Incremental Data Flow Analysis in a Structured Program Editor

Frank Kenneth Zadeck

IBM T. J. Watson Research Center

1.0 Abstract
For many years, the generally accepted methods for solving global data flow analysis problems (GDFAPs) have relied on the bit vector paradigm [Alle 70] [Kemn 81]. In these algorithms, the information for each variable is represented as a position in a bit vector. These vectors are propagated from node to node during the analysis of the program. This paper investigates a different paradigm for solving GDFAPs. In this strategy, the data flow solution for each variable is computed independently. A new algorithm for solving some GDFAPs is presented here. While it is expected that this algorithm will run slower in the batch compiling environment, there are conditions under which this algorithm can be expected to be superior to those in the bit vector class.

2.0 Basic Definitions
This section contains basic definitions in the area of Data Flow Analysis. Those readers familiar with this area should feel free to skip to section 3.0.

• Basic Block - A sequence of statements where: if any statement in the sequence is executed, all statements in the sequence are executed.

• The Data Flow Analysis Problem - Given a control flow structure, the object is to discover the nature of the data flow, i.e. which definitions of program quantities can affect which uses, within the program. Global problems are those problems that are concerned with the data flow solution within a single procedure.

• Information Chain - A tuple that connects a location of one type of information, the begin site, with the location of another type of information, the end site. The process of constructing all such tuples is called information chaining.

• Def-Use Chains - Information chains that are built from every definition site to every reachable use site for each variable in the program.

This research was supported under NSF grant MCS81-04006 and was done while the author was at Rice University.

2.1 Introduction to Lattice Theory
The data flow problem can be formalized by casting it in the structure of lattice theory. This section and the following section have been essentially taken from Kam and Ullman, [Kawi 76]. Some definitions have been modified or expanded for use here.

A semilattice L is a set L with a binary meet operation, , such that for all , , in L:

- idempotent
- commutative
- associative

Lattices may have:

- bottom element
- top element

It is assumed here that all semilattices have a bottom element. Given a semilattice, a sequence of elements of L is said to be a chain if for 1 ≤ i ≤ j we have . L is said to be bounded if for each there is a constant such that any chain beginning with has length at most . If L is bounded, then we can take the meet over countably infinite sets if we define:

The fact that L is bounded assures us that the limit does exist.
2.2 Lattice Theory Model for Data Flow Analysis

Continuing to take from Kam and Ullman:

Following Kildall [Kild 73], we treat data flow analysis problems as follows. We choose a semilattice \( L \) and attach to its elements a meaning, normally data which could reach a point in a flow graph. We associate with each node of the flow graph a function \( f \) from \( L \) to \( L \) which intuitively represents how data is transformed when control passes through the block of code represented by that node.

In what follows, we find it necessary to consider the set of all functions which could be associated with some node of the flow graph. That is, having selected a semilattice \( L \) and some intended meaning for lattice elements, the admissible functions are those which reflect the actions of straight line blocks of code on elements of \( L \). We abstract the notion of such a set of functions in the following definition.

Given a bounded semilattice \( L \), a set of functions \( F \) on \( L \) is said to be an admissible set of functions for \( L \) if and only if the following conditions are satisfied:

1. Each \( f \in F \) distributes over meet, \( \wedge \)
   \[ (\forall x, y \in L)(\forall f \in F)[f(x \wedge y) = f(x) \wedge f(y)] \]
2. There exists an identity function \( i \) in \( F \), such that,
   \[ (\forall x \in L)[i(x) = x] \]
3. \( F \) is closed under composition, i.e., \( f \circ g \in F \Rightarrow f \in F \wedge g \in F \)
   where
   \[ (\forall f, g \in F)[f \circ g(x) = f(g(x))] \]
4. For each \( x \in L \), there exists a finite subset \( H \subseteq F \) such that
   \[ x = \bigcap_{f \in H} f(x) \]

2.3 Graph Theory Model for Data Flow Analysis

A graph that represents the control flow of a computer program is called a program flow graph. A directed graph \( G = (N, E, n_0) \) represents the program.

- \( N \) is the set of nodes. Each node in this graph typically corresponds to a single basic block in the program.
- \( E \) is the set of edges. The edges represent the block to block transfers.
- \( n_0 \) is the initial node in the graph. This node represents the entry point for the routine.

Two special notations for \( E \) will be used in this paper:

- \( \text{Succ}(x) \) is the set of arcs, leaving each node \( x \), that travel in the direction of control flow.
- \( \text{Pred}(x) \) is the set of edges that is obtained if each edge in the \( \text{Succ} \) set is reversed. Each edge in \( \text{Pred} \) runs in the direction against control flow.

2.4 Data Flow Problems

Data flow problems have been formulated to solve a variety of problems in compiler optimization and program reliability. The problems can be divided using various criteria into many different categories. The categories can affect such quantities as:

1. The time and space required to solve the problem.
2. The precision of the information obtained.
3. Applicable algorithms.
4. Uniqueness of the solution.

2.4.1 Forward Versus Backward

Data flow problems can be divided into categories depending on the direction that information must be propagated to produce the desired result.

1. Forward flow problems are those which, given a point in the program, ask what can happen before control reaches the point (i.e., what definitions can affect computations at that point).
2. Backward flow problems are those which, given a point in the program, ask what can happen after control leaves that point (i.e., what uses can be affected by computations at that point).

A few algorithms, such as \( T1-T2 \) analysis by Ullman [Ullm 73], have not been shown adapted to solving backward problems.

2.4.2 Or Versus And Formulations

Data flow problems can be classified according to the type of information desired. All problems fall into one of the following two classes:

1. And, Set Intersection, or Must - This class of problem attempts to discover what information must reach a certain point in the program. The solution technique begins with the assumption that all information reaches a given node and lets the propagation process delete information as it is found to be false.
2. Or, Set Union, or May - This class of problem attempts to discover what information may reach a certain point in the program. The solution technique begins with the assumption that no information reaches a given node, and lets the propagation process add information as it is found to be true.

The \( \text{set intersection} \) class can be transformed into \( \text{set union} \) problems through the use of DeMorgan's laws.

2.4.3 Clustered Versus Single Bit

Problems can be classified as to the amount of information that is found.

1. Single Bit Problems - ask for a specific piece of information about a variable. For example, whether it is live or dead at each block in the program. Since only a single attribute is being propagated, only a single bit is required for each node to store the information.
2. Clustered Formulation - asks for compound information about a variable. For example, which definition points for a variable can effect the value of that variable at each block in the program.

There are several common data flow problems that are clustered and belong to the \( \text{set union} \) class of problems. It is difficult to imagine a useful \( \text{set intersection} \) problem that is clustered.

In this paper, an algorithm is presented to directly solve clustered problems. This algorithm produces a solution to this class of problems in time that is linear in the size of the flow graph. These algorithms do not require flow graph reducibility. A similar technique, capable of solving only the single-bit problems, has been presented by Kou [Kou 77] and Babich and Jazayeri [Baja 78].

2.4.4 Rapid and Fast Problems

Graham and Wegman [GrWe 76] characterize a data flow problem as being fast if the problem satisfies the following criterion in addition to the admissible functions presented in section 2.2:

\[ (\forall f \in F)(\forall x \in L)[f(f(x)) \geq f(x) \wedge n] \]

They present an algorithm to efficiently solve problems of this class in \textit{almost} linear time.

Kam and Ullman [KaUl 76] characterize a data flow problem as being \textit{rapid} if the problem satisfies the following criterion in addition to the admissible functions:

\[ (\forall f, g \in F)(\forall x \in L)[f(g(x)) \geq g(1) \wedge n f(x) \wedge n x] \]

Kam and Ullman's problems can be solved efficiently by the iterative algorithm.

The \textit{fast} problems are a subset of \textit{rapid} problems.
3.0 The Global Algorithm

The technique presented here does not solve all GDFAPs. It does solve a limited subset of problems in time linear in the size of the problem, for general flow graphs. Examples of some common data flow problems are given in section 4.0. The data flow problems that can be solved by this technique are characterized as follows:

Let \( L \) be a semilattice for a data flow problem, and let \( M \) be a function space over \( L \). We say that \((L,M)\) is \textit{cluster partitionable} if:

1. There are three types of sites that can exist within each node.
   - **B sites** ordering of statements within the block.
   - **E sites** These sites \textit{end} information chains. Information is propagated to nodes that contain \( E \) sites. All nodes that contain \( E \) sites are called \( E \) nodes.
   - **S sites** These sites \textit{stop} the propagation of information chains.

   - **B site**
   - **S site**
   - **E site**

All nodes that do not contain any \( B \) sites, \( E \) sites, or \( S \) sites are \( P \) nodes. \( P \) nodes propagate information without changing it.

In some problems, the nodes containing \( B \) sites may also contain \( S \) sites. When this occurs, the \( B \) site is defined to be logically before the \( S \) site. In some problems, the \( E \) site may also be \( S \) site. When this occurs, the \( E \) site is defined to be logically after the \( S \) site.

The Algorithm:

1. Produce a graph \( G_1 \) from \( G(N,E) \). In \( G_1 \), every node \( N \) in \( G \) that contains an \( S \) site is split into two nodes. The first new replacing \( N \) contains all information that occurs before the first \( S \) site for node \( N \). It becomes the target of all in-edges that entered node \( N \). The second new node replacing \( N \) contains all information that occurs after the last \( S \) site for node \( N \) in \( G \). This new node becomes the source of all out-edges that exited node \( N \).

2. Within a block, all \( E \) sites that occur before the first \( S \) site, all \( B \) sites that occur before the last \( S \) site, and all \( B \) sites and \( E \) sites that occur after the first \( S \) site but before the last \( S \) site are to be ignored. The original flow graph is shown in Figure 2. Figure 3 shows the graph with nodes split at the \( S \) sites.

---

3.1 First Version

This algorithm must be executed once for each cluster \( v \).

This data flow technique differs from other data flow techniques in that information chains can be followed through the \( S \) sites of each cluster. Within a block, all \( B \) sites except the \( B \) site that occurs after the last \( S \) site are represented by \( B \) nodes. Each \( B \) node is replaced by \( n B \) nodes. Each \( B \) node becomes the target of all in-edges that entered node \( N \). Each \( B \) node contains all information that occurs before the first \( S \) site for node \( N \). The second new node replacing \( N \) contains all information that occurs after the last \( S \) site for node \( N \) in \( G \). This new node becomes the source of all out-edges that exited node \( N \).

In data flow graphs, the nodes typically correspond to the basic blocks of the program. These basic blocks contain a group of statements. The first step of the algorithm addresses problems that result from the ordering of statements within the block.

There are three types of sites that can exist within each node. Figure 1 indicates how these sites are presented in the following figures.

- **B sites** These sites begin information chains. Information is propagated from nodes that contain \( B \) sites. All nodes that contain \( B \) sites are called \( B \) nodes.
- **S site**
- **E site**

---

1 In this paper, the root of the flow graph is dealt with in a cavalier manner. Real flow graphs from programs are generally rooted. The root is typically the entry node to the routine. The algorithms presented in this paper will propagate the information to the entire graph regardless of the reachability of any node from the root. In data flow problems where this behavior is critical, a separate pass over the graph should be made to produce a new graph that is rooted.

2 These sites to be ignored will only affect chains that are local to a single basic block. These chains can be built and maintained by algorithms that solve the local data flow problem. This simplification allows the asymptotic complexity of this algorithm to be based on the size of the node set and edge set.
continuous series of events, we can model the process of building chains as a series of function calls as we pass through a node. In the area of the node that contains no B sites, E sites, or S sites the function is the identity function:

\[ f(v) = v \]

B sites add information to what is being propagated through the node. Whenever a B site is encountered, the function is:

\[ f(v) = v \land c \]

S sites inhibit any information from being propagated through themselves. Whenever an S site is encountered, the function is:

\[ f(v) = c \]

The propagation function for a node without any S sites is:

Let \( b \) be the set of B sites in node \( n \).

\[ f(v) = \bigcap_{i \in b} c_i \]

The propagation function for the first new node formed from a node with an S site is:

Let \( b \) be the set of B sites in node \( n \) that occur before the first S site in node \( n \).

\[ f(v) = \bigcap_{i \in b} c_i \]

The propagation function for the second new node formed from a node with an S site is:

\[ f(v) = v \]

This step takes no more than \( O(N) \) time and can at most double the size of the graph. The visitation order of nodes is irrelevant.

Figure 3. Location of Strongly Connected Regions

2. Assign a unique number to each B site of \( v \). This number will be needed in steps (3) and (4). This step can be done in \( O(N) \) time since the graph can be traversed in any convenient order. These are the numbers shown in Figure 3.

3. Produce the graph \( G_2 \) from \( G_1 \) so that \( G_2 \) is a directed acyclic graph. The graph, \( G_2 \) is a directed graph. This can be accomplished by collapsing each cycle or strongly connected component into a single node. These new single nodes of \( G_2 \) contain all of the in-arcs and out-arcs of each of the members of the cycle in \( G_1 \). The box in Figure 3 identifies the strongly connected component. Figure 4 show the graph after that region has been collapsed to a single node.

![Figure 4. Reduction of Strongly Connected Regions](image)

The numbering of the B sites is used here so that each B site can be sorted out later. When a region is collapsed, all B sites in that region are inherited by the new node. The numbering still allows each B site to remain unique.

An algorithm developed by Tarjan [Tarj 72] finds the strongly connected components of a directed graph in \( O(|N| + |E|) \) time. A version of this algorithm can also be found as Algorithm 5.4 on pages 189-195 of The Design and Analysis of Computer Algorithms [AHOU 74] by Aho, Hopcroft and Ullman. This latter version has been modified slightly for use in the next section.

Step 3 is motivated by the observation that every B site reachable from any node in a cycle is reachable from every node in the cycle since execution may proceed around cycles in the graph an arbitrary number of times. No cycle contains an S site since all S sites were split in step (1).

The function at the new node is:

\[ f(v) = \bigcap_{i \in c_i} c_i \land v \]

This step is the key to the whole algorithm. It sets this method apart from all other methods for solving data flow problems. This step only works because we are solving a single cluster and can therefore modify the edges of the graph in a manner that is dependent on the content of the nodes of the graph.

4. Traverse the graph \( G_2 \) in depth first search order. Begin the depth first search at the E sites. Do not visit any node more than once. (Use a simple Visited field at each node to keep track of this.) Propagate the information by doing the meet operation as the depth first search returns. The nodes will be visited in postorder.\(^3\) This visitation scheme imposes the same ordering as topological sorting.

Figure 5 shows the B sites reachable from each node after the propagation phase. The propagated data is shown in italic numbers.

\(^3\) Visit all sons of \( N \) before visiting \( N \).
The function executed at each node is:

Let \( s \) be the set of immediate successors of node \( n \).

\[
V_n = (\bigcap_{i \in s} f_i(x_i)) \cap f_n(x_n)
\]

This operation takes time \( O((N + E) \times \text{information evaluation time}) \).

5. Map the information back into the original flow graph. The bit vectors of all nodes in a collapsed region are all the same. This is done in Figure 6.

None of the above steps requires more than \( O((N + E) \times \text{information evaluation time}) \). The entire algorithm must be executed once for each cluster. This makes the overall worst case complexity of \( O((N + E) \times \text{information evaluation time}) \).

3.2 Second Version

The second presentation will demonstrate the algorithm at a more concrete level. In this version, several of the steps of the previous presentation have been merged.

This version deals only with a subset of the program graph. The calculation of which nodes are in the subset is presented in section 6.0. For the purposes of the presentation here, it should be assumed that the entire graph is being processed.

The \( \text{SetInfo} \) routine is the calling routine for \( \text{SearchC} \) and \( \text{SearchD} \). This routine has seven parameters. These parameters define the data flow problem and the area of the graph to be visited.

- \( \text{NodeList} \) - The list of nodes that may be searched.
- \( \text{EdgeList} \) - The set of edges that connect the nodes. This list may be either the \text{Successor} set or the \text{Predecessor} set depending on whether the problem is a \text{backward problem} or a \text{forward problem}.
- \( B \) - The set of nodes that contain \( B \) \text{sites} for the cluster.
- \( E \) - The set of nodes that contain \( E \) \text{sites} for the cluster.
- \( S \) - The set of nodes that contain \( S \) \text{sites} for the cluster.
- \( \text{Info} \) - The information to be updated by this call.

Two data structures used in \( \text{SetInfo} \) must be explained here. For an explanation of \( \text{DFNumber} \) and \( \text{LowLink} \), see pages 189-195 of [Ahu74].

- \( \text{Visited} \) - An instance of this variable is required for each node in the graph. The variable can have one of four possible values. The \text{outside} value is used to indicate that the node is outside of the area that is to be addressed by this call. All nodes within the affected area are initialized to \text{new}. As the nodes are visited by the algorithm to reduce the strongly connected regions they are placed on a stack and given the \text{onstack} value. As the nodes are popped off of the stack, they are given the value \text{old} so that they will not be visited again.

- \( \text{MapToNew} \) - This structure maps the node indices from the original program flow graph to the new directed acyclic graph.

\begin{verbatim}
{ Global Version of Partitioned Data Flow Analysis }
procedure SetInfo(NodeList, EdgeList, B, E, S, Info) {
    Output - Info }
    Count = 1.
    empty Stack
    Visited[*] = outside
    NewEdgeList[*] = o

    for each node in NodeList do
        Visited[node] = new
    endfor

    for each node in E \ NodeList do
        NewEdgeList[node] = EdgeList[node]
        for each n in EdgeList[node] do
            if ( Visited[n] = new )
                call SearchC(n)
            endif
        endfor
    endfor

    NewVisited[*] = outside
    for each node in NodeList do
        NewVisited[MapToNew[node]] = new
    endfor

    { Do the depth first search }
    for each node in E \ NodeList do
        call SearchD(MapToNew[node])
    endfor

    { Copy the newly calculated Info back. }
    for each node in NodeList do
        Info[node] = NewInfo[MapToNew[node]]
    endfor

    end

Figure 7. SetInfo
\end{verbatim}
4 Parameters to subroutines must be put into both the count and the numbering of B sites is done at some other level. 

procedure SearchC( node )

DFNumber[ node ] = Count
Count = Count + 1
LowLink[ node ] = DFNumber[ node ]
push node on Stack
Visited[ node ] = onstack

The following information is required as input to the data flow analysis:

4.1 Input Structures

Def[v] is a list of nodes for variable v, whose values are changed by the execution of this block. This includes targets of assignment statements and parameters to input statements.

Mod[v] is a list of nodes for variable v, whose values may be changed by the execution of this block. Assignment into single values of arrays and variables passed to subroutines are two examples of statements that can generate Mod.

UnDef[v] is a list of nodes for variable v, whose values become undefined by statements in the node x. Loop index variables in languages such as BCPL and Fortran become undefined in the exit node from the loop. Also, all local variables are initially undefined on entry to the subroutine and become undefined on exit from the subroutine.

Use[v] is a list of nodes for variable v, that are referenced by statements in node x. This set is typically made from all variables that appear in any expression, in parameters to subroutines, and in output statements.

4.1.2 Output Structures

The following information is produced by the algorithms discussed here.

Def[v,x] contains the list of definition sites for variable v, that are reachable from node x. This set is calculated top-down and is the output of the Reaching Definitions problem.

Mod[v,x] contains the list of modification sites for variable v, that are reachable from node x. This set is calculated top-down and is the output of the Reaching Definitions problem.

UnDef[v,x] contains the list of undefined sites for variable v, that reach node x. This set is calculated top-down.
Intuitively, this problem cannot be solved by the PVT since information of each path must be incremented by 1 to accumulate the path length. The formulation of this problem does not satisfy the definition of PVT algorithm is powerful enough to solve all problems in these classes. While it is true that the PVT algorithm can solve most of the problems that are useful in many areas.

4.2 The Data Flow Problems

1. Reaching Definitions - Determine the set, Def, for each node, x, of variable definitions that can enter node x. The term variable definition means a statement that can potentially modify the value of a variable. This is a forward problem and the output is typically used to build Use-Definition Chains which are used in compiler optimization [Hech 77].

2. Live Variables - Determine the set, Uses, for each node, x, of variables that are live or may be used after control passes out of node x. This problem is the backward analogue of the Reaching Definitions Problem and its output can also be used to build Use-Definition Chains [Hech 77].

In addition to being used in compiler optimization, the information produced by a solution of this problem can be directly used to discover if the output of a definition site is ever used. This has applicability in the area of software reliability since unused definitions may be the result of programming errors.

3. Unsafe Uses - Determine the set, UnDefs, for each node x of variable undefinitions that can reach node x. A use of a variable is unsafe, for each node x, if there exists a path that the program may take to node x, where the variable has not been defined. By characterizing the problem in this manner, it is not only possible to determine if the use is safe, but the location of the kill site is determined if the use is unsafe. This problem is associated with reliability of software [FoOs 76].

4. Value Printing Problem - This problem is broken into two subproblems, the Forward Printing Problem and the Backward Printing Problem. The output of these two problems is then processed to find the list of nodes that a particular value of a single variable is live within. These two problems were defined by Fosdick and Osterweil in the DAVE system. To date their only use has been in the area of reliability [FoOs 76].

These two problems have never been formulated for solution by a classical technique. The technique used by Fosdick and Osterweil is similar to the technique proposed by K. This technique was used because the number of Use-Def pairs that were needed was small compared to the number of possible Use-Def pairs in a given program.

In the Forward Printing Problem, a list of nodes for each Def, Mod or UnDef in the program is created. The list contains each node that is reachable from each beginning node without passing through another Def, Mod or UnDef.

In the Backward Printing Problem, a list of nodes for each Use in the program is created. The list contains each node that is reachable from each Use node without passing through another Def, Mod or UnDef.

4.3 Problems not Solvable by PVT

As explained in section 3.0, the PVT algorithm works by dividing the problem into small clusters that are solved separately. The key property of a cluster is that each strongly connected component of a cluster has the same information available for each node of that component. This allows the strongly connected regions to be reduced to single nodes, transforming the program flow graph into a directed acyclic graph. A data flow problem that is amenable to this technique is called a partitionable problem.

While it is true that the PVT algorithm can solve most of the clustered problems that are rapid, it should not be inferred by this that the PVT algorithm is powerful enough to solve all problems in these classes. The following problem is rapid but cannot be solved by PVT. The problem is broken into two subproblems, the Forward Printing Problem and the Backward Printing Problem. The output of these two problems is then processed to find the list of nodes that a particular value of a single variable is live within. These two problems were defined by Fosdick and Osterweil in the DAVE system. To date their only use has been in the area of reliability [FoOs 76].

These two problems have never been formulated for solution by a classical technique. The technique used by Fosdick and Osterweil is similar to the technique proposed by K. This technique was used because the number of Use-Def pairs that were needed was small compared to the number of possible Use-Def pairs in a given program.

In the Forward Printing Problem, a list of nodes for each Def, Mod or UnDef in the program is created. The list contains each node that is reachable from each beginning node without passing through another Def, Mod or UnDef.

In the Backward Printing Problem, a list of nodes for each Use in the program is created. The list contains each node that is reachable from each Use node without passing through another Def, Mod or UnDef.

Suppose you wish to find the length of the shortest path from each node in a flow graph to every other node in the data flow graph. The formulation of this problem is as follows:

$\text{n} = \min \text{ for each element in the cluster.}$

There are no P sites or S sites. The operation at each B site is a meet with the constant vector:

$[\infty, \infty, \ldots, 0, \ldots, \infty]$

where the number in the i-th position for node i is 0.

The formulation of this problem does not satisfy the definition of PVT since an additional operation must be done at the P nodes. The value of each path must be incremented by 1 to accumulate the path length. Intuitively, this problem cannot be solved by the PVT since information is lost when the strongly connected regions are collapsed and this information, the length of the shortest path, is utilized by this problem. This problem can be solved by the techniques of Kam and Ullman or Graham and Wegman.

As a second example consider a common data flow problem that is not distributive such as constant propagation. This problem is described by Hecht [Hech 77] as:

Try to discover at the top of each node in the flow graph, those variables that are assigned the same constant value on all paths from the initial node to the top of a given node.

This problem fails to be fast or rapid since it has the property that new information may be acquired each time propagation proceeds around any loop. Additionally, this problem has another characteristic that cannot be handled by PVT, namely, it fails to be partitionable into clusters. In the constant propagation problem, whenever a constant is propagated to a new node, that propagation may make other constant propagations possible. There is no way to make the clusters independent of each other.

While there are many problems that the PVT cannot solve, the PVT does provide a mechanism to efficiently solve a rich set of data flow problems that are useful in many areas.

4.4 The Incremental Environment

The primary motivation for this work has been to provide an algorithm that is suitable to produce and maintain data flow information while the program is being edited by the programmer. Unlike the batch compiling environment in which a many changes to the input program are made...
between compilers, in the incremental environment, the granularity of change is small. The supposition is that most operations done by the programmer are trivial modifications to single statements in the program. It is expected that the affected area of these changes is also fairly small in most cases. It is therefore justified to use algorithms that can exploit this locality.

The algorithms presented in the next section behave in a manner that is uniquely suited to this environment. They can be used in a structured program editor to provide the programmer with correctly updated data flow information as the program is being modified. This analysis produces information that relates to the correctness of the program.

The idea of doing this analysis is not new. The Dave system [FoOs 75] by Fosdick and Osterweil produce such information but in a batch environment. The Scope system [Masl 80] produces a less precise form of information and it is driven by specific programmer queries only.

6.0 Local Algorithms

The purpose of this section is to provide algorithms to maintain the data flow information in the presence of incremental changes to the program. These algorithms also perform the analysis required to detect many anomalies in the program.

The algorithms presented here are intended to respond to changes that fall into three broad categories:

1. Type I operations are changes to individual references in the program.
2. Type II operations are changes that add or delete edges (control flow paths) to the program.
3. Type III operations are changes that add or delete basic blocks to the program.

Two data flow problems discussed in section 4.0, the Live Variable Problem and the Unsafe Uses Problem will be used as examples throughout this section.

Several sets are maintained by the incremental algorithms defined in this section. These sets are Use, Def, UnDef, and Mod defined in section 4.1.1 and Users, Defs, UnDefs, and Mods defined in section 4.1.2.

6.1 Special Local Algorithms

The following procedure, SearchL, is used to find the area affected by a single change in either the forward or backward direction. SearchL calculates a list of nodes that are affected by a change at a particular section of the graph. This list is then used as input by the higher level algorithms. These higher level algorithms do specific operations at each of the nodes.

This procedure begins the search at some node specified at a higher level. This node is typically the one that is currently being edited by the user, and is identified as being pointed to by the cursor. SearchL only pursues a single value. The procedure has five parameters:

NodeList The list of nodes that may be searched.
EdgeList The set of edges that connect the nodes. This list may be either the Successor set for backward problems or the Predecessor set for forward problems.
S The set of nodes that contain S sites for the cluster.
c The name of the variable whose use or definition has been modified by the user.
Visited The list of nodes that have been visited.

The worst case complexity of SearchL is O(NodeList + EdgeList). The use of Visited assures that no node is visited more than once. The Visited field of each node must be checked once for each in-edge. These algorithms only modify the graph for a single value of a single variable.

The expected case complexity should be much better than the number of nodes in the graph. SearchL visits only those nodes that a single value may flow through. While it is true that sometimes the entire graph must be visited, many values are short-lived.

The term value means the result of a single definition point of a single variable.
6.2.1 Addition of a Def

There is a problem associated with adding a Def to a block. When a Def is added, it may break some or all use-definition chains for that variable that pass through that block. There are no data structures in this algorithm for encoding the various paths that may be available around the Def that is to be added. Therefore, SetInfo must be called to fix up the chains. The algorithm that is used to add a Def works by deleting all old chains that pass through the node, and then rebuilding the chains. The chains are rebuilt by propagating the chains that border the affected area into the affected area.

```
procedure AddDef(c, v)

  { Add the Def to the current node. }
  Def[v] = Def[v] + c

  { Delete all the backwards Uses for v that pass through c. }
  call SearchL(G, Pred, Def[v] U UnDef[v], c, Visited)

  for each node in Visited do
  endfor

  { Rebuild the Uses. }
  B = Visited \ (Use[v])
  E = Visited \ (Def[v] U UnDef[v] U Mod[v])
  S = Visited \ (Def[v] U UnDef[v])
  call SetInfo(Visited, Succ, Defs, UnDefs, Mods)

  Defs[c, v] = Defs[c, v] + c

  for each node in Defs[c, v] do
    if (Uses[node, v] = 0)
      then print
        "The value calculated at node v is never used."
      end if
  endfor

  { Delete all the forward Defs for v that pass through c. }
  call SearchL(G, Succ, Defs[v] U UnDef[v], c, Visited)

  for each node in Visited do
  endfor

  { Rebuild the Defs, UnDefs, and Mods. }
  B = Visited \ (Def[v] U UnDef[v] U Mod[v])
  E = Visited \ (Use[v])
  S = Visited \ (Def[v] U UnDef[v])
  call SetInfo(Visited, Pred, B, E, S, Defs U UnDefs U Mods)
end
```

The algorithm must ensure that all of the Def sites that can reach c still have valid uses. This same test must also be made for the newly added Def.

6.3 Algorithms For Type II Changes

Type II changes are all control flow changes. These changes occur as changes in the edges in the program flow graph. Type II changes are the result of operations on goto statements and statements that implement restricted branches.

- Type II algorithms look at all variables that can have live values in the region changed.
- Type II algorithms do not look at all values of the variable. The algorithms only deal with those values that either affect or are affected by the part of the program pointed to by the editing cursor.
- Type II algorithms will only track these values through part of the program. This part of the program includes only the area in which a single computed value is live.

Type II algorithms are invoked by specifying pointers to two basic blocks. These parameters point to the source and destination of the edge being manipulated.

- `cin` is a pointer to the source node that is affected by editing operations where more than one node is involved. Operations that make arbitrary changes in the flow graph must be specified by a source node `cin` and a destination node `cout`.
- `cout` is a pointer to the destination node that is affected by editing operations where more than one node is involved.

Algorithms are given for two general problems:

1. Addition of an arbitrary edge.
2. Deletion of an arbitrary edge.

6.3.1 Addition of an Edge

The routine `SearchL` is not used when adding an edge to a graph. `SearchL` has been merged into `AddEdge` in order to allow additional criteria for selecting the visitation order. These additional criteria keep subtrees from being visited where the information is already correct. These already correct subtrees are the result of other paths through the tree that carry the same information as the newly added edge. The criterion applied is based on the observation that propagations need not be done if the information to be propagated is a subset of the information already there.

Once the edge is added to the `EdgeList`, the values that are propagated forward, `Def`, `UnDef`, and `Mod` are available at the source node, `cin`, and must be propagated to all nodes that are reachable from the destination node, `cout`.

---

8 The term user queries is a generic term which encompasses all operations where the user wishes to inquire about information that is maintained by the system. In these operations, the information that is being maintained by the system is made available to the user. Examples of these queries are:

- Print all the locations that can use this definition for v.
- Print all the locations that can define the value for v that is used here.

These queries are not of the type that can be determined by inspection of the symbol table since they require knowledge of the flow of values for each variable. These queries can be very important to the user. The output of these queries can provide information that can aid the user assessing the effects of small changes in the program. The Scope system by Masinter [Masin 80] provides a similar feature for a Lisp based system.
6.3.2 Deletion of an Edge

The same feature that makes for a faster algorithm when adding an edge causes a certain amount of difficulty when deleting an edge. Because it is unknown from which path a value comes, a conservative strategy must be used to update the information caused by deleting an edge. The strategy used is to remove all information that is propagated from the deleted edge, and then to rebuild the information in the affected area by propagating the information from the in-edges of that affected region. The errors that must be detected stem from Defs that do not have any Uses after the edge has been deleted.

The DelEdge algorithm is the only algorithm that is not linear in the size of the affected area. Every other algorithm presented in this paper has the property that no node is visited unless the information at that node is changed in some matter. This algorithm visits extra nodes in some cases.

An interesting point to note is that it is not necessary to check that each Use has a Def site that reaches it. This is true since the three sets, Defs, UnDefs, and Mods are propagated in the forward direction and cover all paths. As long as the Use site is reachable, in the standard graph theory sense, there must either be a path that contains a Def or Mod which is acceptable, or the path is reachable by an UnDef, in which case the path had already been detected as a possible error path.

6.4 Algorithms for Type III Changes

Type III changes are modifications that add or delete nodes from the program flow graph. Type III changes differ from Type II changes in that Type III changes always modify the number of basic blocks in the program while Type II changes only deal with interconnection changes among the basic blocks.

Type III operations occur when:

1. Addition of a goto statement into or from the interior of a basic block. That basic block must be serially split into two blocks at the point where the new edge enters.
2. Deletion of a goto statement from the head of a basic block. If there were only two in-edges into that block and the remaining in-edge is the only out-edge from another block, the two blocks can be serially combined.
3. Addition or deletion of a section of code. Code is added to the program by asking the editor to add an empty template at the cursor position. In these operations, the user fills in the body of the template. Type I and Type II operations are used to fill in these templates.

The operation of adding the template is implemented by adding a series of empty basic blocks in the area pointed to by the editing cursor. This operation of adding the templates can be implemented by three primitive operations: serial splitting, parallel splitting, and addition of self-loop.

There are three operations over basic blocks that are used to grow a program. These operations also have inverse operations that are used to shrink a program. Unlike other operations that have been considered here, the process of growing the flow graph is a perfect inverse of the process of shrinking the flow graph.
The operations considered in this section are:

1. Serial splitting of a basic block.
2. Combining two serial basic blocks.
3. Parallel splitting of a basic block.
4. Combining two parallel basic blocks.

### 6.4.1 Serial Splitting of a Block

The term *serial splitting* means to split a single block where the resultant blocks are connected such that after control leaves the first resultant block, it will enter the second resultant block. When a block is split, the first few statements of the split block go into the first resultant block and the remaining statements go into the second block. If the assumption is made that the first resultant block retains the label of the old block, then all that must be done in the splitting operation is to relabel the *User*, *Defs*, * Mods*, and *UnDefs* that are propagated from the resultant block.  

The only work that is done in either serial splitting or combining of basic blocks is to renumber the information in the basic blocks that are reachable for the block being changed. There are no error conditions to be checked.

```plaintext
procedure SerSplit(OldBlock, NewBlock, SplitPtr)
    call SerSplitSet(Use, UnDef, Pred)
    call SerSplitSet(Def, Defs, Succ)
    call SerSplitSet(Mod, Mods, Succ)
    call SerSplitSet(UnDef, UnDefs, Succ)

    procedure SerSplitSet(Use, Uses, Edges)
        for each v in SymbolTable do
            if (OldBlock ∈ Use[v] and (Loc(v) > SplitPtr))
                then do
                    Sets(v) = Sets(v) + OldBlock + NewBlock
                enddo
            enddo
        enddo

    procedure SerSplitSet(Def, Defs, Edges)
        for each v in SymbolTable do
            if (OldBlock ∈ Defs[v] or Loc(v) > SplitPtr)
                then do
                    Sets(v) = Sets(v) - OldBlock + NewBlock
                enddo
            enddo
        enddo

    procedure SerSplitSet(Mod, Mods, Edges)
        for each v in SymbolTable do
            if (OldBlock ∈ Mods[v] or Loc(v) > SplitPtr)
                then do
                    Sets(v) = Sets(v) - OldBlock + NewBlock
                enddo
            enddo
        enddo

    procedure SerSplitSet(UnDef, UnDefs, Edges)
        for each v in SymbolTable do
            if (OldBlock ∈ UnDefs[v] or Loc(v) > SplitPtr)
                then do
                    Sets(v) = Sets(v) - OldBlock + NewBlock
                enddo
            enddo
        enddo
enddo
```

**Figure 15. SerSplit**

### 6.4.2 Remaining Operations

The operations to parallel split and parallel combine basic blocks, and the operations to add and delete self-loop blocks are null operations with respect to the data flow information. This is a result of the restriction that all blocks acted on in this manner must be empty. While this restriction may seem unreasonable at first, it must be realized that these operations only result from the addition or deletion of templates from the program. Templates must be empty. The only operation that must be done is to allocate or free a new basic block. The implementation of this is outside the scope of this presentation.

The algorithms that could be derived for more of these operations become equivalent to doing a series of *Type I* operations to delete the information from one block, followed by a series of *Type I* operations to reinsert the information in the other block. The purpose of this presentation is to describe a series of primitive operations out of which any high level operation can be built, rather than to provide concrete algorithms for all possible situations.

### 7.0 Conclusions

In this paper, a new algorithm has been presented to perform data flow analysis over a limited set of problems. The algorithm is capable of building *Use-Definition Chains* and solving many similar problems. A precise characterization of problems solved has also been demonstrated. The asymptotic complexity of the global algorithm is linear in the number of edges in the program flow graph.

An incremental version of this algorithm has also been given. The suitability of this algorithm to detect many common programming errors in a high level program editor has also been explored.

The complexity of the incremental algorithms fall into two categories. In one category, the algorithm must visit only those nodes where the data flow solution must actually change. In the second category, a slightly larger number of nodes must be visited, but this number should be less than the number of nodes in the graph. In both categories, each node must be visited a fixed number of times for each in-edge, thereby retaining the linear complexity of the algorithm.

Unlike many other algorithms for data flow analysis, this algorithm, in both the incremental and global versions, is unaffected by any complexity in structure in the program flow graph.

### 7.1 Other Work Using the Single Variable Paradigm

This work is not the first to suggest the single variable paradigm. Kou [Kou 77] first gave an algorithm for solving a simple version of the Live Variable problem. Babich and Jazayeri [BaJa 78] give an incremental version of this algorithm that is useful on parse trees. Neither of these algorithms is capable of building the use-def chains.

### 7.2 Other Incremental Algorithms for GADFPs

Ryder, in her PhD. dissertation [Ryde 82], has made an attempt to solve the incremental update version of the global data flow problem. She has modified several classical data flow algorithms to work in an incremental fashion. The data flow algorithms specifically addressed in her thesis are: Allen-Cocke Interval Analysis, Hecht-Ullman T1-T2 Analysis, and Tarjan Interval Analysis. It is implied that the methods used could be easily extended to other data flow techniques.

The work that Ryder has done is limited to a subset of the changes required in many environments. Only *Type I* changes, i.e. local, single variable changes are considered. There are no provisions for dealing with any changes to the structure of the program flow graph. This places severe restrictions on the environments in which this work can be used.

In the area of programming environments, it is required that the user be able to make changes that affect the program structure and still have the reliability information produced quickly.

There is no implication that this work could not be extended to handle *Type II* and *Type III* changes. This was simply outside the scope of the topics presented by Ryder.

---

9 A better version of this program would pick the resultant block with the smallest number of program references. This would result in a slight, nonasymptotic, speedup of this operation.
Bibliography


[Bala 78] W. A. Babich and Mehdi Jazayeri. The method of attributes for data flow analysis (two parts). Acta Informatica, 10:253-272. See also Tr-77-007, Department of Computer Science, University of North Carolina.


