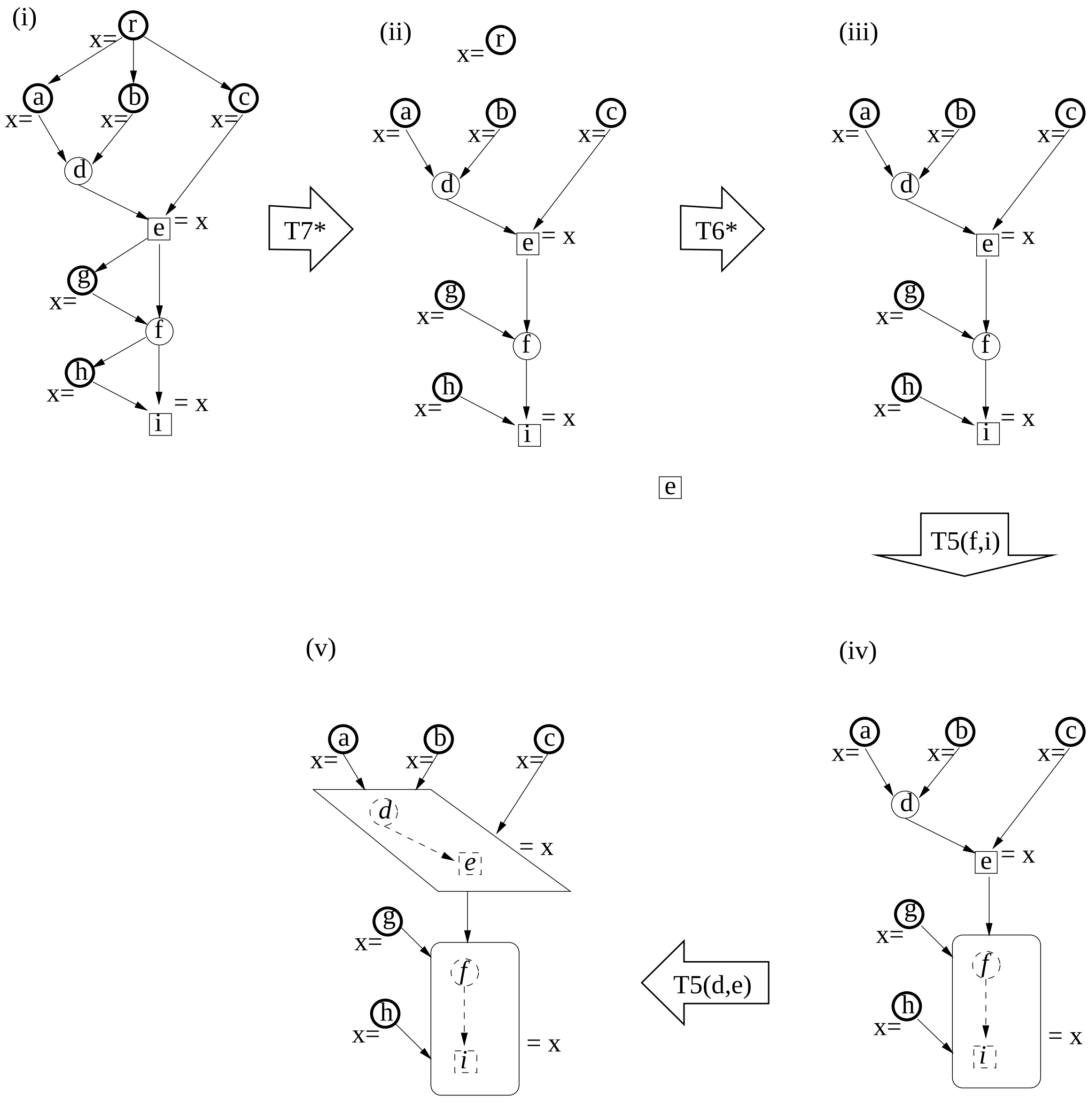

Figure 4: An example illustrating our algorithm for constructing a partially equivalent flow graph.

8 Interprocedural Extensions

We have discussed sparse evaluation as it applies to intraprocedural analysis (or analysis of single procedure programs). However, the ideas outlined in this paper can be easily extended to the case of interprocedural analysis. Assume that the input program consists of a set of procedures, each with its own control-flow graph. Some of the vertices in the graphs may correspond to "call"s to other procedures. We assume that, as part of the input, all the non-call vertices in each control-flow graph have been annotated as being a m-node or a p-node. Vertices representing procedure calls, however, are not annotated as part of the input.

Clearly, any procedure all of whose nodes are p-nodes can be "eliminated", and any call to this procedure may be marked as being a p-node. Iterative application of this idea, in conjunction with our algorithm for the intraprocedural case, suffices to construct the sparse evaluation representation of multi-procedure programs, in the absence of recursion.

Recursion complicates issues only slightly. Define a procedure P to be a p-procedure if all the nodes in procedure P and all the nodes in any procedure that may be directly or transitively called from P are p-nodes. Define P to be a m-procedure otherwise.

The set of all m-procedures in the program can be identified in a simple linear time traversal of the call graph. Initially mark all procedures containing a m-node as being a m-procedure. Then, traverse the call graph in reverse, identifying all procedures that may call a m-procedure, and marking them as being m-procedures as well.

Once this is done, we may mark a call node as being a m-node if it is a call to a m-procedure and as a p-node otherwise. Then, we can construct the sparse evaluation representation of each procedure independently, using our intraprocedural algorithm.

9 Related Work

The precursor to sparse evaluation forms was the Static Single Assignment form [5, 6], which was used to solve various analysis problems, such as constant propagation and redundancy elimination, more efficiently. Choi *et al.* [2] generalized the idea and defined the Sparse Evaluation Graph. Cytron and Ferrante [4], Sreedhar and Gao [16], and Pingali and Bilardi [14, 15] improve upon the efficiency of the original Choi *et al.* algorithm for constructing the Sparse Evaluation Graph. (We will discuss the relative efficiencies of the various algorithms in detail soon.) Johnson *et al.* [11, 10] define a different equivalent flow graph called the *Quick Propagation Graph* (QPG) and present a linear time algorithm for constructing it. Duesterwald *et al.* [8] show how a congruence partitioning technique can be used to construct an equivalent flow graph.

We now briefly compare our work with these different algorithms and representations in terms of the following three attributes.

Simplicity

Our work was originally motivated by a desire for a simpler algorithm for constructing Sparse Evaluation Graphs, one that did not require the dominator tree, which had been a standard prerequisite for most previous algorithms for constructing Sparse Evaluation Graphs. The Johnson *et al.* algorithm [10] does not require a dominator tree, but it has its own prerequisites, namely the identification of single-entry single-exit regions and the construction of a *Program Structure Tree.* Subsequent to our work, we became aware of an $O(n \log n)$ algorithm by Duesterwald *et al.* [8] for generating sparse evaluation forms. This algorithm is based on congruence partitioning and does not require the dominator tree either.

We believe that our algorithm is simpler to understand and implement than these previous algorithms for constructing sparse representations. (Of course, the dominator tree and the Program Structure Tree do have other applications, and if they are being built any way, then our algorithm does not offer any particular advantage in terms of implementation simplicity.)

Compactness

We have shown that the Compact Evaluation Graph is, in general, smaller than the Sparse Evaluation Graph and the Quick Propagation Graph. Consequently, dataflow analysis techniques can benefit even more by using this smaller representation.

We have also presented a quadratic algorithm for constructing the equivalent flow graph with the smallest number of vertices possible. This may be of interest for complicated and expensive analyses, such as pointer analysis, where it may be worth spending the extra time to reduce the number of vertices in the graph.

Duesterwald *et al.* present an $O(|V| \log |V|)$ algorithm for constructing an equivalent flow graph, which we believe is exactly the Sparse Evaluation Graph. They also describe another $O(|V| \log |V|)$ algorithm that can lead to further reductions in the size of graph. They then suggest iteratively applying both these algorithms until the graph can be reduced no more, leading to an $O(|V|^2 \log |V|)$ algorithm. Our algorithm for constructing the equivalent flow graph with the minimal number of vertices is similar in spirit to this $O(|V|^2 \log |V|)$ algorithm, but constructs an even smaller graph more efficiently.

Efficiency

Comparing the efficiency of the various algorithms for constructing the SSA form and the different equivalent flow graphs can be somewhat tricky. In particular, under some situations, the worst-case complexity measure does not tell us the full story.

If we are interested in the problem of constructing a single equivalent flow graph from a given graph, then comparing these algorithms is easy. The linear algorithms due to Pingali and Bilardi, Sreedhar and Gao, Johnson and Pingali, as well as our own linear time algorithm are all asymptotically optimal. One could argue that our algorithm has a smaller constant factor because of its simplicity.

Often, however, we may be interested in constructing multiple equivalent flow graphs from a given control-flow graph (each with respect to a different set of m-nodes). Our previous observations remain more or less valid even in this case. For each equivalent graph desired, one has to spend at least $\Omega(|V|)$ time building the map from the vertices of the original graph to the vertices of the equivalent flow graph. All the linear time algorithms should perform comparably (upto constant factors) for typical control-flow graphs, where $|E|$ is $O(|V|)$.

Now, assume that we are interested in constructing multiple *partially* equivalent flow graphs from a given control-flow graph. The problem of constructing the SSA form falls into this category — the true generalization of the SSA form appears to be the *partially* equivalent flow graph, not the equivalent flow graph. In particular, every subproblem instance specifies a set S of m-nodes as well as a set R of nodes where the dataflow solution is required. For each such subproblem, we need to construct a partially equivalent flow graph, and a mapping from every vertex in R to a vertex in the equivalent flow graph. Our algorithm, as well as Sreedhar and Gao's algorithm, will spend $\Omega(|V| + |E|)$ time for the construction of each partially equivalent flow graph. The original SSA algorithm [5, 6], in contrast, constructs all the partially equivalent flow graphs in parallel, sharing the linear time graph traversal overhead. For control flow graphs that arise in practice, this algorithm usually constructs each partially equivalent flow graph in "sub linear" time, even though, in the worst case, this algorithm can take quadratic time to construct each partially equivalent flow graph. Hence, many believe that, in practice, this algorithm will be faster than the algorithms that always take linear time for every partially equivalent flow graph. (See [17] for empirical evidence supporting this.) Fortunately, the work of Pingali and Bilardi [14, 15] shows how the original SSA algorithm can be adapted so that we have the best of both worlds, namely a linear worst-case complexity as well a "sub linear" behavior for graphs that arise in practice.

When is this finer distinction between the different linear time algorithms likely to be significant? One could argue that this difference is unlikely to be very significant for complex analysis problems, where the cost of the analysis is likely to dominate the cost of constructing the equivalent flow graph. Problems such as the reaching definitions problem, however, are simple and have linear time solutions. In this case, the cost of constructing the equivalent flow graph may be a significant fraction of the analysis time, and the above distinction could be significant. On the other hand, some [7] argue that constructing equivalent flow graphs is not the fastest way to solve such simple analysis problems anyway.

10 Conclusion

Previous work has shown equivalent flow graphs to be a useful representation, both for improving the performance of dataflow analysis algorithms as well as for representing dataflow information compactly.

This paper presents a linear time algorithm for computing an equivalent flow graph that is smaller than previously proposed equivalent flow graphs. We have presented a quadratic algorithm for constructing a equivalent flow graph consisting of the minimum number of vertices. We have also shown that the problem of constructing a equivalent flow graph consisting of the minimum number of vertices and edges is NP-hard.

We have shown how the concept of an equivalent flow graph can be generalized to that of a partially equivalent flow graph and have extended our algorithm to generate this more compact representation. For simple partitioned problems, such as the reaching definitions problem, the partially equivalent flow graph directly yields the desired solution, in "factored form".

The results presented here give rise to several interesting questions which appear worth pursuing. How significant is the NP-hardness result in practice? Can minimum size equivalent flow graphs be constructed efficiently for special classes of graphs, such as those that can be generated by structured programming constructs? Are there other graph transformations worth incorporating into our framework?

References

[1] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[2] 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.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] Ron Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. In *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 461–476. Springer-Verlag, 1993.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[6] 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.

[7] Dhananjay M. Dhamdere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, 1992.

[8] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 357–373. Springer-Verlag, 1994.

[9] S.L. Graham and M. Wegman. A fast and usually linear algorithm for global dataflow analysis algorithm. *J. ACM*, 23(1):172–202, 1976.

[10] R. Johnson, D. Pearson, and K. Pingali. The program tree structure: Computing control regions in linear time. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[11] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[12] G. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM.

[13] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[14] Keshav Pingali and Gianfranco Bilardi. Apt: A data structure for optimal control dependence computation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 32–46, 1995.

[15] 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.

[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. *Efficient Program Analysis Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.

[18] J.D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2:191–213, 1973.

[19] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report Computer Science Technical Report Number 840, University of Wisconsin, Madison, April 1989.