Demand Interprocedural Dataflow Analysis

Susan Horwitz
Thomas Reps
Mooly Sagiv

Technical Report #1283

August 1996
(REVISED)

Demand Interprocedural Dataflow Analysis

Susan Horwitz, Thomas Reps, and Mooly Sagiv[†]
University of Wisconsin

Abstract

An exhaustive dataflow-analysis algorithm associates with each point in a program a set of "dataflow facts" that are guaranteed to hold whenever that point is reached during program execution. By contrast, a *demand* dataflow-analysis algorithm determines whether a single given dataflow fact holds at a single given point.

This paper presents a new demand algorithm for interprocedural dataflow analysis. The algorithm has four important properties:

- It provides precise (meet-over-all-interprocedurally-valid-paths) solutions to a large class of problems.
- It has a polynomial worst-case cost for both a single demand and a sequence of all possible demands.
- The worst-case total cost of the sequence of all possible demands is no worse than the worst-case cost of a single run of the best known exhaustive algorithm for the same class of problems.
- Experimental results show that in many situations (*e.g.*, when only a small number of demands are made, or when most demands are answered *yes*) the demand algorithm is faster than the current best exhaustive algorithm.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.3.4 [**Programming Languages**]: Processors – *compilers, optimization*; E.1 [**Data Structures**] – *graphs*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory – *graph algorithms*

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: demand dataflow analysis, distributive dataflow framework, graph reachability, interprocedural dataflow analysis, interprocedurally realizable path, interprocedurally valid path, meet-over-all-valid-paths solution

---

[†]Currently at the University of Chicago.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937), by the Air Force Office of Scientific Research under grant AFOSR-91-0308, and by a grant from Xerox Corporate Research.

Part of this work was done while the authors were visiting the University of Copenhagen.

A preliminary version of this paper appeared in SIGSOFT 95: Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering (Washington DC, October 10-13, 1995) [Hor95]

Authors' addresses: Computer Sciences Department, Univ. of Wisconsin, 1210 West Dayton Street, Madison, WI 53706, USA; Computer Science Department, University of Chicago, Ryerson Hall, 1100 E. 58th St., Chicago, IL, 60637 USA.
Electronic mail: {horwitz, reps}@cs.wisc.edu; sagiv@cs.uchicago.edu

1. Introduction

An exhaustive dataflow-analysis algorithm associates with each point in a program a set of "dataflow facts" that are guaranteed to hold whenever that point is reached during program execution. This information can be used in a variety of software engineering tools (for example, to provide feedback to the programmer about possible errors such as the use of an uninitialized variable, or to determine whether a restructuring transformation is meaning-preserving) or can be used by an optimizing compiler (in choosing valid optimizing transformations).

It is not always necessary to compute complete dataflow information at all program points. A *demand* dataflow-analysis algorithm determines whether a given dataflow fact holds at a given point [Bab78,Due93,Rep94c,Rep94a,Rep94b,Due95]. Demand analysis can sometimes be preferable to exhaustive analysis for the following reasons:

- ***Narrowing the focus to specific points of interest.*** Software-engineering tools that use dataflow analysis often require information only at a certain set of program points. Similarly, in program optimization, most of the gains are obtained from making improvements at a program's "hot spots"—in particular, its innermost loops. However, current tools typically include a phase during which an exhaustive interprocedural dataflow-analysis algorithm is used. There is good reason to believe that the use of a demand algorithm will greatly reduce the amount of extraneous information computed.
- ***Narrowing the focus to specific dataflow facts of interest.*** Even when dataflow information is desired for every program point p, the full set of dataflow facts at p may not be required. For example, it is probably only useful to determine whether the variables *used* at p might be uninitialized, rather than determining that information for all of the variables in the procedure.
- ***Reducing work in preliminary phases.*** In problems that can be decomposed into separate phases, not all of the information from one phase may be required by subsequent phases. For example, the MayMod problem determines, for each call site, which variables may be modified during the call [Ban79,Coo88]. This problem can be decomposed into two phases: computing side effects disregarding aliases (the so-called DMod problem), and computing alias information [Ban79,Coo89,Coo88]. Given a demand (*e.g.*, "What is the MayMod set for a given call site c?"), a demand algorithm has the potential to reduce drastically the amount of work spent in earlier phases by propagating only relevant demands (*e.g.*, "What are the alias pairs (x, y) such that x is in DMod(c)"?).
- ***Demand analysis as a user-level operation.*** It is desirable to have program-development tools in which the user can ask questions interactively about various aspects of a program [Mas80,Wei84,Lin84,Hor86]. Such tools are particularly useful when debugging, when trying to understand complicated code, or when trying to transform a program to execute efficiently on a parallel machine. Because it is unlikely that a programmer will ask questions about all program points, solving just the user's sequence of demands is likely to be significantly less costly than an exhaustive analysis.

Of course, determining whether a given fact holds at a given point may require determining whether other, related facts hold at other points (and those other facts may not be "facts of interest" in the sense of the second bullet-point above). It is desirable, however, for a demand dataflow-analysis algorithm to minimize the amount of such auxiliary information computed. Certainly the worst-case cost of a demand dataflow-analysis algorithm (for one demand) should be no worse than the worst-case cost of the best exhaustive algorithm. Furthermore, it is desirable that the information computed in response to one demand be reusable, so as to minimize the cost of a *sequence* of demands; we call algorithms that are able to reuse information in this way ***caching demand algorithms***. Ideally, the worst-case total cost of the sequence of demands that produces complete dataflow information should be no worse than the worst-case cost of a single run of the best possible exhaustive algorithm; we call this the ***same-worst-case-cost property***. Since no non-trivial lower bounds (other than undecidability results) are currently known for dataflow analysis, it is not possible to determine whether a demand algorithm has the same-worst-case-cost property; however, it is possible to determine whether a demand algorithm has this property with respect to a particular exhaustive algorithm.

This paper presents a new caching demand algorithm for interprocedural dataflow analysis. The new algorithm, which is an improved version of the one reported in [Hor95], has four important properties:

- It provides precise (meet-over-all-interprocedurally-valid-paths) solutions to a large class of problems.
- It has a polynomial worst-case cost for both a single demand and a sequence of all possible demands.
- It has the same-worst-case-cost property with respect to the exhaustive algorithm given in [Rep95]. That algorithm is currently the best exhaustive algorithm for the class of dataflow problems that can be handled precisely by our demand algorithm: the *IFDS problems* defined in Section 2.1 (*i.e.*, of the exhaustive algorithms that can handle all IFDS problems, the one given in [Rep95] has the best asymptotic worst-case running time).
- Experimental results show that in many situations (*e.g.*, when only a small number of demands are made, or when most demands are answered *yes*) the demand algorithm is faster than the algorithm from [Rep95].

The remainder of the paper is organized as follows: Section 2 provides background material. First, the class of dataflow-analysis problems that can be handled by our algorithm is defined. Second, we show how to transform a dataflow-analysis problem in this class into a special kind of graph-reachability problem. Section 3 presents our new algorithm, which solves demands for dataflow-analysis information by solving equivalent graph-reachability demands. Experimental results on C programs are reported in Section 4. Section 5 discusses related work.

2. Background

2.1. The IFDS Dataflow Framework

The algorithm given in Section 3 can be used to solve any interprocedural dataflow-analysis problem in which the dataflow facts form a finite set D, and the dataflow functions (which are of type $2^D \rightarrow 2^D$) distribute over the meet operator (either union or intersection). We call this class of problems the *interprocedural, finite, distributive, subset problems*, or ***IFDS problems***, for short. The IFDS problems include all *locally separable* problems—the interprocedural versions of classical "bit-vector" or "gen-kill" problems (*e.g.*, reaching definitions, available expressions, and live variables)—as well as non-locally-separable problems such as truly-live variables [Gie81], copy-constant propagation [Fis88, pp. 660], and possibly-uninitialized variables. The IFDS framework was defined in [Rep95], where we presented an efficient exhaustive algorithm for solving IFDS problems. That definition is summarized below.

The IFDS framework is a variant of Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis [Sha81], with an extension similar to the one given by Knoop and Steffen in order to handle programs in which recursive procedures have local variables and parameters [Kno92]. These frameworks generalize Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [Kil73] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem.

In Kildall's framework, an instance of a dataflow-analysis problem consists of a bounded lower semilattice (the dataflow information) with meet operator $\sqcap$, a flowgraph (representing the program), and an assignment of dataflow functions to the edges of the flowgraph. If all of the dataflow functions are distributive, Kildall's algorithm computes the meet-over-all-paths solution to the problem instance. Similarly, in the IFDS framework, an instance of a dataflow-analysis problem (or IFDS problem, for short) consists of the following:

- A finite set D (the dataflow information).
- A meet operator $\sqcap$. The algorithm given in Section 3 requires that the meet operator be union. However, the algorithm can still be used to solve problems for which the meet operator is intersection: such problems can always be transformed to a complementary problem in which the meet operator is union, and the algorithm can then be applied.
- A ***supergraph*** G^* (a collection of flowgraphs, one for each procedure). In supergraph G^*, a procedure call is represented by two nodes, a ***call*** node and a ***return-site*** node. In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call—represented by call-node c and return-site node r—G^* has three edges: an intraprocedural ***call-to-return-site*** edge from c to r; an interprocedural ***call-to-start*** edge from c to the start node of the called procedure; an interprocedural ***exit-to-return-site*** edge from the exit node of the called procedure to r.

(The call-to-return-site edges are included so that the IFDS framework can handle programs with local variables and parameters; the dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables and value parameters that holds at the call site to be combined with the information about global variables and reference parameters that holds at the end of the called procedure.)

- An assignment of distributive dataflow functions (of type $2^D \to 2^D$) to the edges of the supergraph.

Given an instance of an IFDS problem, a dataflow fact $d \in D$, and a flowgraph node n, the demand algorithm given in Section 3 determines whether fact d is in the meet-over-all-valid-paths solution at node n. The distinction between meet-over-all-paths and meet-over-all-*valid*-paths is necessary to capture the idea that not all paths in G^* represent potential execution paths. A ***valid path*** is one that respects the fact that a procedure always returns to the site of the most recent call. To understand the algorithm of Section 3, it is useful to distinguish further between a ***same-level valid path*** (a path in G^* that starts and ends in the same procedure, and in which every call has a corresponding return) and a ***valid path*** (a path that may include one or more unmatched calls).

Example. Figure 1 shows an example program and its supergraph G^*. In G^*, the path

$$start_{main} \to n1 \to n2 \to start_p \to n4 \to exit_p \to n3$$

is a (same-level) valid path; the path

$$start_{main} \to n1 \to n2 \to start_p \to n4$$

is a non-same-level valid path (because the call-to-start edge $n2 \to start_p$ has no matching exit-to-return-site edge); the path

$$start_{main} \to n1 \to n2 \to start_p \to n4 \to exit_p \to n8$$

is not a valid path because the exit-to-return-site edge $exit_p \to n8$ does not correspond to the preceding call-to-start edge $n2 \to start_p$.

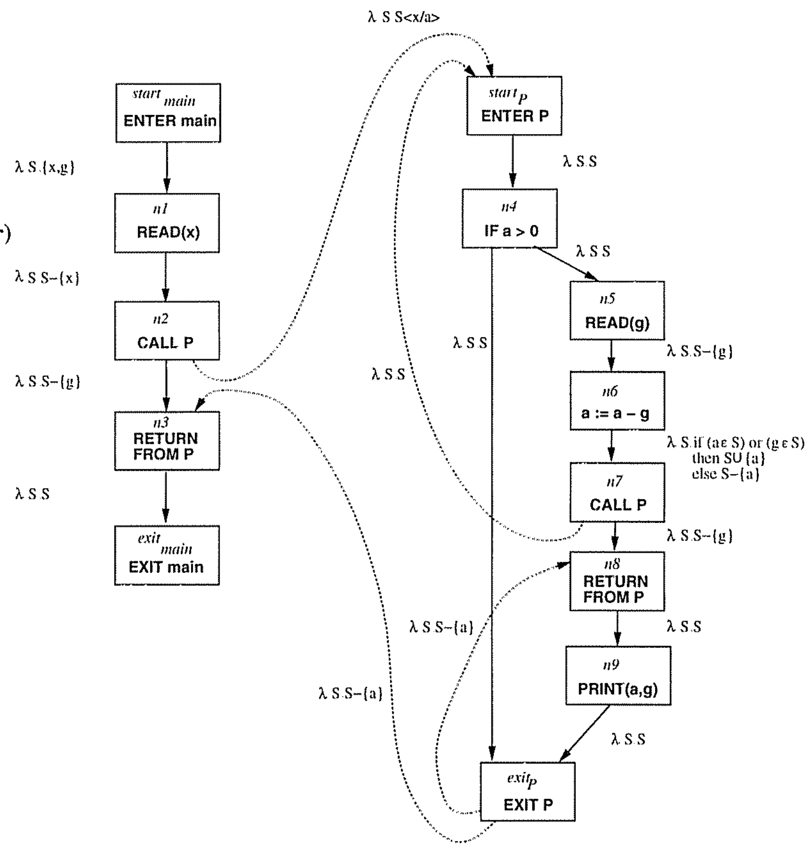
In Figure 1, the supergraph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The "possibly-uninitialized variables" problem is to determine, for each node n in G^*, the set of program variables that may be uninitialized just before execution reaches n. A variable x is possibly uninitialized at n either if there is an x-definition-free valid path from the start of the program to n, or if there is a valid path from the start of the program to n on which the last definition of x uses some variable y that itself is possibly uninitialized. For example, the dataflow function associated with edge $n6 \to n7$ shown in Figure 1 adds a to the set of possibly-uninitialized variables after node $n6$ if either a or g is in the set of possibly-uninitialized variables before node $n6$. □

The IFDS framework can be used for languages with a variety of features (including procedure calls, parameters, global and local variables, and pointers). Encoding a problem in the IFDS framework may in

```
declare g: integer

procedure main
begin
  declare x: integer
  read(x)
  call P(x)
end

procedure P(value a: integer)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```



(a) Example program (b) Its supergraph G^*

Figure 1. An example program and its supergraph G^*. The supergraph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The notation S<x/a> denotes the set S with x renamed to a.

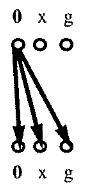
some cases involve a loss of precision; for example, in languages with pointers there may be a loss of precision for problem instances in which there is aliasing. Once a problem has been encoded in the IFDS framework, the demand algorithm presented in this paper provides (with no further loss of precision) an efficient way to determine whether a particular dataflow fact is in the meet-over-all-valid-paths solution to the problem.

2.2. From Dataflow-Analysis Problems to Realizable-Path Reachability Problems

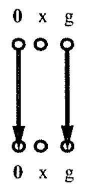
In this section, we show how to convert IFDS problems to "realizable-path" graph-reachability problems. This is done by transforming an instance of an IFDS problem (a supergraph G^* in which each edge has an associated distributive function in $2^D \to 2^D$) into an ***exploded supergraph*** $G^\#$, in which each node $\langle n, d \rangle$ represents dataflow fact $d \in D$ at supergraph node n, and each edge represents a dependence between dataflow facts at different supergraph nodes.

The key insight behind this "explosion" is that a distributive function f in $2^D \rightarrow 2^D$ can be represented using a graph with $2D+2$ nodes; this graph is called f's *representation relation*. Half of the nodes in this graph represent f's input; the other half represent its output. D of these nodes represent the "individual" dataflow facts that form set D, and the remaining node (which we call $\mathbf{0}$) essentially represents the empty set. An edge $\mathbf{0} \rightarrow d$ means that d is in $f(S)$ regardless of the value of S (in particular, d is in $f(\emptyset)$). An edge $d_1 \rightarrow d_2$ means that d_2 is not in $f(\emptyset)$, and is in $f(S)$ whenever d_1 is in S. Every graph includes the edge $\mathbf{0} \rightarrow \mathbf{0}$; this is so that functional composition corresponds to compositions of representation relations (this is explained below).

Example. The main procedure shown in Figure 1 has two variables, x and g. Therefore, the representation relations for the dataflow functions associated with this procedure will each have six nodes. The function associated with the edge from $start_{main}$ to *n1* is $\lambda S.\{x, g\}$; that is, variables x and g are added to the set of possibly-uninitialized variables regardless of the value of S. The representation relation for this function is:



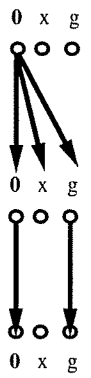
The representation relation for the function $\lambda S.S - \{x\}$ (which is associated with the edge from *n1* to *n2*) is shown below. Note that x is never in the output set, and g is there iff it is in S.



□

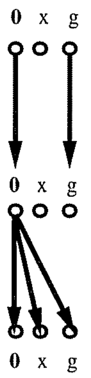
A function's representation relation correctly captures the function's semantics in the sense that the representation relation can be used to evaluate the function. In particular, the result of applying function f to input S is the union of the values represented by the "output" nodes in f's representation relation that are the targets of edges from the "input" nodes that represent either $\mathbf{0}$ or a node in S. For example, consider applying the dataflow function $\lambda S.S - \{x\}$ to the set $\{x\}$ using the representation relation shown above. There is no edge out of the initial x node, and the only edge out of the initial $\mathbf{0}$ node is to the final $\mathbf{0}$ node, so the result of this application is $\emptyset$. The result of applying the same function to the set $\{x, g\}$ is $\{g\}$, because there is an edge from the initial g node to the final g node.

The composition of two functions is represented by "pasting together" the graphs that represent the individual functions. For example, the composition of the two functions discussed above: $\lambda S.S - \{x\} \circ \lambda S.\{x, g\}$, is represented as follows:



Paths in a "pasted-together" graph represent the result of applying the composed functions. For example, there is a path in the graph shown above from the initial **0** node to the final g node. This means that g is in the final set regardless of the value of S to which the composed functions are applied. There is *no* path from an initial node to the final x node; this means that x is not in the final set, regardless of the value of S.

To understand the need for the $\mathbf{0} \rightarrow \mathbf{0}$ edges in the representation relations, consider composing the two example functions in the opposite order: $\lambda S.\{x, g\} \circ \lambda S.S - \{x\}$. This function composition is represented as follows:



Note that both x and g are in the final set regardless of the value of S to which the composed functions are applied. This is reflected in the graph shown above by the paths from the initial **0** node to the final x and g nodes. However, if there were no edge from the initial **0** node to the intermediate **0** node, there would be no such paths, and the graph would not correctly represent the composed functions.

Returning to the definition of the exploded supergraph $G^{\#}$: Each node n in supergraph G^* is "exploded" into $D + 1$ nodes in $G^{\#}$, and each edge $m \rightarrow n$ in G^* is "exploded" into the representation relation of the function associated with $m \rightarrow n$. In particular:

(i) For every node n in G^*, there is a node $\langle n, \mathbf{0} \rangle$ in $G^\#$.

(ii) For every node n in G^*, and every dataflow fact $d \in D$, there is a node $\langle n, d \rangle$ in $G^\#$.

Given function f associated with edge $m \to n$ of G^*:

(iii) There is an edge in $G^\#$ from node $\langle m, \mathbf{0} \rangle$ to node $\langle n, d \rangle$ for every $d \in f(\emptyset)$.

(iv) There is an edge in $G^\#$ from node $\langle m, d_1 \rangle$ to node $\langle n, d_2 \rangle$ for every d_1, d_2 such that $d_2 \in f(\{ d_1 \})$ and $d_2 \notin f(\emptyset)$.

(v) There is an edge in $G^\#$ from node $\langle m, \mathbf{0} \rangle$ to node $\langle n, \mathbf{0} \rangle$.

Because "pasted together" representation relations correspond to function composition, a path in the exploded supergraph from node $\langle n, d_1 \rangle$ to node $\langle m, d_2 \rangle$ means that if dataflow fact d_1 holds at supergraph node n, then dataflow fact d_2 will hold at node m. By looking at paths that start from node $\langle start_{main}, \mathbf{0} \rangle$ (which represents the fact that no dataflow facts hold at the start of procedure *main*) we can determine which dataflow facts hold at each node. However, recall that we are not interested in *all* paths in the exploded supergraph; we are only interested in those that correspond to *valid* paths in the supergraph G^*. We call those paths in $G^\#$ its ***realizable paths***; similarly, we call a path in $G^\#$ that corresponds to a *same-level valid path* in G^* a ***same-level realizable path***. [Rep94a] includes a proof that dataflow fact d holds at supergraph node n iff there is a realizable path in $G^\#$ from node $\langle start_{main}, \mathbf{0} \rangle$ to node $\langle n, d \rangle$.

Example. The exploded supergraph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Figure 1 is shown in Figure 2. The dataflow functions are replaced by their representation relations. In Figure 2, closed circles represent nodes that are reachable along realizable paths from $\langle start_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along realizable paths. (For example, note that nodes $\langle n8, g \rangle$ and $\langle n9, g \rangle$ are reachable only along non-realizable paths from $\langle start_{main}, \mathbf{0} \rangle$.) As stated above, this information indicates the nodes' values in the meet-over-all-valid-paths solution to the dataflow-analysis problem. For instance, the meet-over-all-valid-paths solution at node $exit_p$ is the set $\{g\}$. (That is, variable g is the only possibly-uninitialized variable just before execution reaches the exit node of procedure p.) In Figure 2, this information can be obtained by determining that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle exit_p, g \rangle$, but not from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle exit_p, a \rangle$. □

3. A Demand Algorithm for IFDS Problems

In this section, we show how to solve demand IFDS problems by solving equivalent realizable-path reachability demands. The algorithm, called the ***Demand-Tabulation Algorithm***, is presented in Figure 3. The top-level function of the algorithm is called IsMemberOfSolution. The call IsMemberOfSolution($\langle \bar{n}, \bar{d} \rangle$) returns *true* iff there is a realizable path from node $\langle start_{main}, \mathbf{0} \rangle$ to node $\langle \bar{n}, \bar{d} \rangle$ in $G^\#$. Such a path exists iff the meet-over-all-valid-paths solution to the dataflow-analysis problem at node $\bar{n}$ of G^* includes dataflow fact $\bar{d}$.

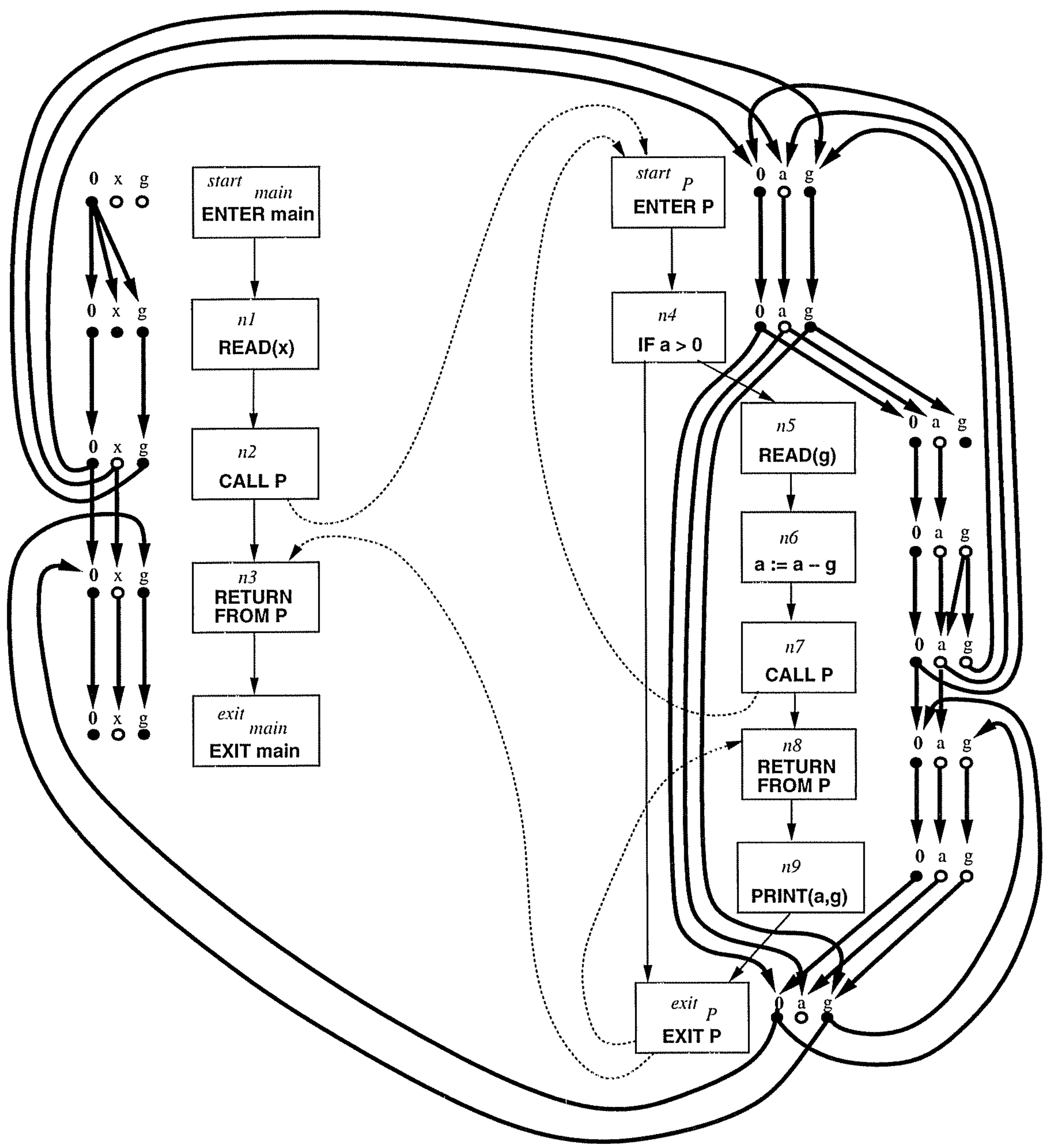


Figure 2. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}$ that are reachable along realizable paths from $\langle start_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along such paths.

```
     declare
       $G^\# = (N^\#, E^\#)$:              global exploded supergraph
       PathEdge, SummaryEdge:         global edge set, initially empty            /* These sets are preserved across calls */
       ReachableNodes:                global node set, initially $\{\langle n, \mathbf{0}\rangle \mid n \in N^*\}$   /* This set is preserved across calls */
       VisitedNodes:                  global node set, initially empty            /* This set is preserved across calls */
     function IsMemberOfSolution($\langle \bar{n}, \bar{d}\rangle$: exploded supergraph node) returns boolean
     declare $en$: exploded supergraph node or $Failure$
     begin
[1]  $en$ = BackwardDFS($\langle \bar{n}, \bar{d}\rangle$)
[2]  if $en = Failure$ then
[3]     return($false$)
[4]  else
[5]     UpdateReachableNodes($en$)
[6]     return($true$)
[7]  fi
     end

     function BackwardDFS($\langle \bar{n}, \bar{d}\rangle$) returns exploded supergraph node or $Failure$
     declare EdgeWorkList: edge set;  NodeStack: node stack
     begin
[8]    push $\langle \bar{n}, \bar{d}\rangle$ onto NodeStack
[9]    while NodeStack is not empty do
[10]     pop a node $\langle n, d\rangle$ from NodeStack
[11]     if $\langle n, d\rangle \in$ ReachableNodes then
[12]       return($\langle n, d\rangle$)
[13]     else if $\langle n, d\rangle \notin$ VisitedNodes then
[14]       insert $\langle n, d\rangle$ into VisitedNodes
[15]       switch $n$
[16]         case $n$ is a return-site node :
[17]           let $c$ be the call node that corresponds to $n$, and let $p$ be the procedure called at $c$
[18]             EdgeWorkList := $\emptyset$
[19]             for each $d'$ such that $\langle exit_p, d'\rangle \to \langle n, d\rangle \in E^\#$ do Propagate($\langle exit_p, d'\rangle \to \langle exit_p, d'\rangle$, EdgeWorkList) od
[20]             BackwardTabulateSLRPs(EdgeWorkList)
[21]             for each $d'$ such that $\langle c, d'\rangle \to \langle n, d\rangle \in (E^\# \cup$SummaryEdge) and $\langle c, d'\rangle \notin$ VisitedNodes do push $\langle c, d'\rangle$ onto NodeStack od
[22]           end let
[23]         end case
[24]         case $n$ is the start node of procedure $p$ :
[25]           for each $c \in$ Callers($p$) do
[26]             for each $d'$ such that $\langle c, d'\rangle \to \langle n, d\rangle \in E^\#$ and $\langle c, d'\rangle \notin$ VisitedNodes do push $\langle c, d'\rangle$ onto NodeStack od
[27]           od
[28]         end case
[29]         default :
[30]           for each $\langle m, d'\rangle$ such that $\langle m, d'\rangle \to \langle n, d\rangle \in E^\#$ and $\langle m, d'\rangle \notin$ VisitedNodes do push $\langle m, d'\rangle$ onto NodeStack od
[31]         end case
[32]       end switch
[33]     fi
[34]   od
[35]   return($Failure$)
     end

     procedure UpdateReachableNodes($\langle n_0, d_0\rangle$: exploded supergraph node)
     declare NodeWorkList: node set = $\emptyset$
     declare $\langle n, d\rangle$, $\langle m, d'\rangle$: exploded supergraph node
     begin
[36]   insert $\langle n_0, d_0\rangle$ into NodeWorkList
[37]   while NodeWorkList $\neq \emptyset$ do
[38]     Select and remove an exploded supergraph node $\langle n, d\rangle$ from NodeWorkList
[39]     insert $\langle n, d\rangle$ into ReachableNodes
[40]     remove $\langle n, d\rangle$ from VisitedNodes
[41]     for each $\langle m, d'\rangle$ such that ($\langle n, d\rangle \to \langle m, d'\rangle \in (E^\# \cup$SummaryEdge)) and $\langle n, d\rangle \to \langle m, d'\rangle$ is not an exit-to-return-site edge, and
                                  ($\langle m, d'\rangle \in$ VisitedNodes) and ($\langle m, d'\rangle \notin$ ReachableNodes) do
[42]       Insert $\langle m, d'\rangle$ into NodeWorkList
[43]     od
[44]   od
     end
```

Figure 3. The Demand-Tabulation Algorithm determines whether dataflow fact $\bar{d}$ holds at flowgraph node $\bar{n}$. Procedures BackwardTabulateSLRPs and Propagate are given in Figure 4.

IsMemberOfSolution consists of a backward phase (performed by function BackwardDFS) followed by a forward phase (performed by procedure UpdateReachableNodes). BackwardDFS performs a backward depth-first search of $G^{\#}$ starting from "demand" node $\langle \overline{n}, \overline{d} \rangle$, to determine whether the demand node can be reached via a realizable path from node $\langle start_{main}, \mathbf{0} \rangle$. UpdateReachableNodes is called only when BackwardDFS is successful. The purpose of UpdateReachableNodes is to update two sets, ReachableNodes and VisitedNodes, that are maintained across calls to IsMemberOfSolution in order to prevent repeating work done on a previous call.

The ReachableNodes and VisitedNodes sets are used and maintained as follows:

ReachableNodes

An exploded-graph node $\langle n, d \rangle$ is placed in set ReachableNodes when it has been determined that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle n, d \rangle$. Before the first call on IsMemberOfSolution is performed, ReachableNodes is initialized to $\{ \langle n, \mathbf{0} \rangle \}$ for all supergraph nodes n.

As soon as function BackwardDFS encounters a node $\langle n, d \rangle$ that is in ReachableNodes, the backward depth-first search is terminated, and node $\langle n, d \rangle$ is returned (line 12). (The fact that $\langle n, d \rangle$ is reachable via a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ together with the fact that BackwardDFS only visits nodes from which there is a realizable path to the "demand" node $\langle \overline{n}, \overline{d} \rangle$, means that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle \overline{n}, \overline{d} \rangle$.)

Procedure UpdateReachableNodes starts at the node $\langle n, d \rangle$ returned by BackwardDFS, and performs a forward traversal of $G^{\#}$, following only the edges that were traversed backwards by BackwardDFS. All of the nodes that it encounters are reachable from $\langle start_{main}, \mathbf{0} \rangle$ via a realizable path, so they are added to ReachableNodes. (This update of ReachableNodes is part of what makes the Demand-Tabulation Algorithm a caching algorithm. The algorithm would still be correct if UpdateReachableNodes followed all edges other than exit-to-return-site edges, but it would increase the time required for a single demand.)

VisitedNodes

Between invocations of IsMemberOfSolution, the exploded-graph nodes in set VisitedNodes are those for which it has been determined that there is *no* realizable path from $\langle start_{main}, \mathbf{0} \rangle$. During an invocation of IsMemberOfSolution, nodes visited for the first time are added to this set (line 14); those determined to be reachable from $\langle start_{main}, \mathbf{0} \rangle$ by a realizable path are transferred from this set to the ReachableNodes set by procedure UpdateReachableNodes (lines 39 and 40). Note that when BackwardDFS returns *Failure*, none of the nodes that have been added to VisitedNodes by BackwardDFS are reachable from $\langle start_{main}, \mathbf{0} \rangle$, so it is not necessary for IsMemberOfSolution to call UpdateReachableNodes.

An interesting aspect of BackwardDFS is how it ensures that only nodes from which there is a realizable path to demand node $\langle \overline{n}, \overline{d} \rangle$ are visited. This is accomplished by the call to BackwardTabulateSLRPs at line 20, which occurs when the node $\langle n, d \rangle$ popped from the stack corresponds to a return-site node (*i.e.*, n is a return-site node in G^*). The purpose of BackwardTabulateSLRPs is to find ***summary edges***, which represent transitive dependences due to procedure calls: A summary edge of the form $\langle c, d_1 \rangle \rightarrow \langle r, d_2 \rangle$ (where c is a call node and r is the matching return-site node) represents a same-level realizable path from $\langle c, d_1 \rangle$ to $\langle r, d_2 \rangle$. Summary edges are recorded in the (global) set named SummaryEdge. After calling BackwardTabulateSLRPs, BackwardDFS can continue its backward traversal across the newly discovered summary edges (line 21).

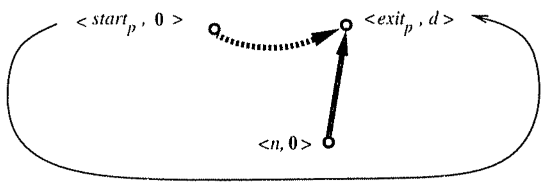
```
     declare $G^\# = (N^\#, E^\#)$:          global exploded supergraph
     declare PathEdge, SummaryEdge: global edge set, initially empty     /* These sets are preserved across calls */
     procedure BackwardTabulateSLRPs(EdgeWorkList: edge set)
     begin
[45]   while EdgeWorkList $\neq \emptyset$ do
[46]     Select and remove an edge $\langle n, d_1 \rangle \rightarrow \langle exit_p, d \rangle$ from EdgeWorkList
[47]     if ($\langle n, d_1 \rangle$ is $\langle start_p, \mathbf{0} \rangle$) or ($\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle \notin$ PathEdge) then
[48]       switch $n$
[49]         case $n$ a return-site node :
[50]           let $c$ be the call node that corresponds to $n$, and $q$ be the procedure called at $c$
[51]             for each $d_2$ such that $\langle exit_q, d_2 \rangle \rightarrow \langle n, d_1 \rangle \in E^\#$ do Propagate($\langle exit_q, d_2 \rangle \rightarrow \langle exit_q, d_2 \rangle$, EdgeWorkList) od
[52]             for each $d_2$ such that $\langle c, d_2 \rangle \rightarrow \langle n, d_1 \rangle \in (E^\# \cup$SummaryEdge) do Propagate($\langle c, d_2 \rangle \rightarrow \langle exit_p, d \rangle$, EdgeWorkList) od
[53]           end let
[54]         end case
[55]         case $n$ the start node of procedure $p$ :
[56]           for each $c \in$ Callers($p$) do
[57]             let $q$ be $c$'s procedure, and $r$ be the return-site node that corresponds to $c$
[58]               for each $d_3$, $d_4$ such that $\langle c, d_4 \rangle \rightarrow \langle n, d_1 \rangle \in E^\#$ and $\langle exit_p, d \rangle \rightarrow \langle r, d_3 \rangle \in E^\#$ do
[59]                 if $\langle c, d_4 \rangle \rightarrow \langle r, d_3 \rangle \notin$ SummaryEdge then
[60]                   Insert $\langle c, d_4 \rangle \rightarrow \langle r, d_3 \rangle$ into SummaryEdge
[61]                   for each $d_2$ such that $\langle r, d_3 \rangle \rightarrow \langle exit_q, d_2 \rangle \in$ PathEdge do Propagate($\langle c, d_4 \rangle \rightarrow \langle exit_q, d_2 \rangle$, EdgeWorkList) od
[62]                 fi
[63]               od
[64]             end let
[65]           od
[66]         end case
[67]         default :
[68]           for each $m$, $d_2$ such that $\langle m, d_2 \rangle \rightarrow \langle n, d_1 \rangle \in E^\#$ do Propagate($\langle m, d_2 \rangle \rightarrow \langle exit_p, d \rangle$, EdgeWorkList) od
[69]         end case
[70]       end switch
[71]     fi
[72]   od
     end
     procedure Propagate($\langle n, d_1 \rangle \rightarrow \langle exit_p, d \rangle$: edge, EdgeWorkList: edge set)
     begin
[73]   if $d_1$ is $\mathbf{0}$ then
[74]     $n := start_p$
[75]   fi
[76]   if $\langle n, d_1 \rangle \rightarrow \langle exit_p, d \rangle \notin$ PathEdge then
[77]     Insert $\langle n, d_1 \rangle \rightarrow \langle exit_p, d \rangle$ into PathEdge
[78]     Insert $\langle n, d_1 \rangle \rightarrow \langle exit_p, d \rangle$ into EdgeWorkList
[79]   fi
     end
```

Figure 4. Procedure BackwardTabulateSLRPs finds summary edges and records them in set SummaryEdge.

BackwardDFS calls three auxiliary subprograms: Callers, Propagate, and BackwardTabulateSLRPs. Function Callers(p) returns the set of call nodes that represent calls on p; procedures Propagate and BackwardTabulateSLRPs are shown in Figure 4. As discussed above, the purpose of BackwardTabulateSLRPs is to find summary edges, and to record them in the set named SummaryEdge. In order to do this, BackwardTabulateSLRPs finds ***path edges*** (which represent same-level realizable paths in $G^{\#}$) whose targets are nodes of the form $\langle exit_p, d \rangle$ (*i.e.*, nodes of $G^{\#}$ that correspond to exit nodes of G^*). It records all such path edges in the (global) set named PathEdge.

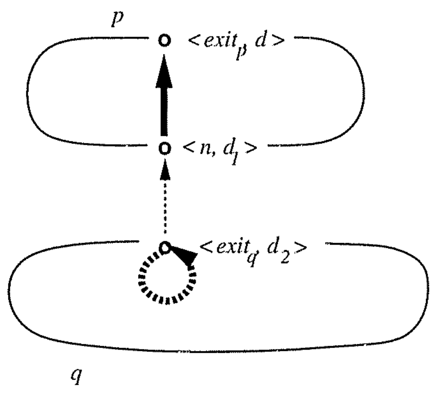
Procedure BackwardTabulateSLRPs is a worklist algorithm that starts with an initial worklist containing a set of zero-length path edges (edges of the form $\langle exit_p, d \rangle \rightarrow \langle exit_p, d \rangle$); on each iteration of the main loop it deduces the existence of additional path edges and summary edges.

In terms of leading quickly to a "yes" answer to a demand, the best thing that can happen in BackwardTabulateSLRPs is to discover a path edge whose source is a "**0**" node (*i.e.*, a path edge of the form $\langle n, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$). In this case, the answer to the current demand is guaranteed to be "yes." However, BackwardTabulateSLRPs cannot simply quit, because it is vital that the PathEdge and SummaryEdge sets be left in a consistent state to ensure that subsequent calls to IsMemberOfSolution return the correct answer. In particular, BackwardTabulateSLRPs must finish finding all path edges whose targets are some node *other* than $\langle exit_p, d \rangle$. On the other hand, there is no need to process any more path edges to $\langle exit_p, d \rangle$. Therefore, on discovering such an edge, BackwardTabulateSLRPs inserts the path edge $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$ into PathEdge and into the worklist (lines 74, 77 and 78). This situation is illustrated below. The solid bold arrow represents the path edge whose source is a "**0**" node, and the dotted bold arrow represents the new path edge that is inserted into PathEdge and the worklist.

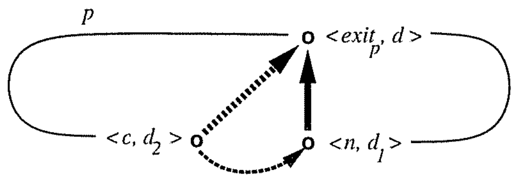


Furthermore, when a path edge is taken off the worklist (line 46) it is processed only if it is itself of the form $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$, or if that path edge has not yet been discovered.

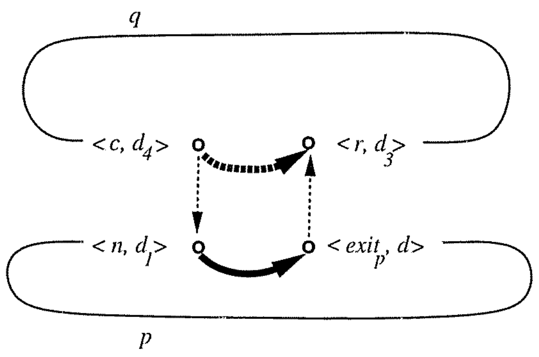
The configurations that are used by BackwardTabulateSLRPs to deduce the existence of path edges and summary edges are depicted in Figure 5. The first two diagrams of Figure 5 correspond to the case where n is a return-site node; the next two diagrams correspond to the case where n is a start node; and the final diagram corresponds to the default case. In Figure 5, the bold dotted arrows represent edges that are inserted into sets PathEdge and SummaryEdge if they were not previously in those sets.



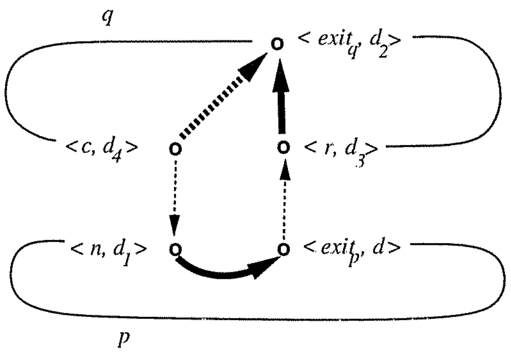
Line 51



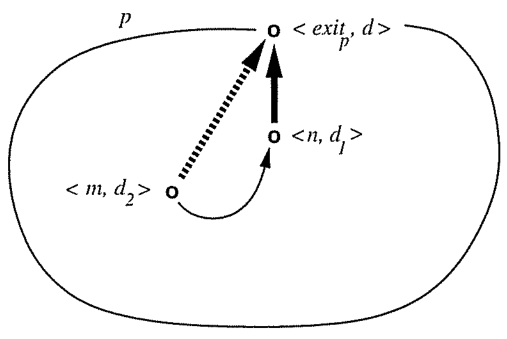
Line 52



Line 60



Line 61



Line 68

→ $E^{\#}$ edge corresponding to a control–flow–graph edge

⇢ $E^{\#}$ edge corresponding to a call–to–return–site edge or edge in SummaryEdge

⇢ $E^{\#}$ edge corresponding to a call–to–start or exit–to–return–site edge

➡ edge in PathEdge

➡ (possibly new) edge in PathEdge

➡ (possibly new) edge in SummaryEdge

Key

Figure 5. These five diagrams show how procedure BackwardTabulateSLRPs deduces the existence of new path and summary edges.

Example. When IsMemberOfSolution is called with the exploded supergraph node $\langle n9, g\rangle$ from the example shown in Figure 2, (*i.e.*, the demand "Might g be uninitialized at node $n9$?" is made), the following steps are performed (all line numbers refer to lines in Figure 3):

1. BackwardDFS is called with node $\langle n9, g\rangle$.
2. Node $\langle n9, g\rangle$ is pushed onto NodeStack at line 8, and then popped off (into $\langle n, d\rangle$) at line 10.
3. Node $\langle n9, g\rangle$ is inserted into VisitedNodes at line 14. The default case of the switch (line 29) is taken, and node $\langle n8, g\rangle$ is pushed onto NodeStack.
4. Node $\langle n8, g\rangle$ is popped from NodeStack; $n8$ is a return-site node, so the case on line 16 is selected, and BackwardTabulateSLRPs is called for the first time. This causes summary edge $\langle n7, g\rangle \rightarrow \langle n8, g\rangle$ to be inserted into SummaryEdge (and causes several other edges to be inserted into PathEdge).
5. Node $\langle n7, g\rangle$ is pushed onto NodeStack at line 21.
6. Node $\langle n7, g\rangle$ is popped from NodeStack; the default case is taken, and node $\langle n6, g\rangle$ is pushed onto NodeStack.
7. Node $\langle n6, g\rangle$ is popped from NodeStack; the default case is taken, but there are no edges that satisfy the for-loop condition (line 30).
8. NodeStack is now empty, so BackwardDFS returns *Failure*, and IsMemberOfSolution returns *false*. □

3.1. Cost Of The Demand-Tabulation Algorithm

In this section we discuss the time and space requirements of the Demand-Tabulation Algorithm. To express these costs in terms of the size of the (unexploded) supergraph, we will use the following parameters:

N	the number of nodes in supergraph G^*
E	the number of edges in supergraph G^*
Call	the number of call nodes in supergraph G^*
D	the size of set D

The maximum number of exploded supergraph and summary edges (and thus, the worst-case time and space requirements of the Demand-Tabulation Algorithm) varies depending on what class of dataflow-analysis problems is being solved. There are two interesting sub-classes of the distributive dataflow-analysis problems: the *h-sparse* problems and the *locally separable* problems.

Definition 3.1. A problem is ***h-sparse*** if all problem instances have the following property: For each function f on an ordinary intraprocedural edge or a call-to-return-site edge of G^*, the number of edges in $G^\#$ that represent function f, excluding edges that emanate from the **0** node, is at most hD. □

In general, when the nodes of G^* represent individual statements and predicates (rather than basic blocks), and when there is no aliasing, we expect most distributive problems to be h-sparse (with $h \ll D$): Each statement changes only a small portion of the execution state, and accesses only a small portion of the state as well. Therefore, the dataflow functions, which are abstractions of the statements' semantics, should be "close to" the identity function. The identity function is represented using $D + 1$ edges; thus, the number of edges needed to represent each dataflow function should be roughly D.

Example. When the nodes of G^* represent individual statements and predicates, and there is no aliasing, every instance of the possibly-uninitialized variables problem is 2-sparse. The only non-identity dataflow functions are those associated with assignment statements. The outdegree of every non-**0** node in the representation of such a function is at most two: a variable's initialization status can affect itself and at most one other variable, namely the variable assigned to. □

Definition 3.2. A problem is ***locally separable*** if all problem instances have both of the following properties:

- Intraprocedural dataflow functions have only component-wise dependences: For each function f on an ordinary intraprocedural edge or a call-to-return-site edge of G^*, for each dataflow fact d, either d is not in $f(S)$ for any S, or d is in $f(S)$ for all S, or d is in $f(S)$ iff d is in S. In other words, while there is no restriction on the number of out-going edges from the initial **0** node of a function's representation relation, every other initial node d must either have no out-going edges, or a single out-going edge to final node d.
- Corresponding calls and returns have related dataflow functions: If the representation relation for the dataflow function associated with a call-to-start edge $c \rightarrow s$ includes the edge $\langle c, d_1 \rangle \rightarrow \langle s, d_2 \rangle$, then the representation relation for the dataflow function associated with the corresponding exit-to-return-site edge $e \rightarrow r$ either includes the edge $\langle e, d_2 \rangle \rightarrow \langle r, d_1 \rangle$, or exploded node $\langle e, d_2 \rangle$ has no outgoing edge. □

The locally separable problems are the interprocedural versions of the classical separable problems from intraprocedural dataflow analysis (also known as gen/kill or bit-vector problems). All locally separable problems are 1-sparse, but not vice versa.

Another parameter that affects the running time of the Demand-Tabulation Algorithm is the "bandwidth" for the transmission of dataflow information between procedures. In particular, the times given here rely on the fact that it is always possible to construct $G^{\#}$ so the maximum outdegree of a non-**0** node in a call-to-start edge's representation relation, and the maximum indegree of a non-**0** node in an exit-to-return-site edge's representation relation are both 1. (See the Appendix of [Rep95] for a more complete discussion of this issue.) The implementation reported in Section 4 constructs $G^{\#}$ so that these properties hold.

The table in Figure 6 summarizes the worst-case size of the exploded supergraph $G^{\#}$ (in terms of the number of exploded edges) as well as the worst-case number of summary edges that might be added by the Demand-Tabulation Algorithm for distributive, h-sparse, and locally-separable dataflow-analysis problems (the number of summary edges added in the worst-case is the same for a single demand and for a sequence of all possible demands). In practice, we have found that the actual numbers are much smaller than those

Class of functions	Graph-theoretic characterization of the dataflow functions' properties	Number of edges in $G^{\#}$	Number of added summary edges
Distributive	Up to $O(D^2)$ edges/representation-relation	$O(ED^2)$	$O(CallD^2)$
h-sparse	At most $O(hD)$ edges/representation-relation	$O(hED)$	$O(CallD^2)$
Locally separable	$O(D)$ edges/representation-relation	$O(ED)$	$O(CallD)$

Figure 6. Worst-case space requirements for the exploded supergraph for three different classes of dataflow-analysis problems.

in this table (see Figure 9). The table in Figure 7 summarizes the worst-case times required for the Demand-Tabulation Algorithm for six different classes of problems. In each case, the time given is the worst-case time for a single demand. The details of the analysis of the running time of the Demand-Tabulation Algorithm can be found in [Rep94a].

The most efficient exhaustive algorithm known for the class of IFDS problems is the one given in [Rep95]. Its worst-case running times are almost identical to the times given in Figure 7; the only difference is that for an intraprocedural, locally separable problem, the bound for the exhaustive algorithm is $O(ED)$, while the bound for the Demand-Tabulation Algorithm is $O(E)$. The similarity in the worst-case running times of the two algorithms reflects the fact that (theoretically) a dataflow fact at one point might depend on all other facts at all other points. In practice, however, we have found that the Demand-Tabulation Algorithm (applied to a single demand) is much faster than the exhaustive algorithm (see Figure 11).

3.2. The Same-Worst-Case-Cost Property

We have designed the Demand-Tabulation Algorithm so that it has the same-worst-case-cost property with respect to the exhaustive algorithm of [Rep95]. In particular, a call to IsMemberOfSolution can re-use the sets ReachableNodes, VisitedNodes, PathEdge, and SummaryEdge, whose values are preserved across calls. When the Demand-Tabulation Algorithm is used with a request sequence that places demands on all

Class of functions	Asymptotic running time	
	Intraprocedural problems	Interprocedural problems
Distributive	$O(ED^2)$	$O(ED^3)$
h-sparse	$O(hED)$	$O(Call\,D^3 + hED^2)$
Locally separable	$O(E)$	$O(ED)$

Figure 7. Asymptotic running time of the Demand-Tabulation Algorithm (for answering a single demand) for six different classes of dataflow-analysis problems.

nodes of $G^{\#}$, BackwardDFS and UpdateReachableNodes will each traverse a given edge in $G^{\#}$ at most once during the processing of the request sequence. BackwardTabulateSLRPs will traverse a given summary edge or an edge of $G^{\#}$ in procedure p at most D times: once for each node of the form $\langle exit_p, d \rangle$. (The information accumulated in sets PathEdge and SummaryEdge prevents procedure BackwardTabulateSLRPs from performing additional work, and the information accumulated in ReachableNodes and VisitedNodes prevents BackwardDFS and UpdateReachableNodes from performing additional work.) In general, this is bounded by $O(E\,D^3)$, which is the same amount of work that could be performed in the worst case by the exhaustive algorithm given in [Rep95]. Thus, the Demand-Tabulation Algorithm has the same-worst-case-cost property with respect to the exhaustive algorithm.

While this is an important property, it does not, of course, mean that the Demand-Tabulation Algorithm will always outperform the exhaustive algorithm. First, the constant factors are different for the two algorithms. Second, there will be problem instances for which the exhaustive algorithm will not achieve its worst-case cost. Therefore, there will be times when the exhaustive algorithm will outperform the Demand-Tabulation Algorithm (see Figure 12).

4. Experimental Results

4.1. Background to the Experiments

We have carried out two experiments to compare the performance of the Demand-Tabulation Algorithm to that of the exhaustive algorithm of [Rep95], and two further experiments to study the trade-offs between the benefit and overhead of the caching performed by the Demand-Tabulation Algorithm. In all of our reported results, running times reflect the trimmed mean of five data points (*i.e.*, all experiments were run five times, and the average running times were computed after discarding the high and low values).

Three different analysis algorithms were used in the study: (1) the Demand-Tabulation Algorithm, as described above, (2) a non-caching version of the Demand-Tabulation Algorithm (that returns *true* as soon as it visits a node of the form $\langle n, \mathbf{0} \rangle$, reinitializes the set VisitedNodes to $\varnothing$ after each invocation of IsMemberOfSolution, does not maintain the set ReachableNodes, but *does* preserve the sets PathEdge and SummaryEdge across calls to IsMemberOfSolution),[1] and (3) the exhaustive algorithm reported in [Rep95].

The three algorithms described above were implemented in C and used with a front end that analyzes a C program, builds the program's control-flow graph, and then generates the corresponding exploded supergraph for five dataflow-analysis problems:

[1]The non-caching algorithm does not have the same-worst-case-cost property with respect to the exhaustive algorithm. In the worst case, on a request sequence that places demands on all nodes of $G^{\#}$, the non-caching algorithm could perform as much as $\Omega(N\,E\,D^3)$ work, which is worse than the $O(E\,D^3)$ bound on the work performed by the exhaustive algorithm.

Possibly-Uninitialized Variables

This is the problem that we have used as our running example.

Simple Uninitialized Variables

This is the locally separable version of the possibly-uninitialized variables problem, in which a variable is considered to be initialized whenever it is the target of an assignment, regardless of whether the right-hand-side expression includes possibly-uninitialized variables. (So every *simple* uninitialized variable is also *possibly* uninitialized, but not vice versa.)

Live Variables

This is the standard, locally separable problem in which variable x is considered to be live at supergraph node n iff there is a path from n to the end of the program on which x is used before being defined. It is useful to identify assignments to non-live variables: Programming tools might flag them as indicating possible logical errors, and optimizing compilers can use this information to perform dead-code elimination (*i.e.*, by removing such assignments).

Truly Live Variables

This is a non-locally-separable (and more accurate) version of the live-variables problem in which variable x is considered to be truly live at supergraph node n iff there is a path from n to the end of the program on which x is used in a truly live context before being defined, where a truly live context means: in a predicate, or in a call to a library routine, or in an expression whose value is assigned to a truly live variable [Gie81]. Because it is non-locally-separable, the truly-live-variables problem is in some sense a harder problem than the live-variables problem; its results are also more accurate (every truly live variable is also live, but not vice versa) and thus, for example, can lead to more opportunities for dead-code elimination.

Some assignments to variables that are live but not truly live can be discovered by repeatedly solving the live-variables problem and removing assignments to non-live variables until no more assignments to non-live variables are found. However, there are two potential disadvantages to solving the live variables problem rather than the truly live variables problem: the problem needs to be solved repeatedly, and in the presence of cycles in the control-flow graph, there may be assignments to non-truly-live variables that are never discovered (and thus cannot be reported as logical errors or removed).

Constant Predicates

This is a non-locally-separable problem that seeks to determine, for every predicate that consists of a single identifier, whether that predicate is guaranteed to have a constant value (either *true*—non-zero—or *false*—zero). To do this, it performs a simple kind of copy-constant propagation, tracking, for every scalar variable x, whether x might be non-zero, zero, or $\perp$ (an unknown value). Given a predicate that consists of just the identifier x, if the dataflow fact $<x$, non-zero$>$ holds at that points, while neither fact $<x$, zero$>$, nor fact $<x, \perp>$ holds at that point, then the predicate is guaranteed to be *true* (and similarly, it is possible to determine when the predicate is guaranteed to be *false*).

In our experiments, procedure calls via pointers to procedures, and aliasing due to pointers were handled by our C front end as follows:

- For all of the dataflow-analysis problems, every call via a pointer was considered to be a possible call to every procedure of an appropriate type that was passed as a parameter or whose value was assigned to a variable somewhere in the program.
- For all of the dataflow-analysis problems, every memory write via a pointer was considered to be a possible write of every piece of heap-allocated storage and of every variable to which the "address-of" operator (&) was applied somewhere in the program.
- For the live variables, truly live variables, and constant-predicates problems, every memory read via a pointer was considered to be a possible read of every piece of heap-allocated storage and of every variable to which the "address-of" operator was applied somewhere in the program.
- For the possibly-uninitialized variables problem, memory reads were considered to read only the value of the pointer itself. This is because the results of this analysis are suitable for providing feedback to the programmer rather than for guiding an optimizing compiler; it is more important to avoid overwhelming the programmer by reporting hundreds of possibly-uninitialized variables than to be sure that absolutely every possibly-uninitialized variable has been reported. (For the simple uninitialized-variables problem, reads via pointers are irrelevant, since a variable that is the target of an assignment is considered to be initialized regardless of which variables are used to compute the assigned value.)

Of course, the results of the two live-variable analysis problems and of the constant-predicates problem might be improved if we first did a pointer analysis and then used the results of that analysis in setting up the dataflow functions (rather than treating pointers as described above). However, it is interesting to note that even with this very simple treatment of pointers we are able to identify a significant number of assignments to dead variables (see Figure 8) and some constant predicates. Furthermore, the goal of our experiments was to compare the performance of the caching-demand, non-caching-demand, and exhaustive algorithms. We were looking for insights about the *characteristics* of a dataflow problem that predict which algorithm will be best; these characteristics should be independent of the particular problems used, or how they were defined.

Tests were carried out on a Sun SPARCstation 20 Model 61 with 128 MB of RAM. The study used 53 C programs—some standard UNIX utilities, some programs from the SPEC integer benchmark suite [92], and some programs used for benchmarking in previous studies [Lan93,Aus94]. For each program, the table in Figure 8 gives the number of lines of preprocessed source code (with blank lines removed), the parameters that characterize the size of the control-flow graphs (number of procedures, number of call sites, number of control-flow graph nodes), and, for each of the five dataflow-analysis problems, the number of "interesting" program-point/dataflow-fact pairs (see the next paragraph), and the number of these pairs that are in the meet-over-all-valid-paths solution (*i.e.*, if demands are made for all "interesting" program-point/dataflow-fact pairs, this is the number of demands that would be answered *yes*).

Recall that demand analysis is potentially preferable to exhaustive analysis whenever the full set of all dataflow facts at all points is not required. In this case, it may be more efficient to use a demand algorithm, issuing demands only for the program-point/dataflow-fact pairs of interest. To test whether this is true in practice, one of our experiments compares the time required by the exhaustive algorithm to the time required by the Demand-Tabulation Algorithm to answer all "interesting" demands, where "interesting" is defined for each of our dataflow problems as follows: For the two versions of the "uninitialized-variables" problem, every use of a scalar variable x gives rise to the demand "might x be uninitialized here?"; for the two versions of the "live variables" problem, every assignment to a scalar variable x gives rise to the demand "is x live here?"; and for the "constant predicates" problem, every instance of a predicate that consists only of the identifier x gives rise to three demands: "might x be zero here?", "might x be non-zero here?", and "might x be $\perp$ here?".[2]

The table in Figure 9 provides information about the size of the dataflow domain for each dataflow problem for each test program, the sizes of the exploded supergraphs, and the number of summary edges added by the Demand-Tabulation Algorithm when processing all "interesting" demands.

In the following subsections, the times reported for the experiments include the time used to build the exploded supergraphs and to perform dataflow analysis on those graphs (they do not include the time used by the front end to build the test programs' control-flow graphs). Figure 10 shows the ratios of the times used to build the exploded supergraphs to the times used for analysis. Five graphs are shown: one for the exhaustive algorithm, one for the Demand-Tabulation Algorithm used to answer a single demand (using the average time for 20 randomly selected demands), one for the Demand-Tabulation Algorithm used to answer all interesting demands, one for the non-caching demand algorithm used to answer a single demand (using the average time for the same 20 demands used for the Demand-Tabulation Algorithm), and one for the non-caching algorithm used to answer all interesting demands.

It is interesting to compare these ratios with the ratios predicted by the table in Figure 7 (showing the asymptotic running times of the Demand-Tabulation Algorithm for distributive, h-sparse, and locally separable problems). For the h-sparse problems (truly live variables, possibly uninitialized variables, and constant predicates), analysis time is related to the size of the graph by factors of D^2 and D^3. Therefore, one would expect that graph-construction time would tend to take less of the total time as the size of the graph increases. This expectation is born out by our measurements. However, for the locally separable problems (live variables and simple uninitialized variables), analysis time is linear in the size of the graph. Therefore, one would expect the ratio of graph-construction time to total time to be independent of the size of the

[2] Recall that a particular predicate is constant only if one of the first two demands is answered *yes*, while the other two demands are answered *no*. Therefore, the number of demands answered *yes* reported for this problem in Figure 8 is much greater than the number of predicates found to be constant.

Example	Lines of source code	CFG statistics			uninit-vars statistics			live-vars statistics			const-preds statistics	
		P	*Call*	*N*	*# demands*	*# yes*		*# demands*	*# yes*		*# demands*	*# yes*
						possibly uninit	*simple uninit*		*truly live*	*live*		
xref	68	8	13	204	65	0	0	88	79	81	42	14
queens	111	4	4	402	97	0	0	180	165	165	51	22
hash	146	6	9	224	49	1	1	90	74	76	12	4
misr	216	6	8	399	135	0	0	177	159	159	9	3
exptree	238	13	17	438	171	0	0	178	154	158	54	21
dry	241	14	17	329	117	6	5	120	91	100	6	2
chomp	262	21	44	902	247	0	0	361	317	319	54	28
diff.diffh	303	14	49	653	219	4	4	320	277	277	57	21
genetic	336	17	32	691	245	0	0	290	232	236	21	4
anagram	344	15	22	599	199	2	2	261	231	232	39	14
allroots	427	7	19	427	156	0	0	226	188	190	6	2
ul	451	14	35	1366	310	2	1	578	514	514	195	73
ks	574	14	17	1145	389	0	0	491	444	446	24	11
compress	657	15	28	1760	428	0	0	772	679	682	165	67
stanford	665	47	79	1424	675	6	3	612	517	538	51	24
clinpack	695	12	43	1269	583	15	1	595	539	541	12	4
travel	725	15	23	554	316	11	5	290	265	267	75	40
lex315	747	17	102	1130	261	2	2	451	438	440	138	80
sim	748	15	47	2178	1869	635	6	1234	1181	1182	411	159
mway	806	21	42	1819	777	145	12	811	703	705	63	27
pokerd	1099	25	84	2167	720	67	6	996	875	875	192	77
ft	1185	28	48	1017	365	0	0	419	395	395	45	12
ansitape	1222	36	108	2247	447	1	1	1012	848	849	168	75
loader	1255	30	79	2143	542	60	5	894	886	886	165	72
gcc.main	1285	31	97	2406	830	6	1	1128	1017	1017	309	143
voronoi	1394	47	104	1421	696	3	3	686	577	610	168	69
ratfor	1531	52	266	2792	847	232	1	1245	941	946	411	142
livc	1674	86	203	4462	1404	0	0	2052	1611	1613	24	10
struct.beauty	1701	33	212	2797	711	18	6	1276	1062	1068	327	125
diff.diff	1761	41	126	4983	1178	75	4	1839	1538	1540	465	213
xmodem	1809	26	153	3425	743	21	10	1487	1232	1232	543	328
compiler	1908	38	349	3680	619	126	0	1543	1175	1179	714	448
learn.learn	1954	34	77	3849	597	7	0	1475	1172	1172	309	152
gnugo	1963	28	88	3397	1184	120	39	1413	1148	1150	624	260
triangle	1968	18	42	3435	3217	104	29	1765	1610	1622	582	215
football	2075	58	257	6002	2252	36	12	2953	2906	2906	687	329
dixie	2439	36	86	2902	697	36	14	1256	1131	1131	222	89
eqntott	2470	61	215	4598	1766	337	21	2117	1913	1916	405	177
twig	2555	76	222	4383	1278	16	1	2049	1813	1817	312	121
arc	2574	90	254	5912	1903	72	0	2693	2384	2384	666	351
cdecl	2577	32	203	3474	730	2	0	1520	1346	1383	333	151
lex	2645	62	329	6709	2577	201	4	3192	2748	2800	861	363
patch	2746	54	264	5265	1569	216	41	2592	2433	2440	771	377
yacr2	2911	52	157	3934	2314	54	2	1939	1795	1795	468	214
assembler	2994	52	247	5227	1192	28	2	2233	2157	2169	516	185
unzip	3261	40	126	3585	1237	185	16	1715	1559	1565	462	229
tbl	3462	83	314	6538	2149	135	15	2966	2564	2564	1119	512
gcc.cpp	4061	54	216	5828	2874	27	4	2815	2516	2525	1425	766
simulator	4239	99	408	5873	1323	310	9	2217	2181	2181	996	463
agrep	4906	64	136	7837	3919	33	6	3687	3312	3322	1116	571
ptx	5001	48	130	7326	4354	734	96	3475	3268	3281	1830	837
li	6054	132	555	5960	1480	343	17	3267	3022	3065	1500	1161
bc	6745	98	675	8423	2511	660	13	3267	3022	3065	654	298

Figure 8. Test program information.

Example	uninit-vars statistics					live-vars statistics					const-preds statistics		
	D	$G^{\#}$ edges		summary edges		D	$G^{\#}$ edges		summary edges		D	$G^{\#}$ edges	sum-mary edges
		possibly uninit	*simple uninit*	*possibly uninit*	*simple uninit*		*truly live*	*live*	*truly live*	*live*			
xref	18	3493	3376	11	11	18	3627	3489	99	27	43	7380	0
queens	40	16729	16556	22	18	45	18702	17706	42	14	82	32893	23
hash	13	2493	2401	2	0	15	3207	3109	15	10	34	5986	0
misr	30	11377	11181	12	6	31	12500	11825	44	18	85	31431	0
exptree	27	10596	10395	123	112	28	11329	11051	203	86	79	29508	35
dry	27	8641	8510	50	50	35	11658	11192	220	57	58	16681	0
chomp	23	18786	18290	261	151	25	20881	19555	400	119	67	51995	162
diff.diffh	63	36198	35799	160	128	72	44792	43181	444	141	100	39295	158
genetic	35	24157	23848	295	270	42	31205	30647	829	247	100	64908	67
anagram	72	35979	35594	66	62	86	45830	44543	101	45	115	40131	27
allroots	29	12280	12003	32	0	34	14947	14725	162	44	73	29044	0
ul	45	62661	62123	679	649	51	71871	69634	847	645	133	180002	976
ks	48	54686	53695	157	81	56	62673	60180	91	47	91	97043	0
compress	94	165091	164214	396	344	108	212791	184215	460	380	181	294058	136
stanford	66	93488	92944	296	258	85	133601	128626	2279	159	106	124818	14
clinpack	43	55205	54551	210	198	54	71690	68990	1654	235	115	141760	0
travel	90	50598	50118	201	150	103	58891	57549	357	177	166	82632	4
lex315	41	38734	38253	553	439	48	51208	50846	601	176	109	89476	102
sim	81	192191	190725	1195	1110	85	209054	203021	1765	1064	235	544236	10
mway	119	203320	200451	1205	941	144	244501	237500	3021	306	256	399681	348
pokerd	67	140864	139371	607	568	75	166329	158840	1494	672	109	194389	447
ft	39	38717	38252	202	161	44	43850	43237	537	314	79	70695	140
ansitape	126	273206	271608	6414	4393	174	464349	399937	35016	2410	286	553269	1113
loader	64	125521	124495	471	267	70	144834	141695	1485	500	130	223230	0
gcc.main	107	272511	269954	3273	2943	112	298455	289970	3624	2208	223	521955	1331
voronoi	81	110663	108563	3596	2971	85	126283	120223	5030	1030	226	278519	280
ratfor	59	177753	176195	6587	7401	93	317360	312737	11150	7040	175	476097	936
livc	57	239190	236321	5616	5597	121	565720	561393	4369	951	157	611871	0
struct.beauty	83	261733	260599	6887	6610	111	382174	374492	8322	5400	208	597034	1540
diff.diff	160	613366	605402	1771	1601	182	947107	696872	3601	1241	388	1361962	874
xmodem	121	406443	404237	1621	1607	134	475438	460730	6014	1732	271	827700	1960
compiler	44	177068	176087	10305	8389	52	229519	226545	13993	8903	130	472640	8675
learn.learn	89	337739	336109	1872	1824	124	488639	474089	1572	1410	175	610777	2326
gnugo	53	167169	164727	1313	943	62	206718	201418	2041	740	121	341553	66
triangle	108	334696	331110	0	0	137	426522	384299	885	222	232	631468	4
football	97	484289	481017	5195	5148	129	728595	719297	5310	2941	250	1122236	1529
dixie	87	260924	258759	1572	1335	117	354942	348047	1964	1012	169	471333	594
eqntott	71	348804	346608	3948	3797	95	505852	488345	11998	6775	208	969982	1415
twig	135	651217	636825	7905	7719	154	1415560	744234	41523	6776	397	1787337	1511
arc	158	954734	951715	11819	11536	187	1189180	1172834	14945	10404	385	2161776	5439
cdecl	74	271604	270265	7201	7011	106	427621	416435	12266	7587	217	751691	1611
lex	130	972295	969098	20229	19857	157	1219914	1199864	20890	13767	388	2753136	4080
patch	144	799454	794874	11950	11974	162	966122	933514	15668	10586	331	1685318	8429
yacr2	106	348885	346615	3245	3311	107	357509	353996	4158	1215	280	840228	59
assembler	70	378857	375720	2633	2324	86	473472	454803	8500	2563	154	774266	385
unzip	179	711922	707762	8092	7266	208	856644	831084	16929	6239	424	1595253	3157
tbl	89	589267	585684	11373	10672	135	983139	977401	12325	6758	226	1380852	9775
gcc.cpp	138	830388	822277	5314	5026	168	1154487	1053956	18954	5380	316	1722977	4565
simulator	61	372914	370631	5041	3703	75	493171	486588	10557	3400	121	642022	1196
agrep	151	1191039	1187014	6043	5849	186	1526409	1481386	9424	4112	319	2321891	5858
ptx	271	2017638	1995183	4971	4628	285	2320122	2127202	10831	3409	622	4213669	3687
li	113	698042	695496	38862	27950	124	818403	809486	51551	44612	325	1795698	4826
bc	152	1372175	1367767	49964	49084	174	1688319	1661439	56647	33835	352	2813877	22448

Figure 9. Graph sizes and added summary edges.

graph. This expectation is *not* born out by our measurements; instead, the ratio tends to decrease as the size of the graph increases.

4.2. Experiments

Experiment 1: Single demand vs. Exhaustive

Our first two experiments compared the Demand-Tabulation Algorithm with the exhaustive algorithm. Our first experiment reflects what might happen when dataflow analysis is used in the context of a tool that intersperses demands and program modifications (so if an exhaustive algorithm is used, it must be re-run whenever a demand is made following a modification). In this case, it is reasonable to compare the time required by the exhaustive algorithm with the time required by the demand algorithm to answer a single demand. Therefore, for this study, we recorded the following data for each dataflow-analysis problem and for each test program: (1) the time used by the exhaustive algorithm to build the exploded supergraph and to find the meet-over-all-valid-paths solution; (2) the average time used by the Demand-Tabulation Algorithm to build the exploded supergraph and to answer a single demand (using 20 randomly selected demands). This data is summarized in Figure 11.

Although there are a few cases where the Demand-Tabulation Algorithm is slower than the exhaustive algorithm, they are all tests for which the running times are trivial (less than 3 seconds). It seems clear that overall, the Demand-Tabulation Algorithm is preferable to the exhaustive algorithm when the goal is to answer a single demand.

Experiment 2: Sequence of demands vs. Exhaustive

Our second comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm reflects what happens when "complete" dataflow information is desired (*i.e.*, when it is desired to know, for all "interesting" program-point/dataflow-fact pairs, whether that pair is in the meet-over-all-valid-paths solution). Therefore, for this study we recorded the time used by the Demand-Tabulation Algorithm to answer the sequence of all "interesting" demands, for each dataflow-analysis problem and for each test program, and compared those times to the times required by the exhaustive algorithm. This data is summarized in Figure 12.

The Demand-Tabulation Algorithm outperforms the exhaustive algorithm in all cases for the *constant-predicates* and *live-variables* problems, and in all but three cases for the *truly-live-variables* problem; it is clearly the algorithm of choice in these cases. For the two versions of the *uninitialized-variables* problem, the Demand-Tabulation Algorithm is almost always slower than the exhaustive algorithm; sometimes significantly so. The Demand-Tabulation Algorithm is clearly not the algorithm of choice for these problems.

Graph-Construction Time vs Total Time

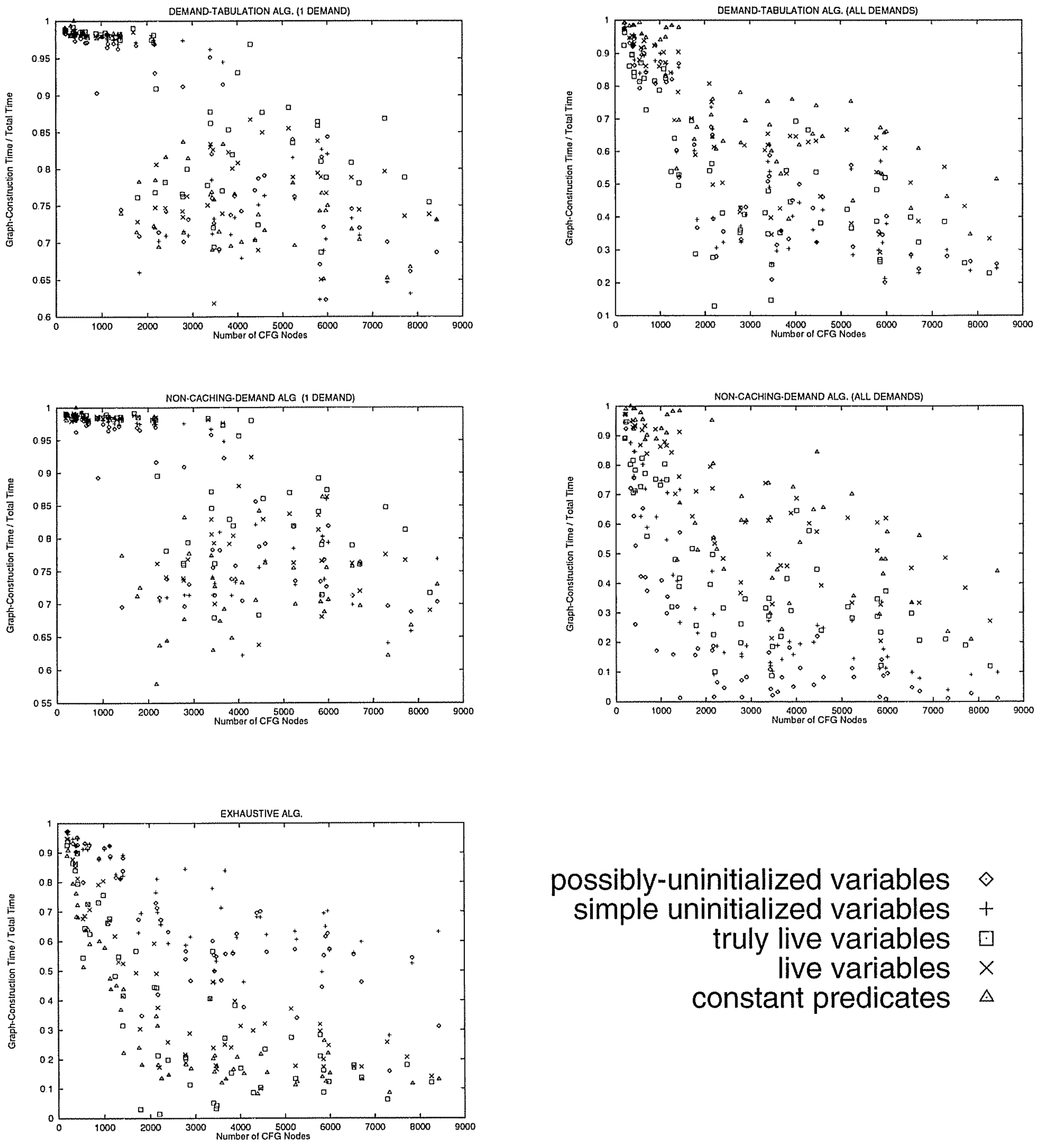


Figure 10. Ratios of times used to build the exploded supergraph to times used for analysis for the three different algorithms.

Experiment 1: Single Demand vs Exhaustive

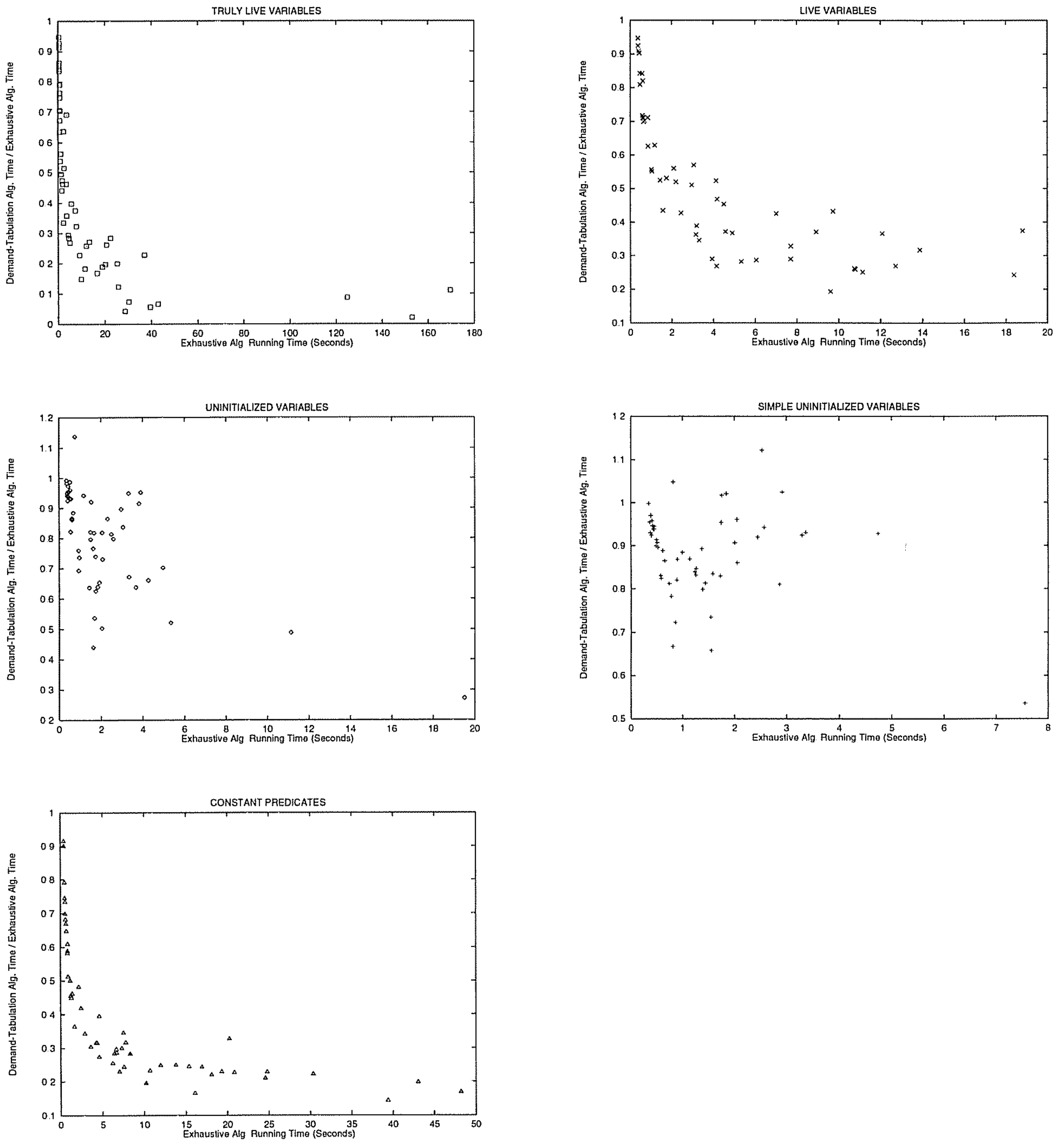


Figure 11. First comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. The exhaustive algorithm is used to find the entire meet-over-all-valid-paths solution, and the Demand-Tabulation Algorithm is used to answer a single demand.

Experiment 2: Sequence of Demands vs Exhaustive

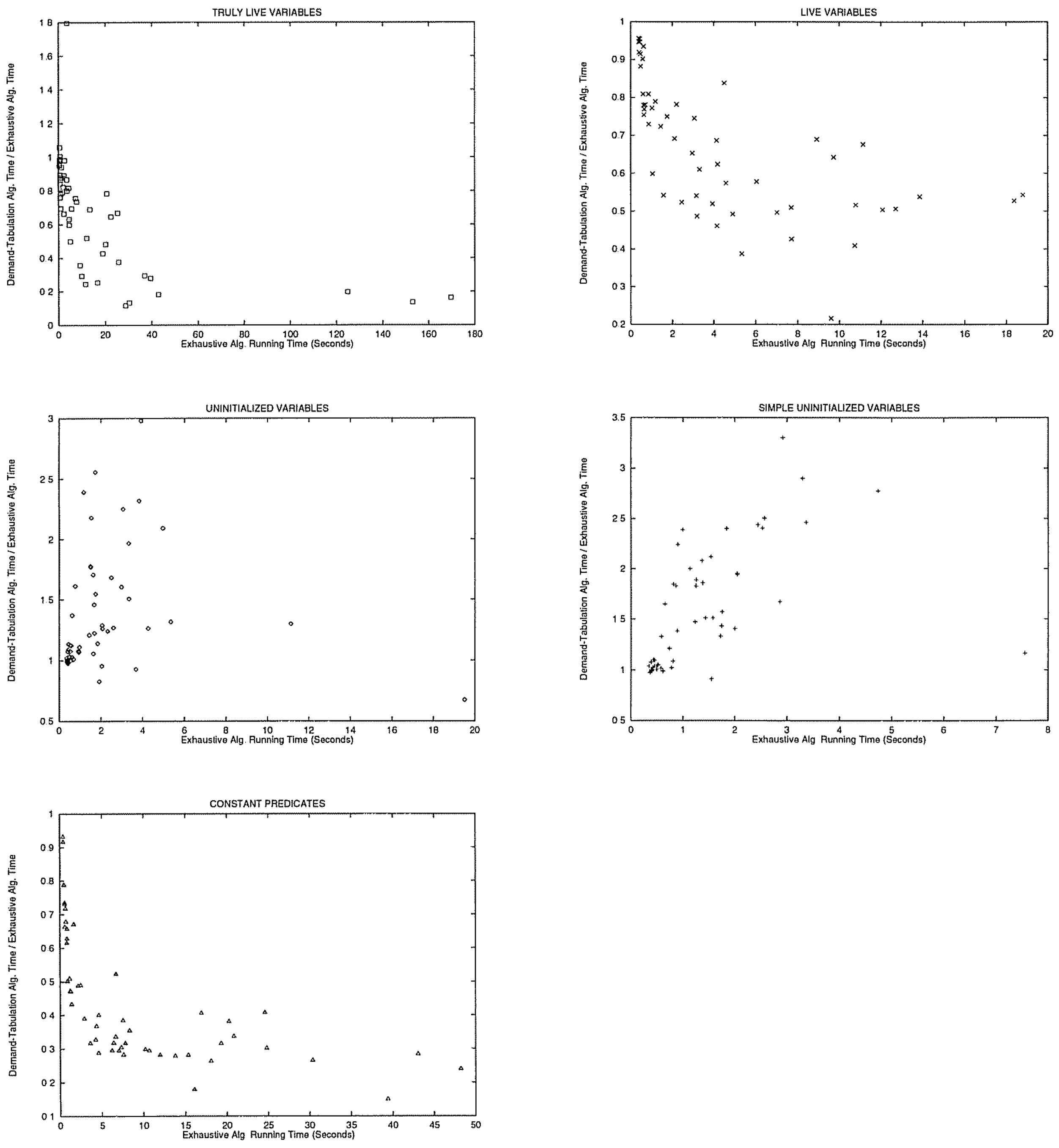


Figure 12. Second comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. The exhaustive algorithm is used to find the entire meet-over-all-valid-paths solution, and the Demand-Tabulation Algorithm is used to answer all "interesting" demands.

We believe that there are two characteristics of dataflow problems that are reasonable predictors of the relative speeds of the Demand-Tabulation Algorithm (applied to all interesting demands) and the exhaustive algorithm:

1. The number of demands, relative to the size of the exploded graph.

2. The percentage of demands with "yes" answers.

If the number of demands is very small, clearly the Demand-Tabulation Algorithm will visit many fewer nodes than the exhaustive algorithm, and so less time is likely to be required for the Demand-Tabulation Algorithm. If most demands are answered "yes", the nodes visited by the Demand-Tabulation Algorithm will also be visited by the exhaustive algorithm; however, since demands are not placed for all facts at all points, the Demand-Tabulation Algorithm should still be faster. However, if most demands are answered "no", the Demand-Tabulation Algorithm may visit many more nodes than the exhaustive algorithm: demands answered *no* correspond to unreachable exploded supergraph nodes, so the exhaustive algorithm does not visit those nodes or any of their predecessors; however, the demand algorithm starts at those nodes and visits *all* predecessors, eventually discovering that none of them is in the ReachableNodes set.

In the case of the two *live-variables* problems, most of the demands ("is x live at this assignment?") lead to a *yes* answer, while in the case of the two *uninitialized-variables* problems, most of the demands ("might x be uninitialized at this use?") lead to a *no* answer.

The graph in Figure 13 plots the percentage of demands that are answered *yes* versus the ratio of the running times of the two algorithms for the five dataflow-analysis problems.

Based on the results of our first two experiments, we hypothesize that when the goal is to answer demands at most program points, and it is expected that most demands will be answered *no*, the exhaustive algorithm will be the algorithm of choice. However, when the expected number of demands is small (for example, in an interactive tool, or in a restructuring tool that is likely to demand dataflow information only for a small part of a program before performing a transformation, or for a problem like the *constant-predicates* problem), or it is expected that most demands will be answered *yes*, then the Demand-Tabulation Algorithm will be the algorithm of choice.

Experiment 3: Caching vs Non-caching demand (single demand)

The goal of our third and fourth experiments was to study the tradeoffs between the benefit and overhead of caching, first on a single demand and then on a sequence of demands.

For our third experiment we applied the non-caching demand algorithm to the same 20 randomly selected demands used in Experiment 1 (starting the algorithm from scratch for each demand as was done for the Demand-Tabulation Algorithm), and we computed the average running time for a single demand. The results of this experiment are shown in Figure 14.

Experiment 2: Time Ratios vs. Percentage of "yes" Answers

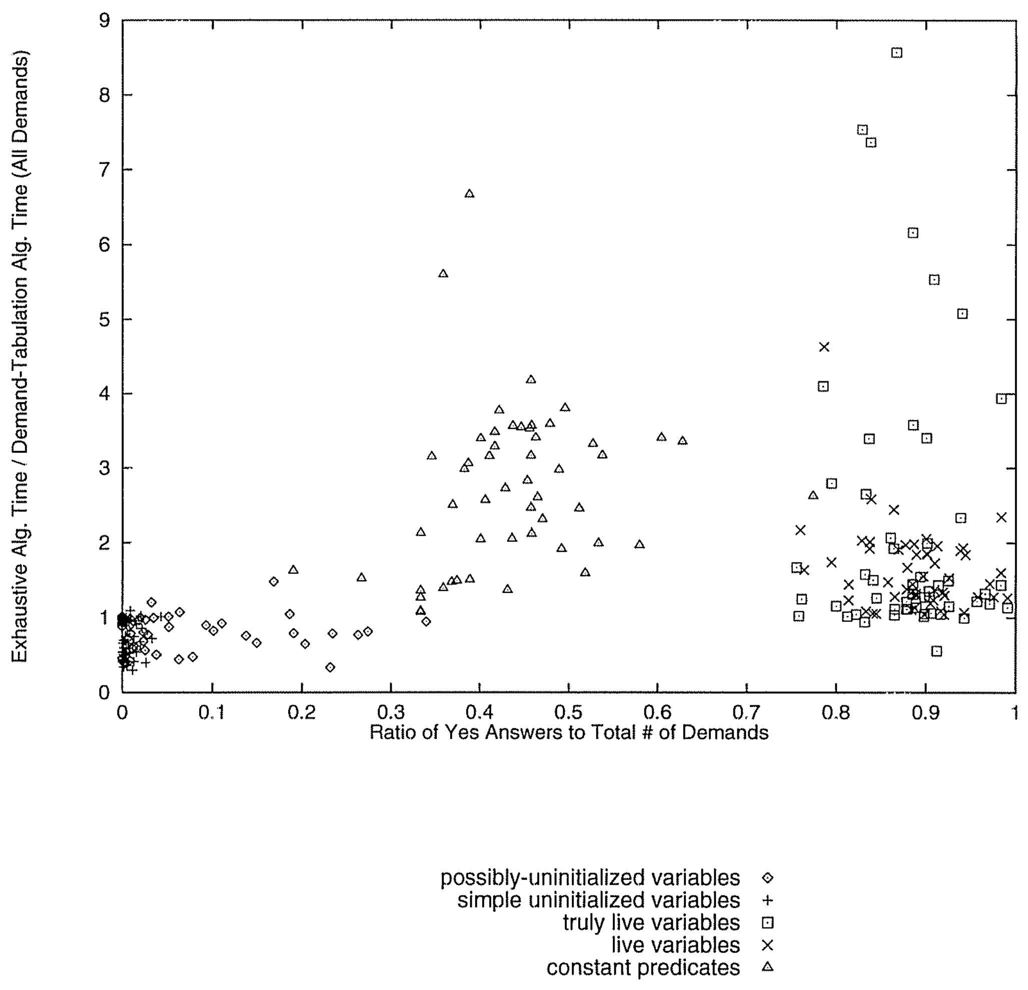


Figure 13. When most demands are answered *yes*, the Demand-Tabulation Algorithm is likely to outperform the exhaustive algorithm; the situation is reversed when most demands are answered *no*.

As expected, caching involves some overhead (about 50% in the worst case for our experiments). This is due partly to the extra work required to maintain the ReachableNodes and VisitedNodes sets, and partly due to the fact that more time is required to build the exploded supergraph. In particular, the Demand-Tabulation Algorithm must traverse edges in both directions (BackwardDFS traverses edges backwards and UpdateReachableNodes traverses edges forwards) while the non-caching demand algorithm only traverses edges backwards. Thus, the Demand-Tabulation Algorithm must include both predecessor and successor information in the exploded supergraph, while the non-caching demand algorithm only needs to include predecessor information.

Experiment 4: Caching vs Non-caching demand (sequence of demands)

For our final experiment, we applied the non-caching demand algorithm to the same sequences of all "interesting" demands to which the Demand-Tabulation Algorithm was applied in Experiment 2. The results of this experiment are shown in Figure 15.

For a sequence of demands, the benefits of caching outweigh its overhead in all cases for the *possibly-uninitialized variables* and the *simple uninitialized-variables* problems (however, recall that we have already concluded that the exhaustive algorithm is superior to the Demand-Tabulation Algorithm for a sequence of demands for these two problems). For the *truly live variables* problem the caching algorithm is faster in all but five cases (and in those cases it is only about 5% slower). For the *constant-predicates* problem, caching is a win in about half of the cases, and for the *live-variables* problem, in about two-fifths of the cases. However, for both of these problems, it seems that the caching algorithm is still the algorithm of choice:

- In the worst case for the caching algorithm, it took about 1.4 times as long as the non-caching algorithm; in the worst case for the non-caching algorithm, it took about 1.8 times as long as the caching algorithm.
- There is only one case in which the time for the non-caching algorithm is at least 25% less than the time for the caching algorithm, while there are seven cases in which the time for the caching algorithm is at least 25% less than the time for the non-caching algorithm.

5. Relation to Previous Work

Until very recently, work on demand-driven dataflow analysis only considered the *intra*procedural case (*cf.* [Bab78]) and work on *inter*procedural dataflow analysis only considered the exhaustive case (*cf.* [Sha81,Cal88,Cal86,Kno92]). Because in intraprocedural dataflow analysis *all* paths in the control-flow graph are assumed to be valid execution paths, the work on demand-driven intraprocedural dataflow analysis does not extend to the interprocedural case, where the notion of *realizable* paths is important.

One approach to obtaining demand algorithms for interprocedural dataflow-analysis problems was described by Reps [Rep94c,Rep94b]. Reps presented a way in which algorithms that solve demand versions of interprocedural analysis problems can be obtained automatically from their exhaustive counterparts (expressed as logic programs) by making use of the "magic-sets transformation", a general transformation developed in the logic-programming and deductive-database communities for creating efficient demand versions of (bottom-up) logic programs [Roh86,Ban86,Bee87,Ull89]. Reps illustrated this approach by showing how to obtain a demand algorithm for the interprocedural locally separable problems. Subsequent work by Reps, Sagiv, and Horwitz extended the logic-programming approach to the class of

Experiment 3: Caching Demand vs Non-Caching Demand (Single Demand)

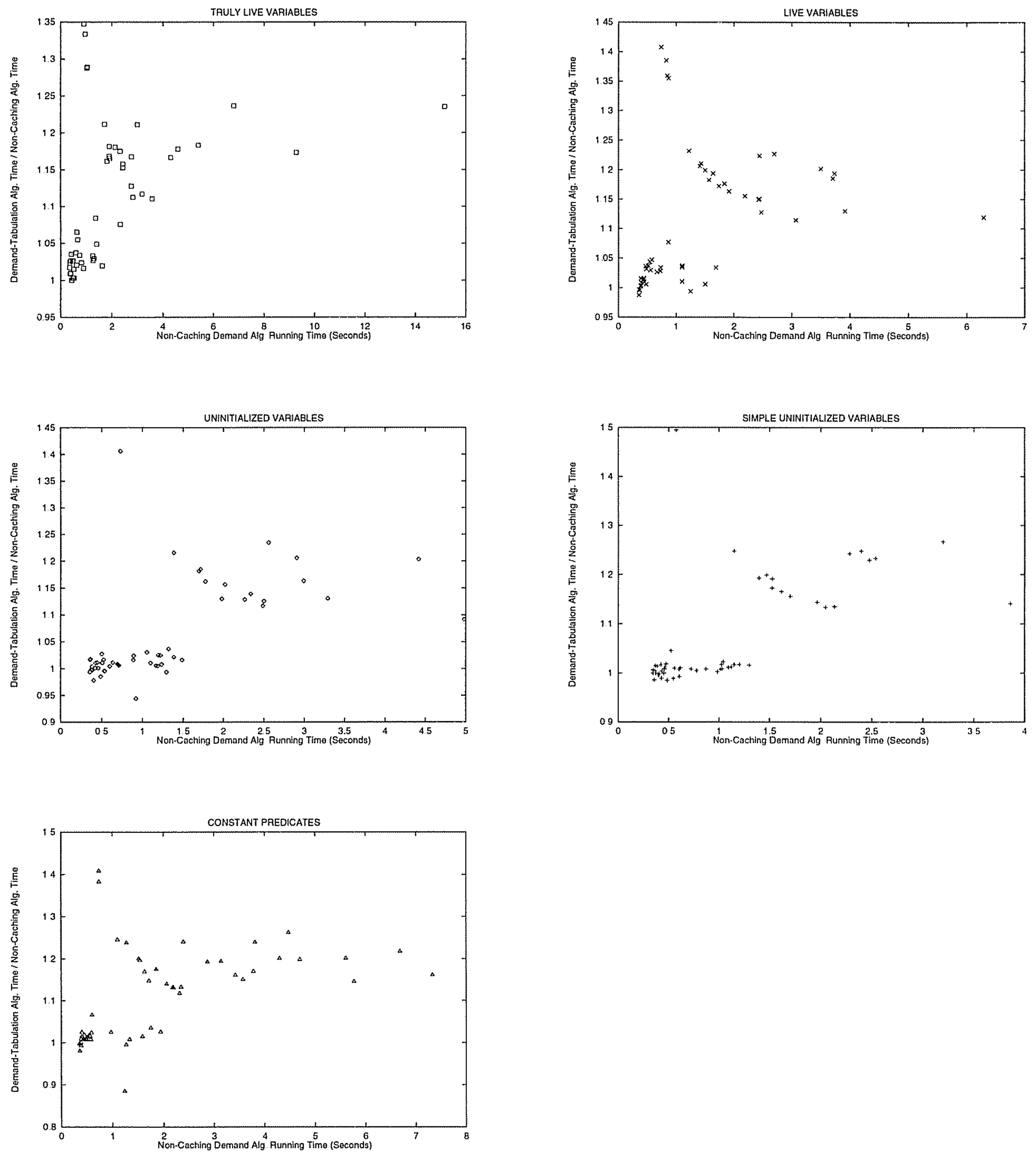


Figure 14. Comparison of the caching and non-caching demand algorithms for a single demand.

Experiment 4: Caching Demand vs Non-Caching Demand (Sequence of Demands)

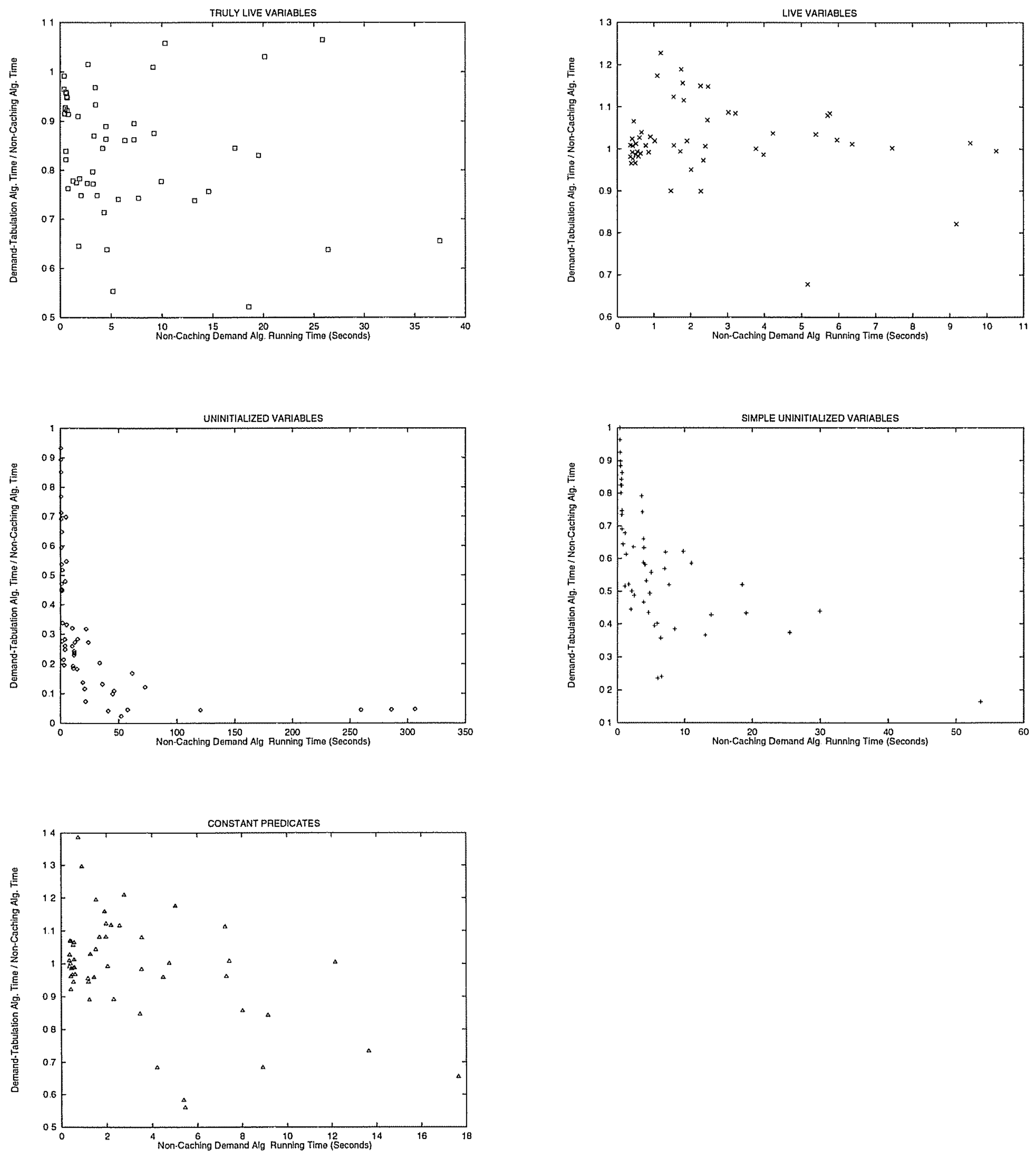


Figure 15. Comparison of the caching and non-caching demand algorithms for a sequence of demands.

IFDS problems [Rep94a,Rep95]. (The latter papers do not make use of logic-programming terminology; however, the exhaustive algorithms described in the papers have straightforward implementations as logic programs. Demand algorithms can then be obtained by applying the magic-sets transformation.)

Several people, leery of the (space, time, and conceptual) overheads involved in using logic databases, questioned whether the logic-programming approach to obtaining demand algorithms for interprocedural dataflow analysis can really produce implementations that are efficient enough to be used in real-world program-analysis tools. Although the jury is still out on this issue (waiting for improved logic-database implementations), it is natural to ask a related question: "Is there a way to adapt the ideas so that they can be used in program-analysis tools written in imperative programming languages?"

The present paper can be viewed as answering this question in the affirmative. The two basic ideas used in the magic-sets transformation are *propagation of queries* and *caching of results*, and it is fairly easy to transfer these notions over to demand algorithms written in an imperative programming language (such as C). The Demand-Tabulation Algorithm given in Section 3 can be viewed as an analog of the magic-sets-transformed exhaustive dataflow analysis program: The operations that push nodes onto NodeStack in BackwardDFS and insert edges into EdgeWorkList in BackwardTabulateSLRPs for subsequent processing can be viewed as "query-propagation" operations; the sets ReachableNodes, VisitedNodes, PathEdge, and SummaryEdge, whose values are preserved across calls, can be viewed as caches of previously computed results (and previously computed "intermediate values").

On the other hand, there are a number of benefits obtained when an imperative programming language is used to implement these ideas. The most important benefit is that the algorithm of Section 3 has a simple, low-overhead implementation in an imperative programming language. The implementation is based on array indexing and linked lists, and involves neither term-unification nor term-matching. In addition, an imperative implementation has the opportunity to exploit specific properties of the problem that are not present in all logic programs (and hence would not be exploited by either the magic-sets transformation or the bottom-up "engine" used for evaluating logic-programs). The two forms of "early cut-off" employed by the algorithm given in Section 3 provide two examples of how such properties can be exploited to improve performance:[3]

- The use of a depth-first-search strategy in BackwardDFS allows BackwardDFS to terminate as soon as a node in ReachableNodes is encountered.
- In BackwardTabulateSLRPs, once it is known that there is a path from some node of the form $\langle m, \mathbf{0} \rangle$ to $\langle exit_p, d \rangle$, the path edge $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$ is inserted into PathEdge. Subsequent tests for membership of this edge in PathEdge allow BackwardTabulateSLRPs to avoid processing more path edges with

[3]A previous version of the Demand-Tabulation Algorithm that was presented in [Hor95] did not employ either of these strategies for early cut-off.

target $\langle exit_p, d \rangle$.

Of course, these early cut-offs can only be taken when the answer to the current demand is *yes*. Thus, these improvements to the algorithm are most significant for problems with a high ratio of *yes* answers.

A related approach to obtaining demand versions of dataflow-analysis algorithms has been investigated by Duesterwald, Gupta, and Soffa, first for intraprocedural problems [Due93] and subsequently for interprocedural problems [Due95]. In their approach, a set of dataflow equations is set up on the flow graph (but as if all edges were reversed). The flow functions on the reversed graph are the (approximate) inverses of the original forward functions. Their algorithm for solving such problems is a demand-driven algorithm that repeatedly propagates a query from a node in the control-flow graph to the node's predecessors. (The appropriate query is generated by applying the inverse dataflow function.) Caching also plays a role: Values of "summary functions" are tabulated; these express how queries at return sites generate queries at call sites.

The Duesterwald-Gupta-Soffa approach is more general than ours because it can handle distributive problems on any finite lattice, while the Demand-Tabulation Algorithm is limited to distributive problems on finite subset lattices. (They can also provide approximate information in cases where the flow functions are monotonic but not distributive.) However, this generality is achieved at some cost. When applied to an IFDS problem, the worst-case cost of the algorithm given in [Due95] is exponential: $O(E\,D\,2^D)$, while the worst-case cost of the Demand-Tabulation Algorithm is polynomial: $O(E\,D^3)$.

This is not the entire story, however, because the Duesterwald-Gupta-Soffa framework can be used as a *conceptual* framework for deriving particular algorithms for specific problems as special cases of their general methods. For instance, this is done for copy-constant propagation in the second half of [Due95], and yields an algorithm with polynomial running time. Our algorithm can be viewed as the specialization of the Duesterwald-Gupta-Soffa framework to the entire class of IFDS problems. (But no further specialization for a particular problem is necessary to obtain an algorithm with polynomial running time. That is, from a specification of the edge transformers for a particular problem instance, our techniques automatically yield an algorithm with polynomial running time.)

Another framework for demand analysis is given in [Sag]. That framework applies to a class of distributive problems that is strictly larger than the IFDS problems and that is incomparable to the class to which the Duesterwald-Gupta-Soffa framework applies. (The framework of [Sag] applies only to distributive problems, whereas the Duesterwald-Gupta-Soffa framework can be applied to some non-distributive problems. However, the Duesterwald-Gupta-Soffa framework requires that the lattice of dataflow values have a finite number of elements, whereas the framework of [Sag] requires only that the lattice have finite height.) The Demand-Tabulation Algorithm can be viewed as a specialization of the algorithm of [Sag] to IFDS problems, but with several improvements (*e.g.*, the two forms of "early cut-off" discussed above, and the elimination of an entire phase of the more general algorithm).

At the other end of the spectrum, it is interesting to compare our work with Callahan's "program-summary-graph" algorithm for flow-sensitive side-effect analyses [Cal88]. As discussed in [Rep95], Callahan's problems fall into the class of locally separable IFDS problems, the subclass of the IFDS problems that corresponds to interprocedural versions of the classic gen/kill problems.[4] From the standpoint of asymptotic worst-case complexity, Callahan's problems can actually be solved more efficiently with the algorithm from [Rep95] than with the algorithm given by Callahan. Because the present paper provides a demand version of the exhaustive algorithm from [Rep95] (where the demand algorithm has the same worst-case complexity as the exhaustive algorithm), our work also provides a demand algorithm for solving all of Callahan's problems.

References

92. , "SPEC Component CPU Integer Release 2/1992," (CINT92), Standard Performance Evaluation Corporation (SPEC), Fairfax, VA (1992).

Aus94.
Austin, T. M., Breach, S. E., and Sohi, G., "Efficient detection of all pointer and array access errors," *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation,* (Orlando, FL, June 20-24, 1994), *ACM SIGPLAN Notices* **29**(6) pp. 290-301 (June 1994).

Bab78.
Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Informatica* **10**(3) pp. 265-272 (October 1978).

Ban86.
Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic sets and other strange ways to implement logic programs," in *Proceedings of the Fifth ACM Symposium on Principles of Database Systems,* (Cambridge, MA), (March 1986).

Ban79.
Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, January 29-31, 1979), (January 1979).

Bee87.
Beeri, C. and Ramakrishnan, R., "On the power of magic," pp. 269-293 in *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* (San Diego, CA, March 1987), (March 1987).

Cal86.
Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L., "Interprocedural constant propagation," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 152-161 (July 1986).

Cal88.
Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).

Coo88.
Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988),

[4]Strictly speaking, Callahan's problems are not quite IFDS problems because they are concerned with computing information that summarize the effects of a procedure, rather than what must be true at a program point in all calling contexts. This is only a minor technical difference, and does not invalidate the point being made above.

ACM SIGPLAN Notices **23**(7) pp. 57-66 (July 1988).

Coo89.
Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, January 11-13, 1989), (January 1989).

Due93.
Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven program analysis," Technical Report TR-93-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA (October 1993).

Due95.
Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven computation of interprocedural data flow," in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, January 23-25, 1995), (January 1995).

Fis88.
Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).

Gie81.
Giegerich, R., Moncke, U., and Wilhelm, R., "Invariance of approximative semantics with respect to program transformations.," pp. 1-10 in *Informatik-Fachberichte 50*, Springer-Verlag, Berlin Heidelberg New York (1981).

Hor86.
Horwitz, S. and Teitelbaum, T., "Generating editing environments based on relations and attributes," *ACM Transactions on Programming Languages and Systems* **8**(4) pp. 577-608 (October 1986).

Hor95.
Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (October 1995).

Kil73.
Kildall, G., "A unified approach to global program optimization," pp. 194-206 in *Conference Record of the First ACM Symposium on Principles of Programming Languages,* (Boston, MA, October 1-3, 1973), (October 1973).

Kno92.
Knoop, J. and Steffen, B., "The interprocedural coincidence theorem," pp. 125-140 in *Proceedings of the Fourth International Conference on Compiler Construction*, (Paderborn, FRG, October 5-7, 1992), *Lecture Notes in Computer Science,* Vol. 641, ed. U. Kastens and P. Pfahler,Springer-Verlag, New York, NY (1992).

Lan93.
Landi, W., Ryder, B., and Zhang, S., "Interprocedural modification side effect analysis with pointer aliasing," pp. 56-67 in *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June 23-25, 1993), (June 1993).

Lin84.
Linton, M.A., "Implementing relational views of programs," pp. 132-140 in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, April 23-25, 1984), (April 1984).

Mas80.
Masinter, L.M., "Global program analysis in an interactive environment," Tech. Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA (January 1980).

Rep94a.
Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," Technical Report 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994).

Rep94b.
Reps, T., "Demand interprocedural program analysis using logic databases," in *Applications of Logic Databases*, ed. R. Ramakrishnan,Kluwer Academic Publishers, Boston, MA (1994).

Rep94c.
Reps, T., "Solving demand versions of interprocedural analysis problems," pp. 389-403 in *Proceedings of the Fifth International Conference on Compiler Construction*, (Edinburgh, Scotland, April 7-9, 1994), *Lecture Notes in Computer Science,* Vol. 786, ed. P. Fritzson,Springer-Verlag, New York, NY (1994).

Rep95.
Reps, T., Sagiv, M., and Horwitz, S., "Precise interprocedural dataflow analysis via graph reachability," in

Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages, (San Francisco, CA, January 23-25, 1995), (January 1995).

Roh86.
Rohmer, R., Lescoeur, R., and Kersit, J.-M., "The Alexander method, a technique for the processing of recursive axioms in deductive databases," *New Generation Computing* **4**(3) pp. 273-285 (1986).

Sag.
Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, (). (To appear.)

Sha81.
Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

Ull89.
Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies,* Computer Science Press, Rockville, MD (1989).

Wei84.
Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).