ELIMINATION ALGORITHMS FOR DATA FLOW ANALYSIS

B.G. Ryder and M.C. Paull

DCS-TR-140

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus, Rutgers University
New Brunswick, NJ 08903
ELIMINATION ALGORITHMS FOR DATA FLOW ANALYSIS

Barbara G. Ryder
Marvin C. Paull

ABSTRACT
A data flow algorithm is one that gathers information about the definition and use of data in a program or a set of programs. A unified model of a family of data flow algorithms, called elimination methods, is presented. The algorithms are characterized by the manner in which they solve the systems of equations that describe data flow problems of interest. These implementation-independent descriptions of the algorithms facilitate comparisons among them and illustrate the sources of improvement in worst-case complexity bounds. This tutorial is valuable as a study in algorithm design; it presents a new view of these algorithms and their interrelations.

1. INTRODUCTION
Historically, compile-time analysis of programs was developed to enable the optimization of compiler-generated code. Compile-time analysis of programs includes control and data flow analyses. Control flow analysis traces the patterns of possible execution paths in a program; data flow analysis traces the possible definition and use of data in the program. In optimization, the information gathered by these analyses is used to transform the program to a semantically equivalent one which executes faster and/or uses less space.

Today, optimization of compiled code is probably the most important use of data flow information. The flexible constructs in modern programming languages necessitate data flow analysis for efficient translation; for example, in a language with late bindings, data flow information can enable the substitution of a compile-time check for an execution-time check. If the type of a variable is constrained to be consistent with its use, data flow

\footnote{\textit{1}\textsuperscript{Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903}}

\footnote{\textit{2}\textsuperscript{Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903}}
information can enable us to ascertain the type of the variable.

Data flow information is also used in many non-compiling applications. Source-to-source transformation systems perform sequences of transformations on programs that take one high-level description of an algorithm into another, the latter optimized for execution. The source-to-source system cannot apply a transformation without insuring that the meaning of the algorithm is maintained: data flow information insures that transformations preserve meaning.

Software tools in interactive programming environments make data flow information available to programmers. The ability to see all the definitions or uses of a variable facilitates design, debugging, maintenance and documentation of code. Interprocedural data flow analysis, which traces data definition and usage across procedure boundaries, is especially suited to this application [Banning 79, Barth 78, Burke 84, Cooper 84]. Interprocedural analysis uncovers all the side effects of a procedure call. Intricate, inadequately documented code is difficult to maintain; interprocedural tools can help enormously [Ryder 74, Ryder 85].

In complex software, a small change in the program is expected to have localized effects, and therefore to produce a small change in the data flow information. An incremental update algorithm for data flow analysis need modify the original data flow solution only to reflect changes in a problem. Thus, an incremental update algorithm is usually more efficient than a complete re-analysis. Clearly, incremental updating has application to programming environments [Cooper 84, Zadeck 84]. The modelling work presented in this tutorial was part of the development of incremental update algorithms for data flow analysis [Ryder 82a, Ryder 83].

Today, there are two families of global data flow algorithms in use: the elimination and the iterative methods. The elimination methods include an original algorithm: Allen/Cocke interval analysis, and improvements on it: Hecht/Ullman T1-T2 analysis, Graham/Wegman analysis, Tarjan interval analysis. Our models of elimination methods describe how each algorithm solves the data flow equations that define useful data flow problems. The iterative methods, called workset, round robin and node listing, solve the data flow equations by initializing them to a safe value and then iterating to a fixed point solution. These methods, which we will not treat here, originated with Kildall's algorithm [Hecht
In the literature, all of these algorithms are described in terms of a specific implementation. It is difficult to see their similarities and differences. Our aim in this paper is to present the elimination algorithms in an implementation-independent manner, to better understand the main ideas each incorporates. To accomplish this we define the data flow problem by a system of equations and describe how each technique solves these equations [Cocke 70]. These models make apparent the similarities and differences of these algorithms, as well as where and why the complexity savings occur in each, a fact not clear from their implementation descriptions. In addition, these models show the algorithms to be general solution procedures applicable to certain systems of equations.

The following is an informal introduction to the data flow analysis algorithms presented; in Section 2 we define these ideas formally. The example in Figures 1 through 5 gives an intuitive idea of problem formulation. In the program fragment in Figure 1 the question we want answered is "Can execution reach statement L with y never having been assigned a value?" To ascertain this, we insert statement K (see Figure 2) and use our analysis to determine if it is possible for y to have the value 9999 at statement L. If so, then in the original fragment, the value of y may be undefined at statement L.

```
if x > z then y := a
else if z > 3 * w then y := b
L: q := 2 * y  /* can y be undefined here? */
```

Figure 1: Program Fragment

```
K: y := 9999
if x > z then y := a
else if z > 3 * w then y := b
L: q := 2 * y  /* can y=9999 here? */
```

Figure 2: Transformed Program Fragment

To analyze the program fragment of Figure 2 we transform it into the graphical representation of Figure 3. We use a directed graph which consists of a finite set of nodes N and a finite set of directed edges between them E=\{\langle i,j \rangle : i,j \in N\}. Since the edges are directed, \langle i,j \rangle \in E does not imply that \langle j,i \rangle \in E. If \langle i,j \rangle \in E then we call j an immediate successor of i and i an immediate predecessor of j. In using a directed graph to represent the program, the nodes N correspond to statements or expressions in the
program and the directed edges E represent all possible execution transfers from one statement or expression to another. We want to calculate the set of possible values for y at statement L. This is tantamount to considering the set of definitions of y (i.e., value-assigning statements for y) which can be propagated along paths containing no subsequent redefinitions of y. For example, definition (ii) of y at statement K can be propagated to node 6 along <1 2 4 6> but not along <1 2 3 6> because definition (ii) at node 3 blocks (ii).

![Graphical Representation of Figure 2](image)

Figure 3: Graphical Representation of Figure 2

We can formalize this notion of which values a variable may have at a node in the graph using equations. Let \( X_i \) be the set of definitions of a variable y which reach node i. Let \( pre_j \) be all definitions of y in the program, if y is not defined at node j, and the empty set otherwise. Let \( def_j \) be the set of definitions of y created at node j, if there are definitions of y at j, and the empty set otherwise. Then we can describe the relations among the \( X_i \) in our example by the system of equations in Figure 4.

\[
X_i = \emptyset : i = 1 = \text{source}
\]

\[
= \bigcup_{j \in \text{pred}(i)} \{\text{pre}_j \cap X_j \} \cup \text{def}_j : 2 \leq i \leq 6
\]

where \( \text{pred}(i) \) is the set of immediate predecessors of i.

Figure 4: Equations for Example of Figure 2

The intersection of \( pre_j \) and \( X_j \) either eliminates all definitions that reach node j if y is defined at j or keeps them all if y is not defined at j.
The solution to the system of equations in Figure 5 tells which definitions of y reach each node. Since the added definition of y (ii) reaches statement L (i.e., node 6) we know that y might be undefined at L during the execution of the original program fragment. In Figure 5 we use a simplified version of the Gaussian elimination-like procedure described in Section 2 to solve the equations. Given a variable whose solution is known, we simply substitute that solution for all occurrences of that variable in the system. Repetition of this substitution procedure solves the system of equations. Section 2 describes the data flow equations more formally, discusses the Gaussian elimination-like solution procedure, and presents a general formulation for the reaching definitions problem which we have used for illustration here.

\[ X_1 = \phi \]

\[ X_2 = \text{def}_1 = \{i\} \]

\[ X_3 = X_4 = (\text{pre}_2 \cap X_2) \cup \text{def}_2 \]
\[ = \{i, ii, iii\} \cap \{i\} \cup \phi \]
\[ = \{i\} \]

\[ X_5 = (\text{pre}_4 \cap X_4) \cup \text{def}_4 \]
\[ = \{i, ii, iii\} \cap \{i\} \cup \phi \]
\[ = \{i\} \]

\[ X_6 = (\text{pre}_3 \cap X_3) \cup (\text{pre}_4 \cap X_4) \cup (\text{pre}_5 \cap X_5) \cup \text{def}_3 \cup \text{def}_4 \cup \text{def}_5 \]
\[ = \phi \cap \{i\} \cup \{i, ii, iii\} \cap \{i\} \cup \phi \cap \{i\} \cup \{i\} \cup \phi \cup \{iii\} \]
\[ = \{i, ii, iii\} \]

Figure 5: Solution to Equations of Figure 4

The data flow solutions are obtained by solving a set of equations by methods analogous to the one illustrated in Figure 5. A straightforward Gaussian elimination-like method yields a solution with \(O(n^3)\) complexity. Each of the algorithms presented here is a refinement of this method. The Allen/Cocke algorithm defines a natural order on the equations that leads to a highly structured coefficient matrix. By ordered substitutions, a smaller system of equations is obtained; thereby the solution of the entire system is reduced to the solution of the smaller system. This process is repeated, yielding successively smaller systems of similarly structured equations, producing an \(O(n^2)\) solution [Allen 77]. The algorithms which improve on interval analysis detect common substitution sequences in the equations and utilize them to reduce the work to \(O(n \log n)\). The Hecht/Ullman algorithm does substitutions for individual terms in the equations in a
nondeterministic manner, but retains a record of them in a 2–3 tree, so as to calculate common substitution sequences only once [Ullman 73]. The Tarjan algorithm uses a constrained substitution order in which all variables on the right-hand side of an equation are substituted for at the same time; that is, a reduced equation for a variable is obtained by substituting for all dependent variables at once [Tarjan 74, Tarjan 81a]. A path compressed tree is used to remember the substitution sequences so as to eliminate duplicate calculations. The Graham/Wegman algorithm uses a different constrained substitution order for individual terms in the equations [Wegman 81]. This order takes advantage of common substitution sequences in the equations by delaying the substitutions for terms involving such sequences until the calculations corresponding to such sequences have been performed.

We present the models for each of the four elimination algorithms, stressing their key ideas. First, the basic concepts of data flow analysis and the Gaussian elimination-like solution procedure are presented. Second, we discuss Allen/Cocke interval analysis; we show its linear performance on a large class of flow graphs, which explains its popularity in practice. Third, we introduce Hecht/Ullman T1–T2 analysis, Tarjan interval analysis and Graham/Wegman analysis, contrasting them to the Allen/Cocke algorithm and to each other, discussing their differences in terms of how they solve the data flow equations. One example is shown worked by all four algorithms. Finally, we summarize our modelling efforts.

2. BASIC CONCEPTS

Data flow analysis is usually performed on some intermediate form of a program or a set of programs. We may use a control flow graph, a directed graph which describes the possible execution paths in a procedure [Hecht 77] or a parse tree representation of a procedure [Farrow 75, Kennedy 77]. To build the control flow graph of a program we partition its statements into basic blocks, maximal single-entry sequences which are exited only at their end [Backus 57]. Each basic block is represented by a node in the control flow graph. There is an edge (i,j) in the control flow graph if, during execution, control can transfer from basic block i to basic block j.\(^3\) Although each basic block has only one

\(^3\)We make the underlying assumption of all static analysis, namely we assume that all paths in the program are executable, since it is an undecidable problem to identify those which are not.
entry, it can have more than one predecessor. Data flow analysis can also be performed on a call graph, a directed graph which describes the possible calling relations between procedures in a software system [Allen 74, Ryder 79]. Each procedure in the system is represented by a node in the call graph. Each directed edge represents a possible procedure invocation. In interprocedural analysis, we trace data flow through the use of reference parameters and global variables [Banning 79, Berth 78, Burke 84, Cooper 84, Schwartz 79, Sharir 77].

The term flow graph will refer collectively to a control flow or call graph. Throughout this tutorial n refers to the number of nodes in the flow graph and e refers to the number of edges. The flow graph has a unique source node (source), which has no predecessors, and one or more exit nodes, each of which has no successors. Each node in the flow graph is associated with a function which describes how the code at the node affects data in the program. Data flow analysis algorithms gather this local information and infer the global data flow from it. The global information then can be specialized to provide data flow information for any node in the flow graph.

We will consider those data flow problems which can be defined by a system of equations $Q = \{Q_m\}_{m=1}^n$ and solved by a Gaussian elimination-like method [Issacson 66]. Each equation $Q_m$ in the system is associated with a node $m$ in the flow graph. We assume that the set of possible solution values, each an $n$-tuple $<z_1, \ldots, z_n>$ which satisfies the system $Q$, admits a partial ordering ($\leq$).

The four classical data flow problems, reaching definitions (REACH), live uses of variables, available expressions, and very busy expressions, all admit such a partial order; all can be formulated as in equation 1 [Hecht 77]. The data flow solutions of the classical problems are sufficient for most compiler optimizations (e.g., dead code elimination, constant propagation, common subexpressions elimination):

$$Q_m: z_m = \bigcup_{j \in S_m} \{ a_{m,j} \cdot n \cdot z_j \cup b_{m,j} \} \cup c_m$$  \hspace{1cm} (1)

where
• \( Z_m \) is the data flow solution either on entry to or on exit from node \( m \).

• \( \theta \) is intersection or union.

• \( a_{m,j} \), \( b_{m,j} \), \( c_m \) are constants derived from local data flow information (possibly null).

• \( S_m \subseteq \{ i \mid 1 \leq i \leq n \} \).

A substitution transformation of \( Q \), \( s(Q,m,j) \), for \( 1 \leq m,j \leq n \) is the result of substituting the right-hand side of \( Q_m \) for an occurrence of \( Z_m \) on the right of equation \( Q_j \), \( m \neq j \) and simplifying the resultant right-hand side of \( Q_j \). Then \( s(Q,m,j) \) differs from \( Q \) by having at most a different \( Q_j \) equation; all other equations are the same. It is clear that a solution of \( s(Q,m,j) \) is also a solution to \( Q \) and vice versa.

An equation \( Q_m \) has a loop breaking rule if there is another equation for \( Z_m \), called \( q_m \), such that

i. \( Z_m \) does not appear on the right-hand side of \( q_m \)

ii. every solution of \( q_m \) is also a solution of \( Q_m \)

iii. for every solution \( S \) of \( Q_m \) there is a solution \( s \) of \( q_m \) such that \( s \subseteq S \).

A set of equations \( Q \) is said to have a loop breaking rule if for each equation in \( Q \) initially there is a loop breaking rule, and for any equation in any set that can result from \( Q \) by a sequence of substitution transformations of \( Q \), there is also a loop breaking rule. A loop breaking transformation of \( Q \), \( b(Q,m) \), for \( 1 \leq m \leq n \) is the result of replacing \( Q_m \) by \( q_m \).

The Gaussian elimination-like solution procedure for these equations consists of applying a sequence of the substitution and loop breaking transformations; the procedure is shown in Figure 6. The complexity of this algorithm is \( O(n^3) \), assuming (as usually holds) that \( b \) and \( s \) are \( O(1) \). It can be shown that if a sequence of these transformations is applied to a system of equations \( Q \) producing the system \( R \), and \( \{ S_m \mid 1 \leq m \leq n \} \) is a solution to \( R \), then it is also a solution to \( Q \). Further, if \( \{ L_m \mid 1 \leq m \leq n \} \) is a solution to \( Q \), then there is a solution of \( R \), \( \{ K_m \mid 1 \leq m \leq n \} \), such that \( K_m \leq L_m \) for \( 1 \leq m \leq n \). If \( Q \) has a loop breaking rule, then the procedure in Figure 6 terminates and obtains the unique minimal solution (in terms of the partial ordering) [Pauli 85].
/* Elimination */
for i = 1 to n do
  begin
    Q ← b(Q,i)
    for j = i+1 to n do Q ← s(Q,i,j)
  end
/* Back Substitution */
for i = n to 2 do
  begin
    for j = i-1 to 1 do Q ← s(Q,i,j)
  end

Figure 6: Gaussian Elimination-like Solution Procedure

For the classical data flow problems, the implementation of this method can involve bit vector or set operations. The partial ordering on the n-tuples is one of component-wise set inclusion for a set implementation and component-wise comparison for a bit vector implementation. The loop breaking rules for these problems are very simple. In equation 1 if \( \theta \) is U as in REACH, then we have:

\[
Q_m: \quad Z = a n Z u \beta
\]  
(2)

where a is a constant and \( \beta \) can contain terms in other variables and constants. The corresponding loop breaking rule substitutes equation \( q_m \) for \( Q_m \):

\[
q_m: \quad Z = \beta
\]  
(3)

In equation 1 if \( \theta \) is n then we have:

\[
Q_m: \quad Z = (a n Z u c) n \beta
\]  
(4)

where a and c are constants and \( \beta \) can contain terms in other variables and constants. The corresponding loop breaking rule substitutes equation \( q_m \) for \( Q_m \):

\[
q_m: \quad Z = c n \beta
\]  
(5)

The verification of a loop breaking rule is a three step process. Clearly, \( Z_m \) does not appear on the right-hand side of \( q_m \) (i.e., i. in the definition of loop breaking rule is satisfied). The solution of the loop breaking rule \( q_m \) must be shown to satisfy the original equation \( Q_m \). Letting \( s = \beta \) in equation 2, we have:

\[
\beta =? (a n \beta) u \beta
\]

which is clearly true (i.e., ii. is satisfied). For every solution \( S \) of \( Q_m \) there must be a solution \( s \) of \( q_m \) such that \( s \leq S \). If \( S \) is a solution to \( Q_m \) then:

\[
S = (a n S) u \beta \rightarrow \beta \leq S
\]

Therefore, \( \beta \leq S \) for \( S \) any solution of \( Q_m \) (i.e., iii. is satisfied). By replacing equation 2 by equation 3, we are selecting the minimal solution for \( Z \) from the set of possible solutions.
satisfying $Q_m$.

Data flow problems are called forward or backward depending on the direction of information flow in the flow graph [Kennedy 71, Kennedy 79]. For example in REACH, variable definitions are propagated along paths in the flow graph which represent possible execution paths in the program; this is a forward data flow problem. For such problems, the set $S_m$ in equation 1 is the set of immediate predecessors of node $m$ (i.e., $\text{pred}(m)$). If information is passed backward along paths, in the direction opposite to execution flow, we have a backward data flow problem. We limit our attention to forward data flow problems, although some of the methods developed are applicable to backward data flow problems as well [Allen 77].

If $X_m$ is a data flow solution on entry to node $m$ and its equation has the form of equation 1 then $Y_m$ which is the solution for the same problem on exit from node $m$ will also have an equation of the form of equation 1. In particular, if:

$$X_m = \theta \{ a_{mj} \in \text{pred}(m) \mid X_j \cup b_{mj} \}$$

then

$$Y_m = \theta \{ d_{mj} \in \text{pred}(m) \mid Y_j \cup c_{mj} \}$$

where

- $a_{mj}$ and $b_{mj}$ are constants associated with data flow through node $j$.
- $d_{mj}$ and $c_{mj}$ are constants associated with data flow through node $m$.
- $\text{pred}(m)$ is the set of immediate predecessors of node $m$.

The choice of which system to solve depends in part on the data flow problem being solved and the use for which data flow information. There is an obvious functional relation between $X_m$ and $Y_m$. For a forward problem, $Y_m$ consists of elements of $X_m$ which are not affected by the code at node $m$ plus any relevant side effects of the code at node $m$. There is a linear function $f$ such that $X_m = f(Y_m)$. Given these linear relations between these equation forms, our model can handle data flow solutions on entry to or exit from a node equally well. In our subsequent discussions, we choose the form most convenient
for the algorithm modelled.

In data flow problems the initial equations for some variables, called boundary variables, are particularly simple. $X_{\text{source}}$ is the boundary variable of a forward data flow problem ($\{X_j\}$, for j an exit node of the flow graph, are the boundary variables of a backward data flow problem). The initial equation for a boundary variable depends upon the specific data flow problem and the equation form being used. For example, for REACH, as illustrated in Figure 4, the initial equation for $X_{\text{source}}$ is:

$$X_{\text{source}} = \#$$  \hspace{1cm} (8)

using equation 6. Using equation 7, the equation is:

$$X_{\text{source}} = \text{def}_{\text{source}}$$  \hspace{1cm} (9)

where $\text{def}_{\text{source}}$ is the set of all value-setting statements in the source node which assign a value to a variable that may be its value on exit from the source node. Correct choice of a boundary variable equation is essential to the correct solution of the data flow problem.

To illustrate these equation forms we extend the example in Section 1 to a general definition of REACH. If a definition of variable $y$ at node $n$ is in $X_m$, we say that definition reaches node $m$; this means that $y$ may have the value assigned to it at node $n$, when execution reaches the code at node $m$. A definition-clear path for variable $y$ from node $n$ to $m$ is a path along which there is no value-setting statement for $y$; therefore, the definition of $y$ at node $n$ reaches node $m$ if there is a definition-clear path for $y$ from $n$ to $m$.

REACH is the problem of finding all definitions which reach a node; $X_m$ is a set of tuples $(i,x)$ that indicate that the definition of variable $x$ at node $j$ reaches the entry of basic block $m$. This solution can be used to optimize the code generated for each basic block. For example, if all the definitions of a variable reaching node $m$ are the same constant value, then we know the variable has that value at $m$ until it is redefined; we can instantiate this constant value in the appropriate places.

Equations 10 completely describe the REACH problem. Any data flow algorithm for REACH must solve these equations. The boundary variable equation for $X_{\text{source}}$ reflects the assumption that no variable definitions reach the source node entry. Note the similarity between these equations and those in Figure 4.
\( X_m = \emptyset \quad : m = \text{source} \)
\[ = U \{ p_j \cap X_j \cup d_j \} \quad : m \neq \text{source} \]

where
- \( \text{pred}(m) \) is the set of all immediate predecessors of \( m \),
- \( X_j \) is the set of all variable definitions reaching the entry to node \( j \),
- \( p_j \) is the set of all variable definitions which may be preserved through node \( j \) (i.e., the set of definitions of variables not redefined at \( j \)),
- \( d_j \) is the set of locally exposed definitions at node \( j \), that is, the set of last definitions of each variable defined at node \( j \) [Hecht 77].

We use use an example of REACH to illustrate the need for loop breaking rules. In performing substitution transformations on equation 10 we may introduce a variable \( X_m \) on the right-hand side of its own equation \( Q_m \). This will occur if node \( m \) is an entry node of a loop in the flow graph. Figure 7 illustrates this with \( m=2 \).

Figure 7: Reach Example with Loop

Consider the substitution transformation \( s(Q,3,2) \); it introduces an \( X_2 \) term in the right-hand side of \( Q_2 \) which is eliminated by a loop breaking transformation \( b(s(Q,3,2),2) \). The resulting equation for \( X_2 \) is:

\( q_2: \quad X_2 = (p_1 \cap X_1) \cup (p_3 \cup X_3) \cup d_1 \cup d_3 \)

Let \( R = \{ Q_1, q_2, Q_3 \} \). Now two substitution transformations solve the system \( R \) (i.e., \( s(s(R,1,2),2,3) \)) yielding:
\[ X_1 = \phi \]
\[ X_2 = \left( \varphi_2 \cap d_2 \right) \cup d_1 \cup d_3 \]
\[ X_3 = \left( \varphi_2 \cap d_1 \right) \cup \left( \varphi_2 \cap d_3 \right) \cup d_2 \]

Thus the loop breaking transformations guarantee the effectiveness of the procedure in Figure 6 on flow graphs with loops.

Recall that each variable in the system of equations is identified with a unique flow graph node; its value is the data flow solution on entry to (or alternatively on exit from) the node. Given this one-to-one relationship between nodes and variables, we use these terms interchangeably; interpretation will be clear from the context. The coefficients and constants in the equations are defined using the local data flow characteristics associated with each node. The coefficients describe data flow side effects of code at a node. The constants describe elements of the data flow solution generated at a node. From a flow graph annotated with this information at each node, we can obtain a system of equations describing an associated data flow problem.

Conversely, given any set of equations of the form of equation 1 we can define a dependency graph, a directed graph corresponding to the interdependencies of variables given by the equations in that system. Each node represents a variable; each directed edge \( (m,n) \) represents the dependence of \( X_n \) on \( X_m \) (i.e., the occurrence of \( X_m \) on the right-hand side of the equation for \( X_n \)). For forward data flow problems, the dependency graph is the flow graph of problem.

Every data flow analysis algorithm must solve the system of equations which describe the problem. We compare and contrast data flow algorithms by examining how they solve these systems. The equations can be solved by a method patterned after straightforward Gaussian elimination with order \( O(n^3) \) complexity. In fact, the elimination methods described here all have better worst case bounds because they take advantage of a special coefficient structure\(^4\). This structure, first utilized by Allen/Cocke interval

\(^4\) The data flow analysis algorithms modeled here also are applicable to other problems represented by systems of equations with similar properties.
analysis, results from the sparseness of the system of equations (i.e., in control flow graphs we assume \( e \) is \( O(n) \)) and the reducibility of the dependency graph.

A reducible directed graph is one with no multiple entry loops [Hecht 77]. In a system with a reducible dependency graph, the variables of the system are naturally partitioned into groups which can affect each other only in a highly constrained manner. In practice, irreducible control flow graphs are rare; therefore, data flow methods which require reducible systems are almost always sufficient [Hecht 77, Knuth 71]. The Hecht/Ullman and Tarjan interval analysis algorithms are restricted to systems of equations with reducible dependency graphs. The Graham/Wegman algorithm can handle irreducible systems straightforwardly (see Section 6.3). Allen/Cocke interval analysis can be adjusted to handle irreducibilities as well (see Section 3.3). We should bear in mind that an irreducible system can always be solved straightforwardly, if inefficiently, by the Gaussian elimination-like method in Figure 6.

3. ALLEN/COCKE INTERVAL ANALYSIS

Historically, interval analysis is a concept first developed in the elimination algorithm in [Allen 71]. The clever idea of interval analysis is to take advantage of the reducibility of the flow graph to convert the solution of a system of \( n \) equations to the solution of a smaller system of \( r \) equations. This is accomplished by partitioning the variables into \( r \) subgroups called intervals, single-entry regions corresponding approximately to loops in the flow graph.

During the partitioning of the Allen/Cocke algorithm, a linear order of the variables within each partition element is established; this is called interval order. This order leads to a highly structured coefficient matrix amenable to ordered substitution transformations. The partitioning algorithm performed on the dependency graph is given in Section 3.1. Ordered substitution in the equations results in the reduced equation for each variable becoming a linear function of its interval head variable, where the entry node of an interval is the interval head node. A derived system is formed consisting only of \( r \) reduced interval head variable equations. This new system in turn is partitioned into intervals, each with an interval order, and the same coefficient structure is observed. This

\[5\] A subgraph is defined to be single-entry if all incident edges are incident on a single vertex. Single-exit is defined similarly.
process continues until a final variable remains and its solution is obtained. This variable is identified with an interval head variable in the previous system. The solution is substituted into the reduced equations of variables in that interval. This back substitution process is iterated through the derived systems of equations in reverse derivation order until all solutions are obtained.

In this section we present our formal model of the Allen/Cocke algorithm prefaced by informal discussions of its basic ideas. We show that the Allen/Cocke algorithm has a linear worst case complexity on a control flow graph with bounded maximal loop nesting.

3.1. Finding Intervals

The order in which nodes are added to an interval is called an interval order; it is not unique. Several characteristics of intervals are listed below [Allen 77].

i. The set of interval head nodes on a flow graph is unique.7

ii. The head node of an interval dominates internal interval nodes.

iii. An interval is single-entry.

iv. Any back edge in an interval has the interval head node as its target [Hecht 77].

v. An interval order within a given interval is consistent with the partial ordering imposed by the predecessor relations of the flow graph.

The algorithm for finding the intervals of a flow graph is given in [Allen 77] and paraphrased in Figure 8. This algorithm assures these properties.

3.2. Forward Data Flow Problems

The properties of an interval impose strong conditions on the coefficients of the corresponding equations. Figure 9 shows rows h through j of a coefficient matrix for a forward data flow problem. These rows represent the data flow equations for an interval in the flow graph whose nodes are numbered from h to j in an interval order. Each x represents a possibly non-zero entry. The possibly full row h corresponds to the equation for the entry node of the interval, called the interval head node. We call the entire

6 We are assuming here the dependency graph is reducible (see Section 3.3).

7 Recall that for forward data flow problems, the dependency graph will be the flow graph, which is defined to be single-entry.
INT := null; /*list of intervals*/
I := null; /*each interval*/
H := {s}; /*header list initialized to source node*/

while (H != null) do
  Destructively select h from H;
  I := [h]; /*form I_h*/
  while (There is a node m not in s, whose immediate predecessors are all in I
        but m is not yet in I) do
    Add m to I;
  endwhile;
  Add I to INT;
  while (There is a node n not in H and not in INT,
        with at least one predecessor in I) do
    Add n to H;
  endwhile;
endwhile;

Figure 8: Interval Finding Algorithm

interval the interval headed by h or I_h. The lower triangular "chunk" of coefficients in columns h through (j−1) of rows (h+1) to j, are the possibly non-zero coefficients in equations for internal interval nodes of I_h.

columns: 1  h  j  n
row h:  x ... x 0 x ... x x ... x
        0 ... 0 x 0 ... 0 0 ... 0
        0 ... 0 x x 0 ... 0 0 ... 0
          . . . . . . . . . .
          . . . . . . . . . .
row j:  0 ... 0 x x ... x 0 0 ... 0

Figure 9: Forward Problem Coefficient Matrix

A non-zero coefficient in rows (h+1) to j in columns 1 to (h−1) or (j+1) to n inclusive would imply that the interval is not single-entry; therefore, this cannot occur. Similarly, a non-zero coefficient appearing in the upper triangular "chunk" of rows (h+1) to j, in columns (h+1) to j inclusive, would imply that a predecessor of a node in the interval follows that node in an interval order, which likewise cannot occur. The coefficient matrix of the entire system of equations is comprised of blocks of this form along its diagonal. For a reducible flow graph, a depth first spanning tree construction allows removal of back edges from the flow graph, yielding a DAG that imposes an order on the interval head nodes that can be extended easily to an order compatible with each interval
order [Hecht 77].

Because of the coefficient structure in an interval, we can express the solution for an internal interval variable as a linear function of its interval head variable by straightforward substitution transformations. Further, the solution of the data flow problem for an interval head variable can be described completely as a linear combination of solutions for other interval head variables. The loop breaking rules for data flow problems describable by equation 1 are defined by equations 2 through 5; essentially, those rules drop the self-dependent term in the equation. This term represents repetitive data flow information which reaches h, cycles through l, and proceeds through the interval again.

Using REACH to illustrate these observations, we obtain the reduced equations 11 and 12 from the REACH equations 10. For node m in l, m H, there are constants p_{h,m} and q_{h,m} such that

\[ X_m = p_{h,m} n X_h + q_{h,m} \]  \hspace{1cm} (11)

Let H be the set of interval head nodes in the flow graph. For each m H, let H = H \setminus \{h\}. Let e_h be the constant term contributed to X_h by loops containing h. We find e_h by applying substitution transformations to l until all equations are linear functions of X_h. Then, we substitute for any terms in the equation for X_h which involve variables in l, apply the loop breaking rule (see Section 2), and obtain the constant e_h. For each m H, we have linear functions f_{i,h} such that

\[ X_h = \bigcup_{i \in H}^{+} \{ f_{i,h} \{ X_i \} \} + e_h \]  \hspace{1cm} (12)

A REACH problem worked by Allen/Cocke interval analysis is presented in Figure 18 in Section 4.5.

From these equations we see that X_h which appeared in the original system of equations remains in the derived system of equations. Thus, X_h is involved in an interval order on l in the original system of equations and in a different interval order on l_m where m H in the dependency graph of the reduced system of equations. This illustrates how a variable may be associated with k different interval orders in k derived systems.
3.3. Algorithm Statement

The key step in Allen/Cocke interval analysis reduces the solution of a system of \( n \) equations to a smaller system of \( r \) equations, where \( r \) is the number of intervals in the dependency graph of the equations. The algorithmic expression of this reduction is at the end of this section; it consists of two phases: elimination and propagation.

The elimination phase consists of iterating the following three steps: finding intervals in the dependency graph associated with the system of equations, reducing the equations to form a new system of the reduced interval head variable equations, and forming the dependency graph of the reduced system. When an interval order is established on each interval of the reduced system and the variables are ordered thusly, the coefficient matrix structure of the original problem is preserved. We recall that the arguments establishing the coefficient matrix structure utilized only the properties of intervals and an interval order (see Section 3.2). When the original flow graph is reducible, the three step process can be continued, yielding a sequence of systems of equations and a final system of one equation. The sequence of dependency graphs \( \{G_i\}_{i=1}^{k} \) corresponding to the sequence of systems of equations is called the derived sequence of graphs. We call \( G_i^{+1} \) the derived graph of \( G_i \).

The propagation phase consists of first solving the system of one equation. Then that final solution is identified with the solution for the corresponding interval head variable in the previous system. The reduced equations in the previous system are solved by back substitution of the interval head variable solution. This reverse order substitution process is repeated until all solutions are obtained.

In the remainder of this tutorial, whenever we use \( G_i \) we are referring to the derived sequence of graphs in Allen/Cocke interval analysis. When we say that \( y \) in \( G_i^{+1} \) represents \( l_h \) in \( G_i \), we are referring to the fact that all variables in \( l_h \) are represented by variable \( X_h \) in \( G_i^{+1} \), whose corresponding node in \( G_i^{+1} \) is \( y \). The definition of represents is extendible over finite subsequences of the derived sequence. Therefore, if

\[
m_1 \rightarrow l_{m_2} \leq G_i \rightarrow \cdots \rightarrow l_{m_k} \leq G_i^{+k-1}
\]

we say that \( m_k \) represents \( m_1 \) in \( G_i^{+k-1} \).
Allen/Cocks interval analysis can be adjusted to handle irreducible dependency graphs by a technique called node splitting, although this is rarely necessary with control flow graphs [Allen 72, Kennedy 74]. Node splitting involves adding nodes and edges to the flow graph so that its data flow properties are unchanged, but the irreducibility is removed. An example of this transformation is shown in Figure 10 [Hecht 77].

![Diagram showing node splitting transformation](image-url)

Figure 10: Example of Node Splitting Transformation
Model of Allen/Cocke Interval Analysis Algorithm

Elimination Phase:

i. Define the forward data flow equations on the original flow graph, \( G^1 \). Let \( k = 1 \).

ii. Find the intervals in \( G^k \) using the Allen/Cocke interval finding algorithm in Figure 8. Number the variables in each interval according to an interval order. Apply substitution transformations within each interval to obtain the system of reduced equations. Use the relevant loop breaking rule on any self-dependences introduced in the process. Use substitution transformations to render all interval head variable equations independent of non-interval head variables.

iii. Create the dependency graph \( G^{k+1} \) whose nodes are the interval head variables from \( G^k \) and whose edges are defined by the dependencies in the reduced system of equations (i.e., insert an edge \((m,n)\) if the data flow equation for \( X_n \) contains a dependence on \( X_m \)).

iv. If there is more than one node in \( G^{k+1} \) then increment \( k \) by 1 and return to step ii.

Propagation Phase:

vi. Solve the final equation.

vii. Each interval \( I_h \) in \( G^k \) corresponds to a node \( w \) in \( G^{k+1} \). For each interval head node \( h \) in \( G^k \), set \( X_h \) equal to the solution at the node corresponding to \( I_h \) in \( G^{k+1} \). \( X_w \). Then substitute this value of \( X_h \) into the reduced equations in \( I_h \) to solve for all variables. Thus, solve the system of equations associated with \( G^k \).

viii. If \( k = 1 \) stop. Otherwise, decrement \( k \) by 1 and return to step vii.
3.4. Linear Performance of Allen/Cocke Interval Analysis

In this section we show that the equation solution work of Allen/Cocke interval analysis exhibits a linear worst case complexity bound on reducible flow graphs where \( e \) is \( O(n) \) and the maximum loop nesting depth is bounded by a constant. Under these conditions, the interval finding work of the algorithm in Figure 8 will also be \( O(n) \). In general, the worst case performance of the Allen/Cocke algorithm is \( O(n^2) \) even if \( e \) is only \( O(n) \). This bound is achieved on the worst case flow graph which is pictured in Figure 11; this graph of \( n \) nodes has approximately \( 2n \) edges and a loop nested at a depth of approximately \( 2/3 \) \( n \) [Ullman 73]. However, reasonable programs do not contain highly nested loops.

![Pathological Flow Graph for Allen/Cocke Interval Analysis](image)

Empirical surveys of high level programming languages confirm that in actual usage, loop nesting depths of greater than six are rare. Two Fortran surveys in the 1970's reported that typical nesting of do statements was very shallow [Knuth 71, Robinson 76]. Knuth, reporting on programs from Stanford University and Lockheed Corporation, stated that 91% of the do statements were nested less than 4 levels. Robinson, surveying two program populations from students and systems programmers at Brunel University in England, noted that a majority of the do statements were nested less than 4 levels; 76%/84% (student/systems) of the do statements were nested less than 7 levels. The assumption of a 6 level limit on loop nesting with regards to P/L/I usage was supported by Allen [Allen 79]. Thus, in reasonable programs it is valid to assume a maximum loop nesting depth which is a constant \( k, k<<n \), independent of the number of nodes in the flow graph. Under that assumption for a reducible flow graph, Theorem 1 shows the equation solution work of the Allen/Cocke algorithm has \( O(n) \) complexity. Furthermore, a worst case form of the interval finding algorithm is \( O(n) \) on a flow graph [Hecht 77]; by restricting the loop nesting depth to \( k \) here, we restrict the possible length of the derived sequence, resulting in an \( O(n) \) bound on the interval finding over the entire algorithm.
These results corroborate the common observation that the $O(n^2)$ worst case complexity bound is not observed in practice.

**Theorem 1:** Given a reducible flow graph $G$ in which $e$ is $O(n)$, suppose the maximum loop nesting depth is less than or equal to a constant $k$. Assume Allen/Cocke interval analysis is applied to solve a forward data flow analysis problem on $G$. Then, the worst case complexity of the equation solution work of the Allen/Cocke algorithm on that flow graph is $O(n)$, where $n$ is the number of nodes in $G$.  

**Proof:** First, we consider the case where each interval corresponds to a single-entry loop in the flow graph. The loops will be single-entry because of the reducibility of $G$ [Hecht 77]. Second, we argue the case when non-loop intervals occur.

(i.) From our model of Allen/Cocke interval analysis, we see that the terms in the system of equations can be partitioned into two disjoint sets: a set $S_1$, whose elements are substituted for once during elimination and a set $S_2$ of those which are substituted for more than once. The work of elimination can be calculated by considering the sum of the elimination work for terms in $S_1$ and $S_2$. The elimination work for terms in $S_1$ is $O(|S_1|) \leq O(n)$, by our assumption that $e$ is $O(n)$.

The terms in $S_2$ occur in interval head equations in $G$. If a term involving $X_j \in S_2$ occurs in the equation of $X_i$ in $G$, then in step iii. of the Allen/Cocke algorithm a linear function of $X_h$ will be substituted for the variable $X_j$ where $j \notin S_1$ in $G$. Likewise, in $G^2$ a linear function of $X_q$ will be substituted for the variable $X_h$ in the equation of $X_i$ where $h \notin S_1$ in $G^2$. Because $G$ is reducible, this process continues for finitely many steps until the edge representing $(j,i)$ in $G$ no longer exists on some $G^k$. It can be shown that loops are collapsed in innermost to outermost order by the Allen/Cocke algorithm [Ryder 82b]. Each step in the derived sequence represents the collapse of the innermost loop of a nested loop. Since each interval corresponds to a loop and the maximum nesting level is $k$, all loops will be collapsed in $G^{k+1}$. Therefore, edge $(j,i)$ can exist on at most $(k+1)$ graphs in the derived sequence.

---

8We conjecture that Theorem 1 may hold with weaker restrictions on $k$ and/or $e$. 
Then the elimination cost for terms in $S_2$ is bounded above by $c |S_2| (k+1)$ for $c$ a constant. Since the total number of terms is the number of edges in the original flow graph, both $|S_1|$ and $|S_2|$ are no greater than $O(n)$. Therefore, the elimination work is bounded by $O(n)$.

The propagation work is bounded by the number of edges in the entire derived sequence [Hecht 77]. Since the number of edges in successive graphs in the derived sequence decreases, the number of edges in any $G^i$ is bounded by $O(n)$. Therefore, the total number of edges is bounded by $b n (k+1)$ for $b$ a constant and $(k+1)$ the length of the derived sequence. Therefore, $O(n)$ also bounds the work of the propagation phase.

Therefore, here the worst case complexity of the equation solving by Allen/Cocke interval analysis is bounded by $O(n)$.

(iii) There are Allen/Cocke intervals called null loops which do not correspond to loops in the flow graph. However, in the derived graph $G^i$ where $i > 1$, these are trivial intervals of one node, an interval head node [Ryder 82,b]. By the properties of the interval finding algorithm of Figure 8 we can show that if there are no loops in the graph $G^i$ then there are no null loops in $G^i$. Thus, there are no null loops in $G^{k+1}$.

If a term involving a variable from within a null loop exists in $S_1$, our work estimates in (i.) are valid here. If a term involving a variable from within a null loop exists in $S_2$, then we know that it can be substituted for at most $k$ times. Thus, by arguments similar to those in (i.) the equation solution work of Allen/Cocke interval analysis is bounded by $O(n)$.

Q.E.D.

4. HECHT/ULLMAN T1-T2 ANALYSIS

In the next three sections we present models of three closely related data flow algorithms. All are improvements on the Allen/Cocke algorithm. On flow graphs where the number of edges $e$ is $O(n)$, these algorithms achieve a worst case bound of $O(n \log n)^9$ rather than the $O(n^2)$ bound of the Allen/Cocke algorithm. They achieve their performance improvement through the delay of certain calculations and the discovery and utilization of

---

9-The Tarjan algorithm can achieve an almost linear worst case bound of $O(n \alpha(n))$ but its practical implementation explained in Section 5 achieves a $O(n \log n)$ bound [Tarjan 81a].
common substitution factors in the equations. We compare them with the Allen/Cocke algorithm on forward data flow problems.

In this section we present our model of Hecht/Ullman T1-T2 analysis. The grouping of variables is less constrained than in the Allen/Cocke algorithm and is performed using a "nearest neighbor" heuristic. The delay in the calculation of common coefficient/constant factors in the reduced equations yields savings in elimination. These shared factors necessitate the use of a data structure to remember them. A height-balanced 2-3 tree is used [Aho 77]. The Hecht/Ullman method consists of three phases analogous to those of Allen/Cocke interval analysis: parse generation, elimination and propagation. We contrast each of these phases with the corresponding phase in the Allen/Cocke algorithm presented in Section 3. The Hecht/Ullman algorithm can only be applied to programs with reducible flow graphs [Hecht 77]; as discussed previously in Section 2, this is not really a limitation in practice.

First we discuss the Hecht/Ullman parse generation algorithm which determines the variable subgroups and the order of variable substitution in the equations. Graph transformations T1 and T2 applied to the dependency graph of the equations define this order. We define concepts necessary for understanding T1 and T2 transformations and present a short outline of the actual parse algorithm [Hopcroft 72, Ullman 73]. Second, we explain the elimination phase of the Hecht/Ullman algorithm in terms of our model, describing the equation manipulations corresponding to the T1 and T2 transformations. Third, we discuss the propagation phase of the algorithm. Fourth, we state the algorithm. Fifth, we compare the parse and elimination phases with Allen/Cocke interval analysis.

4.1. Parse Generation

The Hecht/Ullman T1-T2 algorithm assumes a data flow problem is described by a system of equations of the form of equation 7 with an associated dependency graph (i.e., the flow graph). The algorithm uses single-entry subgraphs of the flow graph called regions to direct its elimination phase, much as the Allen/Cocke algorithm uses intervals. In a region R there is one vertex, the region head h, such that all edges from outside of region R to nodes in R are incident on h. If a node y is within a region headed by h (i.e., \( R_h \)), then at some time during the algorithm, the reduced equation for \( X_y \) is a linear function of \( X_h \). This is analogous to the relation between internal interval nodes and interval head nodes. Regions are manipulated using the transformations T1 and T2, illustrated in
Figure 12: Examples of T1 and T2 Transformations

Before T1 Transformation
\[ Y_u = p_u \cap (Y_z \cup Y_u) \cup d_u \]

After T1 Transformation
\[ Y_u = p_u \cap Y_z \cup d_u \]

Before T2 Transformation
\[ Y_w = p_w \cap Y_u \cup d_w \]
\[ Y_y = p_y \cap Y_w \cup d_y \]

After T2 Transformation
\[ Y_w = p_w \cap Y_u \cup d_w \]
\[ Y_y = p_y \cap p_w \cap Y_u \cup p_y \cap d_w \cup d_y \]
Figure 12, which correspond to loop breaking and substitution transformations on the system of equations. These substitutions and the meaning of the edge sets $E_1$ and $E_2$ in terms of the equations will be explained in Section 4.2.

A parse of a reducible flow graph is a sequence of $T_1$ and $T_2$ transformations which when applied to the flow graph results in its collapse to one node [Hecht 77]. Each transformation in the parse is called a parse element. A flow graph which is transformed to one node is called collapsed. Hecht and Ullman proved that $T_1$ and $T_2$ form a finite Church-Rosser transformation, which means that they need only be applied finitely many times to a reducible flow graph and that the outcome is independent of the order of their application [Hecht 77].

The parse generation algorithm is derived from an algorithm of Hopcroft and Ullman [Hopcroft 72, Ullman 73]. The algorithm examines a flow graph which is represented by a set of nodes and the lists of in-edges and out-edges associated with each node. The order of the edges on these lists influences the parse generated. An explicit search and test are made for each $T_2$ transformation; $T_1$ transformations result from these tests whenever a self-loop is detected. The same node may appear in more than one $T_1$ transformation in a parse, although an edge can appear in only one parse element.

![Diagram](image)

**Figure 13:** $T_2$ Transformation Candidate

Initially in parse generation, a sweep through all nodes and edges in the flow graph finds $T_2$ candidates and any self-loops. Figure 13 illustrates a subgraph in which node $v$ is a $T_2$ candidate because it has a unique parent. For each $T_2$ candidate $v$, its immediate neighbors are checked for $T_2$ candidacy. First, each immediate descendant of $v$ is checked to see if it becomes a $T_2$ candidate after $T_2(z,v,v)$ and if necessary $T_1(y,v)$ are performed. Second, $z$ the immediate parent of $v$ is checked to see if it becomes a $T_2$ candidate after $T_2(z,v,v)$ and if necessary $T_1(z,v)$ are performed; for example, this may occur if $(v,z)$ prevented $z$ from being a $T_2$ candidate previously, as in Figure 13. The parse of a flow
graph is non-unique; the order of the transformations obviously depends on the flow graph representation. The outline of the parse generation algorithm is given in Figure 14.

L := null; /*L is a list of T2 candidates*/
for i := 1 to n do
  if in-edges(i) contains (i,i) then Generate T1(i, x);
  if in-edges(i) contains only one edge then Add i to L;
endfor;
while L≠null do
  Destructively select v from L;
  Find unique predecessor of v, z;
  Generate T2(z, v, x);
  Determine if z can be added to L,
  perhaps after a T1 transformation of z;
  for each immediate descendant of v, y do
    Determine if y can be added to L,
    perhaps after a T1 transformation of y;
  endfor;
  Add out-edges(v) to out-edges(z);
endwhile;

Figure 14: Parse Generation Algorithm

In practice, when manipulating edge lists, the parse generation algorithm of Figure 14 always merges the smaller region into the larger one by keeping a count of the number of nodes represented by each node in the partially collapsed flow graph [Ullman 73]. This strategy insures a worst case bound for flow graphs where e is O(n) of O(n log n) [Hopcroft 72, Ullman 73].

4.2. T1-T2 Transformations and Elimination

The calculations of the elimination phase are directed by the parse of the flow graph. A region in equation terms is a subgroup of variables all of which have reduced equations which are linear functions of the region head variable. Our descriptions of the T1 and T2 transformations in Section 4.1 are graphical. In this section we explain the sequence of equation manipulations to which they correspond. Each T1 or T2 transformation triggers a coefficient/constant calculation which further reduces at least one of the equations in the system. Examples of these calculations are given in the REACH equations in Figure 12.

A T2 transformation can be applied when a node has a unique predecessor, that is, when the equation of the corresponding variable is a function of one variable. The T2 transformation T2(u, w, E) in Figure 12(ii) merges R_w the region represented by node w in
the partially collapsed flow graph into its unique predecessor region \( R_u \) represented by node \( u \). Here, \( E_2 \) is the set of edges in the original flow graph represented by \( (u,w) \) in the partially collapsed flow graph. In the elimination phase, this T2 parse element corresponds to selecting two subgroups of variables \( (R_u, R_w) \) each with a region head variable \( (Y_u, Y_w) \) and merging them into one subgroup \( (R_u) \). After the merge, there is one region head variable \( Y_u \) representing all the members of the newly merged subgroup; therefore, the reduced equation of each variable in the new subgroup is a linear function of \( Y_u \).

Each edge in the set of edges \( E_2 \) corresponds to a term in the original equation for \( Y_w \). These terms are represented in the partially reduced equation for \( Y_w \) by the term in \( Y_u \). When the parse element \( T2(u,w,E_2) \) is performed, we do a sequence of substitution transformations such that the right-hand side of the reduced equation for \( Y_w \) a linear function of \( Y_u \) is substituted into the equations of any variables currently dependent on \( Y_w \). These will include all variables in \( R_w \) represented by \( w \) in the partially collapsed flow graph, as well as all variables corresponding to immediate descendants of \( w \); in Figure 12(ii), the latter category includes \( Y_y \). Updated reduced equations are obtained for all nodes in \( R_w \) and for these immediate descendant nodes; thus, all dependence on \( Y_w \) in the current system is eliminated.

A T1 transformation is applied to remove a self-loop, or in equation terms, a variable from the right-hand side of its own equation. The T1 transformation \( T1(u,E_1) \) in Figure 12(i), removes a self-loop from node \( u \). \( E_1 \) is the set of edges in the original flow graph represented by \( (u,u) \) in the partially collapsed flow graph. Each edge in \( E_1 \) corresponds to a term in the original equation for \( Y_u \); each was a back edge to \( u \) in the original flow graph. When the \( T1(u,E_1) \) parse element is encountered, the heads of these edges are nodes already in region \( R_u \). When they were merged by previous T2 transformations into \( R_u \) the associated variable substitutions may have resulted in the introduction of \( Y_u \) on the right-hand side of the partially reduced equation for \( Y_u \). The self-loop in Figure 12(ii) represents this dependence. In the elimination phase, when a T1 parse element is encountered, we apply the appropriate loop breaking rule (see Section 2) to eliminate any dependence of \( Y_u \) on itself.

The basic elimination step of the Hecht/Ullman T1-T2 algorithm, associated with the T2 transformation, is the complete removal of a particular variable in the partially reduced system of equations (e.g., performing T2(\( u,w,E_2 \)) removes \( Y_w \) from the system). In practice.
the algorithm actually performs the calculation associated with T2(u,w,E₂), only for variables whose nodes are in the region R_u after the T2 graph transformation is performed; all other calculations are delayed. That is, if the equation for Y_z contains a term Y_w and z ∈ R_u after T2(u,w,E₂) is performed, then replacement of Y_w by a linear function of Y_u (i.e., siO.w,z)) is delayed until the time when z and w are in the same region. At that time, occurrences of Y_w will be replaced by the right-hand side of the then current reduced equation for Y_w. Eventually z and w must be in the same region, as all nodes are finally in the region of the entire graph, R_{source}.

![Diagram](image)

**Figure 15: Delayed Substitutions Example**

For example, in Figure 15 the graph has two possible parses. In both, when T2(3,4,\{(3,4)\}) is performed, node 5 is neither in R_3 nor R_4. The existence of edge (4,5) implies that there is an Y_4 term in the equation of Y_5. The replacement of that Y_4 term is delayed until nodes 4 and 5 are in the same region; this occurs after parse element T2(1,5,\{(2,5) \ (3,5) \ (4,5)\}) is performed. Then, the current reduced equation for Y_4 as a linear function of Y_1 is substituted into the equation for Y_5.

The delay in performing out-of-region variable substitutions enables the Hecht/Ullman

---

10 It has a unique interval order \{1,2,3,4,5\}.
algorithm to avoid performing recalculation of common coefficient factors in some reduced equations. These factors occur because common substitution sequences exist in the system; the example in Figures 16 and 17 illustrate these. Figure 16 shows the Ullman worst case flow graph for Allen/Cocks interval analysis for n=10 (see Figure 11), with a possible parse [Ullman 73]. Figure 17 shows the Hecht/Ullman algorithm applied to a REACH problem formulated on that flow graph, using equations of the form of equation 7. The algebraically simplified, partially reduced equations for $Y_1$, $Y_2$, $Y_3$ and $Y_4$ are displayed at various times during the elimination.

After parse element 7, is performed, the reduced equations for $\{Y_i\}_{i=5}^{10}$ are linear functions of $Y_4$. The equation for $Y_1$ is the same as it was initially because the substitution for the $Y_{10}$ term has been delayed. Likewise, the equations for $Y_2$ and $Y_3$ are the same as initially, with the substitution for the $Y_9$ and $Y_7$ terms delayed. After parse element 9, is performed $Y_3$. $Y_4 \equiv R_3$. The $Y_7$ term in the equation for $Y_3$ is replaced by the right-hand side of the reduced equation for $Y_4$ as a linear function of $Y_3$ and a loop breaking rule is applied. After parse element 11, is performed $Y_2$. $Y_3$. $Y_4 \equiv R_2$. The delayed substitution for the $Y_9$ term in the equation for $Y_2$ is performed using, as a

---

Because node 1 has a predecessor, node 10, it does not satisfy our definition of the source node. Nevertheless, we can analyze this flow graph by considering $Y_1$ a boundary variable whose equation is of the form of equation 9 given in Section 2 plus a term for $Y_{10}$
Figure 17: Hecht/Ullman Algorithm on REACH Problem for Figure 16

Initially

\[ Y_1 = p_1 \cap Y_{10} \cup d_1 \]
\[ Y_2 = p_2 \cap Y_1 \cup p_2 \cap Y_9 \cup d_2 \]
\[ Y_3 = p_3 \cap Y_2 \cup p_3 \cap Y_7 \cup d_3 \]
\[ Y_4 = p_4 \cap Y_3 \cup p_4 \cap Y_6 \cup d_4 \]

Let \( d_1 \ldots d_k = p_1 \cap \ldots \cap p_{k-1} \cap Y_{k-1} \)

**After parse element 7.** \( Y_4 \in R_4', Y_3 \in R_3', Y_2 \in R_2', Y_1 \in R_1 \)

\[ Y_1, Y_2, Y_3 \text{ same as initially} \]
\[ Y_4 = p_4 \cap Y_3 \cup p_4 \cap (p_5 \cap (p_6 \cap (p_7 \cap (p_8 \cap (p_9 \cup d_9) \cup d_8) \cup d_7) \cup d_6) \cup d_5) \]

**After loop breaking,**

\[ = p_4 \cap Y_3 \cup d_5 \]
\[ = a \cap Y_3 \cup b \]

**After parse element 9.** \( Y_1 = R_1', Y_2 = R_2', Y_3 = R_3', Y_4 = R_3 \)

\[ Y_1, Y_2 \text{ same as initially} \]
\[ Y_3 = p_3 \cap Y_2 \cup p_3 \cap (p_7 \cap (p_9 \cap (a \cap Y_3 \cup b) \cup d_7) \cup d_6) \cup d_5 \]

**After loop breaking,**

\[ = p_3 \cap Y_2 \cup d_5 \]
\[ = c \cap Y_2 \cup d \]

\[ Y_4 \text{ same as after parse element 7.} \]
After parse element 11. $Y_1 \in R_1$, $Y_2$, $Y_3$, $Y_4 \in R_2$

$Y_1$ same as initially
$Y_2 = p_2 \land Y_1 \lor p_2 \land (p_9 \land (e \land (c \land (Y_2 \land u \land b) \land u \land d) \land u \land d_9) \land u \land d_2)$
After loop breaking and simplification,
$= p_2 \land Y_1 \land u \land d_2 \land 9 \land 8 \land 4 \land 3 \land 7 \land 5 \land u \land d_2 \land 9 \land 8 \land 4 \land 8 \land 5$
$= e \land n \land Y_1 \land u \land f$

$Y_3$ same as after parse element 9.
$Y_4 = a \land n \land (c \land n \land Y_2 \land u \land d) \land u \land b$

After parse element 13. $Y_1$, $Y_2$, $Y_3$, $Y_4 \in R_1$

$Y_1 = p_1 \land (p_{10} \land (p_9 \land (e \land (c \land (e \land n \land Y_1 \land u \land f) \land u \land d) \land u \land b) \land u \land d_8) \land u \land d_{10}) \land u \land d_1$
After loop breaking and simplification,
$= p_1 \land (p_{10} \land (p_9 \land (e \land n \land e \land f \land u \land a \land n \land d \land u \land b) \land u \land d_8) \land u \land d_{10}) \land u \land d_1$
$= d_1 \land 1 \land 0 \land 8 \land 4 \land 6 \land 5 \land u \land d_1 \land 1 \land 0 \land 8 \land 4 \land 3 \land 7 \land 5 \land u \land d_1 \land 1 \land 0 \land 8 \land 4 \land 3 \land 2 \land 9$

$Y_2$ same as after parse element 11.
$Y_3 = c \land n \land (e \land n \land Y_1 \land u \land f) \land u \land d$
$Y_4 = a \land n \land (c \land n \land (e \land n \land Y_1 \land u \land f) \land u \land d) \land u \land b$
subcalculation, the right-hand side of the reduced equation for $Y_4$ as a linear function of
$Y_2$.

$$Y_4 = a \cap (c \cap Y_2 \cup d) \cup b$$  \hspace{1cm} (13)

After parse element 13 is performed, all variables are contained in $R_1$. The delayed
substitution for the $Y_{10}$ term in the equation for $Y_1$ is performed using, as a subcalculation,
the right-hand side of the equation 13. Therefore, the equations for $Y_1$ and $Y_2$ share a
common inter-regional substitution factor, the right-hand side of equation 13, introduced
by the variable substitutions for the $Y_{10}$ and $Y_9$ terms respectively.

The control flow paths:

$$<2 \ 3 \ 4 \ 8 \ 9 \ 2>$$
$$<2 \ 3 \ 4 \ 5 \ 7 \ 3 \ 4 \ 8 \ 9 \ 2>$$

which are substitution sequences in the system as well, all share subpath $<2 \ 3 \ 4>$, which is
an inter-regional path containing three region heads which are back edge targets. The
variable substitutions along that subpath resulting in equation 13 are only calculated once by
the Hecht/Ullman algorithm. The longer the common inter-regional substitution paths, which
are shared by two or more factors in the system of equations, the larger is the savings.

Efficient use of these delayed common calculations requires the use of an
appropriate data structure. The Hecht/Ullman method builds a 2–3 height-balanced
calculation forest to keep track of the common factors [Aho 78, Ryder 82b, Ullman 73].
At the end of the elimination phase, one tree contains all the reduced equations in factored
form.

The savings provide a solution for a flow graph where $e$ is $O(n)$, whose complexity
is $O(n \log n)$ rather than $O(n^2)$ like the Allen/Cocke algorithm. In Section 4.5, the
Allen/Cocke algorithm is applied to the flow graph in Figure 16. Comparison shows the
calculations saved by the Hecht/Ullman algorithm. We will also solve this example using
Tarjan interval analysis in Section 5.5 and Graham/Wegman analysis in Section 6.4.
4.3. Propagation

The propagation phase of the Hecht/Ullman algorithm involves only the back substitution of the value of the source node variable. By substitution of this solution in each reduced equation, the solution for every other variable is obtained.

4.4. Algorithm Statement

In the first phase of the Hecht/Ullman algorithm a parse generation method forms a parse of the flow graph, establishing an order for the elimination phase substitutions. At the end of this phase, all equations are reduced to linear functions of the source node variable. Then, the propagation phase finds the solution for the source node variable and uses the reduced equations to solve for all other variables in the system.
**Model of Hecht/Ullman T1-T2 Analysis Algorithm**

**Parse Generation:**

i. Find a T1-T2 parse of the flow graph (see Figure 14) to establish a substitution order for the terms in the system.

**Elimination Phase:**

ii. In parse order for each parse element, do

a. If the parse element is $T2(i,j,E_2)$ then perform any delayed substitution transformations necessary to transform the equation for $Y_j$ into:

$$Y_j = a \cap Y_i \cup b$$  \hspace{1cm} (14)

where $a$ and $b$ are constants. Change any dependence on $Y_j$ in equations for variables whose corresponding nodes are in $R_i \cup R_j$ into a dependence on $Y_i$ by a sequence of substitution transformations that substitute the right-hand side of equation 14 for each $Y_j$ term. Delay this substitution for nodes outside $R_i \cup R_j$.

b. If the parse element is $T1(i,E_1)$ then perform any delayed substitution transformations for $Y_j$ where $(i,j) \in E_1$. Apply the relevant loop breaking rule (see Section 2) to eliminate $Y_i$ from the right-hand side of the current reduced equation for $Y_i$.

**Propagation Phase:**

iii. Determine the solution of $Y_{source}$. Substitute the value of $Y_{source}$ into each reduced equation to obtain a solution.
4.5. Comparison to Allen/Cocke Interval Analysis

The complexity distinction between the Allen/Cocke and Hecht/Ullman algorithms arises because the latter finds common factors in the reduced equations which elude the former. Figure 18 illustrates the common factors whose multiple calculations are saved by the Hecht/Ullman computation; it shows Allen/Cocke interval analysis applied to the example of Figure 16, highlighting the equations for $Y_1$, $Y_2$, $Y_3$ and $Y_4$ in the sequence of systems.\footnote{For ease of comparison, we assume that equations of the form of equation 7 are used by the Allen/Cocke algorithm, rather than equations of the form of equation 6.} We use the same names for the constants wherever possible in Figures 17 and 18 for ease of comparison. In calculating the reduced equations of interval head nodes in $G^1$, the $Y_{10}$ term in the equation for $Y_1$ is replaced by a linear function of $Y_4$, defined by the right-hand side of the reduced equation of $Y_{10}$ since $106|Y_4$. Similarly, the $Y_9$ and $Y_7$ terms in the equations for $Y_2$ and $Y_3$ are replaced by linear functions of $Y_4$. Substitution for the $Y_6$ term in the equation of $Y_4$ triggers application of a loop breaking rule resulting in the $Y_4$ equation in $G^2$ shown in Figure 18. In obtaining reduced equations in $G^2$, the $Y_4$ terms in the equations of $Y_1$, $Y_2$ and $Y_3$ each are replaced by a linear function of $Y_3$ derived from the reduced equation for $Y_4$ as a linear function of $Y_3$. A loop breaking rule is applied to the equation of $Y_3$ to obtain the reduced equation of $Y_3$ as a linear function of $Y_2$. In the reduced equation derivation in $G^3$, the two $Y_3$ terms in the equations for $Y_1$ and $Y_2$ each are replaced by a linear function of $Y_2$. Using a loop breaking rule, we obtain the reduced equation of $Y_2$ as a linear function of $Y_1$. Finally, in $G^4$ the $Y_2$ term in the equation of $Y_1$ is replaced by a linear function of $Y_1$. After loop breaking and simplification, we will have calculated the source node reduced equation.

The substitutions represented by the right-hand side of equation 13 in Section 4.2, performed in the derivation of reduced equations, are duplicate work indicating the possibility of savings due to common factors. Essentially, the Hecht/Ullman method perceives the $1_{453}5_{32}$ reduction and calculates the substitutions associated with it only once. The more inter-interval paths occurring in the flow graph, the more common substitution sequences which may occur. For example, in a heavily nested loop structure with many back edges to outer loops, there may be many common factors.
Figure 18: Allen/Cocke Algorithm on REACH Problem for Figure 16

\[ G^2 \quad G^3 \quad G^4 \quad G^5 \]

Let \( d_{i_{1}...i_{k}} = p_{1_{1}} n...n_{p_{i_{k-1}}} n_{d_{i_{k}}} u p_{i_{k}} n...n_{p_{i_{k-2}}} n_{d_{i_{k-1}}} u...u d_{i_{1}} \)

Equations in \( G^1 \)

\[
\begin{align*}
Y_1 &= p_1 n Y_{10} u d_1 \\
Y_2 &= p_2 n Y_1 u p_2 n Y_9 u d_2 \\
Y_3 &= p_3 n Y_2 u p_3 n Y_7 u d_3 \\
Y_4 &= p_4 n Y_3 u p_4 n Y_6 u d_4
\end{align*}
\]

Equations in \( G^2 \)

\[
\begin{align*}
Y_1 &= p_1 n (p_{10} n (p_8 n Y_4 u d_8) u d_{10}) u d_1 \\
Y_2 &= p_2 n Y_1 u p_2 n (p_9 n (p_8 n Y_4 u d_8) u d_9) u d_2 \\
Y_3 &= p_3 n Y_2 u p_3 n (p_7 n (p_5 n Y_4 u d_5) u d_7) u d_3 \\
Y_4 &= p_4 n Y_3 u p_4 n (p_6 n (p_5 n Y_4 u d_5) u d_6) u d_4
\end{align*}
\]

After loop breaking,

\[
\begin{align*}
&= p_4 n Y_3 u d_4 \text{ 6 5} \\
&= a n Y_3 u b
\end{align*}
\]
Equations in $G^3$

$Y_1 = p_1 \land (p_{10} \land (p_8 \land (a \land Y_3 \land b) \land d_8) \land d_{10}) \land d_1$
$Y_2 = p_2 \land Y_1 \land p_2 \land (p_9 \land (p_8 \land (a \land Y_3 \land b) \land d_8) \land d_9) \land d_2$
$Y_3 = p_3 \land Y_2 \land p_3 \land (p_7 \land (p_5 \land (a \land Y_3 \land b) \land d_7) \land d_3) \land d_3$

After loop breaking and simplification,

$= p_3 \land Y_2 \land d_3 \land 5 \land 4 \land 6 \land 5$

$= e \land Y_2 \land d$

Equations in $G^4$

$Y_1 = p_1 \land (p_{10} \land (p_8 \land (a \land (c \land Y_2 \land d) \land b) \land d_8) \land d_{10}) \land d_1$
$Y_2 = p_2 \land Y_1 \land p_2 \land (p_9 \land (p_8 \land (a \land (c \land Y_2 \land d) \land b) \land d_8) \land d_9) \land d_2$

After loop breaking and simplification,

$= p_2 \land Y_1 \land d_2 \land 9 \land 8 \land 4 \land 3 \land 7 \land 5 \land d_2 \land 9 \land 8 \land 4 \land 6 \land 5$

$= a \land Y_1 \land f$

Equations in $G^5$

$Y_1 = p_1 \land (p_{10} \land (p_8 \land (a \land (c \land (e \land Y_1 \land f) \land d) \land b) \land d_8) \land d_{10}) \land d_1$

After loop breaking and simplification,

$= p_1 \land (p_{10} \land (p_8 \land (a \land c \land n \land f \land u \land a \land d) \land b) \land d_8) \land d_{10}) \land d_1$

$= d_1 \land 10 \land 8 \land 4 \land 3 \land 2 \land 9 \land d_1 \land 10 \land 8 \land 4 \land 3 \land 7 \land 5 \land d_1 \land 10 \land 8 \land 4 \land 6 \land 5$
5. TARJAN INTERVAL ANALYSIS

This section presents our model of Tarjan interval analysis which we contrast with Allen/Cocke interval analysis and Hecht/Ullman T1-T2 analysis. The node order for variable substitutions in Tarjan interval analysis is similar to the Allen/Cocke algorithm; however, the definition of a Tarjan interval as a single-entry, strongly connected subgraph [Reingold 77] of the dependency graph of the original system of equations is more restrictive than an Allen/Cocke interval and more closely models the loop structure of the underlying flow graph [Tarjan 74]. The key elements of the Tarjan algorithm are the order of variable substitution and the judicious delay of certain substitutions until a time when common factors can be detected, calculated once, and used.

Tarjan interval analysis consists of three phases: interval finding, elimination and propagation. For clarity we explain these as distinct although the first two can be intermingled. Interval finding defines a node order, reduction order, closely connected to the depth-first spanning tree construction. Variable elimination occurs in each interval according to the reduction order. Some substitutions are delayed, as in the Hecht/Ullman algorithm, enabling the Tarjan algorithm to take advantage of common substitution sequences in the equations. The propagation phase performs back substitutions of known solutions into reduced equations of variables dependent on them. Tarjan interval analysis is applied to programs with reducible flow graphs; once again, this is not a restriction in practice.

First we present the node order used by Tarjan interval analysis to order the variable substitutions during elimination. Second, we consider the elimination phase of Tarjan interval analysis, defining the T3 graph transformation and its corresponding equation manipulations. With several examples we explain how the Tarjan algorithm achieves the same delayed calculation savings as the Hecht/Ullman algorithm. Third, we discuss the propagation phase of the algorithm. Fourth, we state the Tarjan interval analysis algorithm. Fifth, we compare all three algorithms modelled so far.

5.1. Reduction Order and Finding Intervals

Tarjan interval analysis assumes a data flow problem described by a system of equations of the form of equation 6 with an associated dependency graph. Like the Allen/Cocke algorithm, the Tarjan algorithm uses subgraphs of the dependency graph called intervals to direct its elimination phase. An interval here is a single-entry, strongly
connected subgraph, differing from an Allen/Cocke interval, which need not even contain a cycle; the Tarjan interval more closely reflects the loop structure of the flow graph. The term interval in this section refers to Tarjan intervals unless otherwise indicated. \( I_h \) represents the interval headed by \( h \). If \( n \in I_h \), then the reduced equation calculated for \( X_n \) is a linear function of \( X_h \). By definition, the source node is the interval head node of the outermost interval, which need not be strongly connected.

In calculating intervals, Tarjan interval analysis defines a linear order on the nodes called reduction order. Reduction order determines the order in which reduced equations are calculated in an interval, and the relative order among the intervals themselves. The determination of a reduction order for a reducible flow graph is fairly straightforward using a depth-first spanning tree, which we refer to as a DFST [Schwartz 78, Tarjan 72, Tarjan 74]. First, we form a DFST on the flow graph \( G \), rooted at the source node of \( G \), numbering the nodes by a preorder traversal. We obtain the set of all back edges in \( G \); back edge targets become interval head nodes.

Second, for each back edge target \( x \) in reverse preorder, we repeat the following procedure. We calculate the set reachunordered(\( x \)), where a node \( n \) is a member of this set if there is a simple path from \( n \) to \( x \) whose final edge is a back edge [Schwartz 78]. Reachunordered(\( x \)∪\{\( x \}) is the interval \( I_x \). All the nodes in \( I_x \) are removed from \( G \) and represented by node \( x \) in a newly derived flow graph. We set \( \text{HIGHPT}(y) = x \) for all nodes \( y \) in reachunordered(\( x \)), continuing to form reachunordered sets using the newly derived graph. If the original flow graph is reducible, this graph transformation process will result in a final graph consisting of one interval, \( I_{\text{source}} \). This final interval contains all nodes which are not within any strongly connected subgraph of the flow graph, and some nodes representing intervals not nested within any other intervals.

Third, after all the intervals are calculated, we number the nodes according to an ancestor-first, rightmost-first traversal of the DFST, calling this numbering SNUMBER. If \( (y,v) \) is a tree or cross edge, then \( SNUMBER(y) < SNUMBER(v) \). We associate the tuple:

\[
(\text{HIGHPT}(y), SNUMBER(y)) = (y_1, y_2)
\]

with every node \( y \), sorting these tuples so that \( x \) precedes \( y \) in reduction order if and only
if $x_1 > y_1$ or $x_1 = y_1$ and $x_2 < y_2$. From the derivation we see that any DFST has a unique associated reduction order.

Nodes within the same interval will occur as contiguous subsequences in the reduction order, since all HIGHT values within an interval are its interval head node. Therefore, if reduced equations in the system are calculated in reduction order, all the equations in one interval are reduced before any equations in the next interval. Reduction order is an ancestor-first order on each interval; this is similar to Allen/Cocke interval order. When the equation for $X_n = l_h$ is being reduced to a linear function of $X_h$, this ancestor-first property insures that every term on the right-hand side of the equation for $X_n$ already has a reduced equation which is a linear function of $X_h$.

The selection criteria on HIGHT insures that the equations for variables in inner, nested intervals are reduced before the equations for variables in outer, syntactically surrounding intervals, another property shared with Allen/Cocke interval analysis. Nestings of intervals can be traced by following reverse sequences of HIGHT values corresponding to interval head nodes. For example, if $x \in l_h \leq l_q$ then \{HIGHT\}(x) = h, HIGHT(h) = q\}. The HIGHT function yields loop nesting information for the program represented by the flow graph, since Tarjan intervals directly correspond to loops.

We can show that SNUMBER values for nodes within the same interval guarantee that if there is a path from y to v of tree and/or cross edges in the DFST, then $y > v$ in reduction order, substantiating our claim that reduction order within an interval is an ancestor-first order. If only tree edges appear on the path, finite induction on the definition of SNUMBER yields this result. Figure 19 illustrates the case where tree and cross edges are involved; here $v, y \in l_h$ and there is a path of tree edges from y to x represented by a solid line and one cross edge from x to v represented by a dotted line. By finite induction on the definition of SNUMBER, we have SNUMBER(y) < SNUMBER(x). Also SNUMBER is defined as a rightmost-first order on the DFST, so SNUMBER(x) < SNUMBER(v). Therefore, SNUMBER(y) < SNUMBER(v).

These reduction order and interval finding calculations can be accomplished in time

\[13\] An exception is that the source node always is last in reduction order. The sort is performed by a radix sort which is $O(n)$ [Knuth 68, Tarjan 74]
bounded by O(e a(e,n)) where a(e,n) is related to the inverse of Ackermann's function and a(e,n)\geq 3 virtually always [Schwartz 78, Tarjan 81a]. For a flow graph where e is O(n) this reduces to O(n log n). Reference [Schwartz 78] gives a SETL procedure for performing an optimization of the reduction order calculation. A simpler O(n log n) algorithm to compute the Tarjan intervals of a flow graph is also available, using path compressed trees rather than the balanced, path compressed trees needed to achieve the almost linear bound [Tarjan 79]. It is the implementation suggested by Tarjan for practical use [Tarjan 81a].

Figure 20 presents an example of a reduction order calculation. In the flow graph, the DFST edges are solid lines, the back edges are dashed lines and the cross edge appears as a dotted line. Listed in a table are the HIGHPT and SNUMBER values for the nodes, the set of intervals on the flow graph and the reduction order for this DFST. The appearance of \{9,10,12\} in \text{I}_2 indicates that when \text{I}_9 is collapsed to one node 9, that node is an internal interval node in \text{I}_2.

Next, we discuss how the reduction order defined here directs variable substitution during elimination, as the interval order does in the Allen/Cocke algorithm and the parse does in the Hecht/Ullman algorithm.

5.2. T3 Transformations and Elimination

The basic elimination step of Tarjan interval analysis is the application of a T3 transformation, which corresponds to the calculation of the reduced equation of a variable as a linear function of its interval head variable.

A T3 transformation is the composition of a T1 transformation and a T2 transformation, that is T3 = T2 \circ T1 (see Section 4.2). Figure 21 illustrates the transformation T3(u,w,E_1,E_2). Edge (u,w) in the partially collapsed flow graph represents a set of original flow graph edges E_2. Likewise, edge (w,w) represents a set of original
Figure 20: Example of Tarjan Interval Finding Algorithm

Flow Graph

<table>
<thead>
<tr>
<th>Node</th>
<th>HIGHPT(Node)</th>
<th>SNUMBER(Node)</th>
<th>Intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( l_1 = {1,{2,3,4,{5,6,8}},7,{9,10,12}} )</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>( l_2 = {2,3,4,{5,6,8}},7,{9,10,12}}</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>9</td>
<td>( l_5 = {5,6,8} )</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td>( l_9 = {9,10,12} )</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Reduction Order: \( \{10,12,6,8,3,9,4,5,7,2,11,1\} \)
flow graph edges $E_1$. The graphical interpretation of the $T3(u,w,E_1,E_2)$ transformation is the merger of node $w$ into $l_u$ (represented in the partially collapsed flow graph by $u$).

![Figure 21: $T3(u,w,E_1,E_2) = T2(u,w,E_2) \cdot T1(w,E_1)$](image)

The equation manipulations corresponding to the $T3$ transformation in Figure 21 first apply a loop breaking rule to eliminate any self-dependency in the equation for $X_w$ and then apply a sequence of substitution transformations to eliminate the variable $X_w$ from the current system of partially reduced equations. The former occurs only when $w$ itself is an interval head node; the latter is accomplished by substitution of the right-hand side of the reduced equation for $X_w$:

$$X_w = a \cdot n \cdot X_u \cdot u \cdot b$$

where $a, b$ are constants, for any occurrence of $X_w$ in the system of equations.

```c
/* H is queue of interval head nodes, ordered in reduction order*/
while H not null do
    Destructively select first element from H, h;
    for a reduction order pass through all nodes n in l_h do
        Apply $T3(h,n,E_1,E_2)$;
    endfor;
endwhile;
Apply $T1(source,E_1)$ if necessary;
```

Figure 22: Reduction Order Variable Substitution

Figure 22 presents the Tarjan algorithm elimination phase. The variables have $T3$ transformations applied to them in reduction order within each interval. Intervals are also processed in reduction order, for nested intervals, in innermost to outermost order. The final $T1(source,E_1)$ transformation in Figure 22 handles the case of those flow graphs...
where the source node is the outermost loop head\textsuperscript{14}.

The actual calculations in Tarjan interval analysis are performed somewhat differently than in our interpretation. When T3(u,w,E,E) is performed, the substitution of a linear function of \( X_u \) for an \( X_w \) term is accomplished only in the equations of those variables which precede \( X_w \) in the reduction order. For other dependencies on \( X_w \) the variable substitution is delayed, much as in the Hacht/Ullman algorithm. The delayed substitution takes place when the reduced equation for the variable dependent on \( X_w \) is calculated.

For example, if there is a back edge \((w,v)\) in the flow graph such that \( w = l_E \ldots l_{E_v} \) then \((w,v)\) will be in the \( E_1 \) set of the T3 transformation of \( v \). The substitution for the \( X_w \) term in the equation for \( X_v \) will not occur when a T3 transformation is applied to \( w \). It will be delayed until the reduced equation for \( X_v \) is calculated; the actual substitution will occur in step ii.a of the Tarjan algorithm in Section 5.4. At the time the delayed substitution is performed, the current reduced equation for \( X_w \) is a linear function of \( X_v \). It is possible that other delayed substitutions involve the same inter-interval control flow paths on the flow graph from \( v \) to \( u \) or subpaths of these. These common subpaths correspond to the common inter-regional paths referred to in Section 4.2.

5.3. Propagation

The propagation phase of Tarjan interval analysis is fairly straightforward, similar to the Allen/Cocke algorithm. The initial conditions of the data flow problem expressed in the original equation for \( X_{\text{source}} \) enable us to solve the reduced equation for \( X_{\text{source}} \). This solution is substituted into the reduced equations of all variables dependent upon \( X_{\text{source}} \). Some of these variables are interval head variables. Solutions for variables in an interval are obtained by substituting the interval head variable solution into the reduced equations of variables in that interval. This process continues until all solutions are obtained.

5.4. Algorithm Statement

Tarjan interval analysis consists of the same three phases as Allen/Cocke interval analysis: interval finding, elimination and propagation. Interval finding requires establishment of a reduction order on the flow graph using a DFST (see Section 5.1). The elimination

\textsuperscript{14}These graphs violate our definition of source node, but they can be accommodated.
phase performs coefficient/constant substitutions in the equations in reduction order, within intervals which themselves are ordered in reduction order. At the end of elimination, the reduced equation for each variable $X_n$, is a linear function of an interval head variable $X_{h_i}$, where $\{-1\} \subseteq I_i \subseteq \{h\}$ for $k \geq 0$. The propagation phase obtains the solution for the source variable and performs back substitutions in the reduced equations.
Model of Tarjan Interval Analysis Algorithm

Interval Finding:
i. Using a DFST construction, find a reduction order and the intervals of the flow graph (see Section 5.1).

Elimination Phase:
ii. In reduction order, for each interval \( I_m \) and each \( m \in I_m \), perform \( T3(m,n,E_1,E_2) \):
   a. For each edge \((z,n)\in E_1\), apply a substitution transformation, replacing the \( x_z \) term in the equation for \( x_n \) by the right-hand side of the reduced equation for \( x_z \). Apply the relevant loop breaking rule (see Section 2).
   b. For each edge \((w,n)\in E_2\), apply a substitution transformation replacing the \( x_w \) term in the equation for \( x_n \) by the right-hand side of the reduced equation for \( x_w \).

The following reduced equation for \( x_n \) is obtained:

\[
x_n = a \cap x_m \cup b
\]  \hspace{1cm} (15)

for \( a, b \) constants.

c. By a sequence of substitution transformations, substitute the right-hand side of equation (15) for an \( x_n \) term wherever necessary, changing any dependence on \( x_n \) to a dependence on \( x_m \) for the equations of nodes which precede \( n \) in reduction order. Delay all other substitutions for \( x_n \).

Propagation Phase:
iii. Determine the solution of \( x_{\text{source}} \) (Note: If \( x_{\text{source}} \) is in a loop, apply a loop
breaking rule to the reduced equation for \( x_{\text{source}} \) Let \( S = \{x_{\text{source}}\} \).

iv. Iterate until all solutions are obtained:

For each unsolved variable \( x_n \) whose reduced equation is a linear function of \( x_k \in S \), substitute the value of the solution of \( x_k \) into that equation, obtaining the value of \( x_n \). Add \( x_n \) to \( S \).
5.5. Comparison to the Allen/Cooke and Hecht/Ullman Algorithms

In this section, we compare the definitions of intervals used in the Allen/Cocke and Tarjan algorithms, using an interval dependency tree or id-tree to illustrate the source of common substitution factors used by the Tarjan and Hecht/Ullman algorithms. We then show the solution of the REACH example of Figure 16 by the Tarjan algorithm and comment on the data structures used by the Tarjan and the Hecht/Ullman algorithms to remember common factors.

Figure 23 illustrates some differences between Tarjan and Allen/Cocke intervals, defined on the same flow graph. Again DFST edges are solid lines, back edges are dashed lines and cross edges are dotted lines. Since Tarjan intervals correspond to the loop structure of the flow graph, nested loops in Figure 23 appear as explicit nested intervals. For example, loop 3 is nested within loop 2 which is nested within loop 1 and HIGHP(3)=2, HIGHP(2)=1. The same information is found by the Allen/Cocke algorithm, but it must be determined by examination of the derived sequence, so it is less explicit.

In both algorithms, the set of interval head nodes of a reducible flow graph is unique, depending only on the underlying flow graph, not on its representation. We showed this for Allen/Cocke intervals in Section 3.1; for Tarjan intervals it follows since:

i. a flow graph is reducible if and only if it has a unique decomposition into a set of back edges, plus a DAG which consists of a DFST and all of its cross and forward edges

ii. the set of back edges of a reducible flow graph is the set of backwards arcs of any DFST on that flow graph [Hecht 77].

Both an interval order and a reduction order of nodes impose an ancestor-first order within an interval; each is non-unique. An interval order partially depends upon the order of the edges in the representation of the flow graph. A reduction order depends upon the DFST constructed starting at the source node, which is similarly dependent on the graph representation.

To further illustrate the common substitution sequences found by the Tarjan algorithm, we define an interval dependency tree or id-tree to be a directed tree rooted in the source node, whose nodes are the nodes of the flow graph and whose edges reflect the interval structure of the flow graph. A directed edge (h,y) in the id-tree
Figure 23: Comparison of Tarjan and Allen/Cocke Intervals

Tarjan Intervals:

\[ l_1 = \{1, \{2,\{3,4\}\}, 5, \{6,7\}\} \]
\[ l_0 = \{6,7\} \]
\[ l_2 = \{2,\{3,4\}\} \]
\[ l_3 = \{3,4\} \]

Reduction Order: \{'1', '4', '3', '2', '5', '6', '7', '1}'

Allen/Cocke Intervals:

\[ G^1 \quad l_1 = \{1\} \]
\[ l_2 = \{2\} \]
\[ l_3 = \{3,4\} \]
\[ l_0 = \{6,7\} \]

\[ G^2 \quad l_1 = \{1,6\} \]
\[ l_2 = \{2,3\} \]
\[ l_3 = \{5\} \]
\[ l_0 = \{5\} \]

\[ G^3 \quad l_1 = \{1,2,5\} \]

\[ G^4 \quad l_1 = \{1\} \]
signifies that \( h \) is an interval head node in the flow graph and \( \forall i \neq h \). Clearly, the id-tree for a reducible flow graph is unique because the set of interval head nodes of the flow graph is unique. All internal nodes in the tree are interval head nodes.

Each node \( y \) in the id-tree has associated with it the reduced equation for \( X_y \) as a linear function of \( X_h \) for \( h \) the parent of \( y \) in the id-tree. A path \(<p_1, \ldots, p_k>\) in the id-tree represents a set of inter-interval paths in the dependency graph. It follows from the association of a reduced equation with each node in the id-tree that such paths represent variable substitution sequences in the system of equations for the data flow problem. If we traverse the reverse id-tree path \(<p_k, \ldots, p_1>\), successively substituting the right-hand side of the reduced equation for \( X_{p_{i-1}} \) for the \( X_{p_{i-1}} \) term in the equation for \( X_{p_i} \) for \( i=k, k-1, \ldots, 3 \) we obtain an equation for \( X_{p_1} \) as a linear function of \( X_{p_1} \). If subpaths are common to two or more equation calculations, Tarjan interval analysis will accrue savings over the straightforward approach of Allen/Cocke interval analysis, by identifying these common factors and using delayed substitutions to take advantage of them.

![Figure 24: Flow Graph and its Id-Tree](image)

Our first example of common substitution factors involves two transfers out of a nested loop as shown in Figure 24. Edges (4,5) and (4,7) appear in the T3 transformations which add nodes 5 and 7 respectively to interval \( l_1 \). The inter-interval path \(<1, 2, 3>\) in the id-tree is shared by both reduced equation calculations, as \( X_4 \) appears in the equation for \( X_5 \) and \( X_7 \). Again, here the substitution path in the flow graph is represented by the identical path \(<1, 2, 3>\) in the id-tree. In the Tarjan algorithm, the reduced equation for \( X_4 \) as a linear function of \( X_1 \) is calculated once and used in a delayed substitution in both in
the X₅ and X₇ equations. In Allen/Cocke interval analysis, the X₄ term in the equations for X₅ and X₇ would be transformed in turn into a term in X₃, X₂ and X₁ by explicit successive substitutions in each equation separately.

![Diagram](image-url)

**Figure 25:** Graph from Figure 16 and its Interval Dependency Tree

In our second example, we assume there are back edges in the flow graph which share variable substitution subpaths. Figure 25 shows Ullman's worst case graph for Allen/Cocke interval analysis for the case of n=10 (see Figure 16) and its id-tree [Ullman 73]. The data flow effect of back edge (9,2) is calculated when T3(1,2,(9,2),{(1,2)}) is performed. A corresponding substitution path in the flow graph is <2 3 4 8 9 2>. Likewise the data flow effect of (10,1) is calculated when T1(1,{(10,1)}) is performed; the corresponding substitution path in the flow graph is <1 2 3 4 8 10 1>. These two calculations share inter-interval subpath <2 3 4> in the id-tree. By coincidence, <2 3 4> in the id-tree represents the same path <2 3 4> in the flow graph. Thus, if the coefficient/constant substitutions which obtain X₄ as a linear function of X₂ are performed once, they can be used in two different variable substitutions: for the X₉ term in the reduced equation for X₂ and for the X₁₀ term in the reduced equation for X₁. Figure 26 shows the simplified reduced equations calculated at

---

15 Recall from Figure 22 that this final T1 transformation must be applied when the source node is in a loop in the flow graph.
each step of the Tarjan algorithm. After \( T_3(2,8,\{4,8\}) \) the shared computation of \( X_4 \) as a linear function of \( X_2 \) is calculated. In Section 4.5 we present the Allen/Cocke interval analysis algorithm applied to this example; comparison will show the duplicate variable substitutions avoided.

Both the Tarjan and Hecht/Ullman algorithms use an auxiliary data structure, a tree, to store the common factors information. The Tarjan path compressed tree, whose nodes store the partially reduced equations of the corresponding variables, is easier to understand and implement than the height-balanced 2–3 tree of the Hecht/Ullman algorithm, which encodes the factored reduced equations as edge labels in the tree.

6. GRAHAM/WEGMAN ANALYSIS

The Graham/Wegman algorithm is very similar to the two previously defined improvements on Allen/Cocke interval analysis. The groupings of the variables used by the Graham/Wegman algorithms are called S-sets. The elimination process is described using graph transformations similar to those of the Hecht/Ullman algorithm. The Graham/Wegman algorithm substitutes for each term in the system individually as in the Hecht/Ullman algorithm rather than substituting for the entire right-hand side of an equation at once as in the Allen/Cocke algorithm. The specified substitution order for terms in the equations results in common substitution sequences only being performed once. This algorithm makes explicit the delay in substitutions utilized in the Tarjan and Hecht/Ullman algorithms. A transformed version of the original flow graph is used to remember substitution sequences.

In this section we discuss the S-sets and the node order which governs substitution in the equations and describe the graph transformations \( \{S_1,S_2,S_3\} \), their graphical interpretations and corresponding equation manipulations. We present the formal algorithm and compare it with all the algorithms previously described. Finally, we illustrate the algorithm in the example in Figure 16.

6.1. S-sets and St. S2, S3 Transformations

The Graham/Wegman algorithm assumes a data flow problem is defined by a system of equations of the form of equation 6. The Graham/Wegman algorithm defines a node numbering num on the dependency graph of a system of equations for a data flow problem using a depth-first spanning tree (i.e., DFST). The order guarantees that for any
Figure 28: Tarjan Interval Analysis on REACH Problem for Figure 16

Reduction Order \{5,6,4,7,3,8,9,2,10,1\}

Intervals

\[ I_1 = \{1, \{2,\{3,\{4,5,6\}\}\}\} \cup 8,9,10 \}\]
\[ I_2 = \{2,\{3,\{4,5,6\}\}\} \cup 8,9 \}\]
\[ I_3 = \{3,\{4,5,6\}\} \]
\[ I_4 = \{4,5,6\} \]

Initially

\[ X_1 = p_{10} \cap X_{10} \cup d_{10} \]
\[ X_2 = p_{1} \cap X_{1} \cup p_{9} \cap X_{9} \cup d_{1} \cup d_{9} \]
\[ X_3 = p_{2} \cap X_{2} \cup p_{8} \cap X_{8} \cup d_{2} \cup d_{7} \]
\[ X_4 = p_{3} \cap X_{3} \cup p_{7} \cap X_{7} \cup d_{3} \cup d_{6} \]
\[ X_5 = p_{4} \cap X_{4} \cup d_{4} \]
\[ X_6 = X_7 = p_{5} \cap X_{5} \cup d_{5} \]
\[ X_8 = p_{6} \cap X_{6} \cup d_{6} \]
\[ X_9 = X_{10} = p_{8} \cap X_{8} \cup d_{8} \]

Let
\[ d_{i_{k}} = p_{n}\ldots p_{k} \cap d_{i_{k}} \cup p_{n}\ldots p_{k} \cap d_{i_{k}} \ldots d_{i_{k}} \]
\[ p_{i_{k}} = p_{n}\ldots p_{k} \cap d_{i_{k}} \]

After T3(4,5,\{4,5\}), no change in equations

After T3(4,6,\{4,6\})
\[ X_6 = p_{5,4} \cap X_{4} \cup d_{5,4} \]

After T3(3,4,\{3,4\},\{3,4\})
\[ X_4 = p_{3} \cap X_{3} \cup d_{3} \cup d_{5} \cup d_{4} \]
After $T3(3.7,\{(5,7)\})$
\\
$x_5 = p_4 \cap x_3 \cup d_4 \cup d_6 \cup d_5$
$x_7 = p_5 \cap x_3 \cup d_4 \cup d_5 \cup d_6$ \\

After $T3(2.3,\{(7,3)\},\{(2,3)\})$
\\
$x_3 = p_2 \cap x_2 \cup d_2 \cup d_7 \cup d_5 \cup d_4 \cup d_6$ \\

After $T3(2.8,\{(4,8)\})$
\\
$x_4 = p_3 \cap x_2 \cup d_4 \cup d_3 \cup d_7 \cup d_5 \cup d_4 \cup d_6$ \\
$x_8 = p_4 \cap x_2 \cup d_4 \cup d_3 \cup d_7 \cup d_5 \cup d_4 \cup d_6$ \\

After $T3(2.9,\{(8,9)\})$
\\
$x_9 = p_8 \cap x_2 \cup d_8 \cup d_4 \cup d_6 \cup d_5 \cup d_4 \cup d_6$ \\

After $T3(1.2,\{(9,2)\},\{(1,2)\})$
\\
$x_2 = p_1 \cap x_1 \cup d_1 \cup d_9 \cup d_4 \cup d_6 \cup d_5 \cup d_4 \cup d_6$ \\

After $T3(1,10,\{(8,10)\})$
\\
$x_8 = p_8 \cap x_1 \cup d_4 \cup d_3 \cup d_2 \cup d_9 \cup d_4 \cup d_6 \cup d_5 \cup d_4 \cup d_6 \cup d_5$ \\
$x_{10} = p_8 \cap x_1 \cup d_4 \cup d_3 \cup d_2 \cup d_9 \cup d_4 \cup d_6 \cup d_5 \cup d_4 \cup d_6 \cup d_5$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
\\
$x_1 = d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2 \cup d_{10} \cup d_8 \cup d_4 \cup d_3 \cup d_2$ \\

After $T3(1,10,\{(10,10)\})$
edge \((x,y)\), \(\text{num}(x) > \text{num}(y)\) if it is a back arc in the flow graph; otherwise \(\text{num}(x) < \text{num}(y)\).

In deriving this node order, the algorithm partitions the variables into non-disjoint sets called \(S\)-sets. Back arc target nodes are \(S\)-set entry nodes. The \(S\)-set headed by node \(h\) is defined by starting at \(h\) and following, in their reverse direction, paths in the flow graph which end in a back arc to \(h\). \(S\)-sets are analogous to Tarjan intervals; they are strongly connected regions of the flow graph. However, not all nodes in an \(S\)-set are collapsed into the \(S\)-set entry node, as they are in Tarjan interval analysis: nodes in the \(S\)-set which still have corresponding terms in the system of equations after the \(S\)-set is processed remain in the derived flow graph. Thus the Graham/Wegman algorithm makes explicit the delayed substitutions in the system of equations.

Substitutions for terms in the equations occur as follows. \(S\)-sets are considered in reverse \(\text{num}\) order of their entry nodes, ensuring that inner loops are processed before outer loops. Within an \(S\)-set, variables are processed in \(\text{num}\) order; therefore, when a variable is processed it will always have a unique parent variable in the \(S\)-set.

When a variable \(X_n\) is processed, a loop breaking rule is applied to the equation for \(X_n\) and a sequence of substitution transformations applied to the equations of its descendants in the \(S\)-set. Descendants which are not in the same \(S\)-set as \(X_n\) represent delayed substitutions. Therefore, unlike Tarjan intervals, \(S\)-sets are not collapsed to one variable after being processed, since there still may be dependencies on variables in the \(S\)-set in the system of equations. Only when all dependence on a variable is removed from the system of equations is that variable also removed. The final reduced equation of a variable is a linear function of the entry variable of the outermost \(S\)-set containing that variable (see Section 6.4).

The substitutions of the Graham/Wegman algorithm are described in terms of \([S1,S2,S3]\), the three graph transformations described below. As in the Hecht/Ullman algorithm, each transformation has a corresponding sequence of equation manipulations. Transformations \(S1\) and \(S2\) are applied to the flow graph until a final flow graph is obtained, while transformation \(S3\) is a technical device necessary to collapse the outermost \(S\)-set of the flow graph if the source node does not lie on a cycle. When the original flow graph is reducible, \(S3\) results in a final graph of one node [Wegman 81]. All three transformations can only be applied to a node with a unique parent. They are illustrated in Figure 27.
Figure 27: Graham/Wegman S1,S2,S3 Transformations

(i.) $S_1(u,v,(v,v))$

(ii.) $S_2(u,v,w,(v,w))$

(iii.) $S_2(u,v,z,(v,z))$

(iv.) $S_3(u,v,(u,v))$
S1 and S2 are closely related to T1 and T2, respectively, of the Hecht/Ullman algorithm. S1 and T1 are approximately equivalent; T1 does not require a unique parent node for its application. A set of k S2 transformations are the equivalent of a T2 transformation on a node with k descendants in the same S-set. S2(u,v,w,(v,w)) does not eliminate X_v from the system of equations (see Figure 27); however, the sequence S2(u,v,w,(v,w)), S2(u,v,x,(v,x)), S2(u,v,y,(v,y)) will accomplish this when applied to S-set {u,v,w,x,y}. Thus, only when the node in the S2 transformation has a unique descendant will that variable be eliminated from the system of equations by the S2 transformation. S3 is a degenerate S2 transformation, which eliminates a node without descendants, a variable which does not appear in a right-hand side of any equation in the system.

The similarities between \{S1,S2,S3\} and \{T1,T2\} carry over to their interpretation as equation manipulations. The S1 transformation is a loop breaking rule as was the T1 transformation (see Section 4.2). S1(u,v,(v,v)) removes the self-dependence on X_v from the equation for X_v; that is:

\[ X_v = a \cap X_u u b \cap X_v u c \]

becomes:

\[ X_v = a \cap X_u u c \]

The S2(u,v,w,(v,w)) transformation is a substitution transformation which corresponds to substitution of the right-hand side of the equation for X_v into the equation of X_w; that is, if:

\[ X_v = a \cap X_u u c \]

then:

\[ X_w = a \cap X_v u d \]

becomes:

\[ X_w = a \cap a \cap X_u u e n c u d \]
removing dependence on \( X_v \) from the equation for \( X_w \). If \( X_v \) appears only in the \( X_w \) equation (i.e., \( v \) has only one descendant node \( w \)) then this transformation removes \( X_v \) from the system of equations.

6.2. Propagation

The propagation phase of this algorithm resembles that of Tarjan interval analysis. We obtain a solution for the final S-set entry variable. We substitute this solution into all reduced equations dependent upon it, obtain their corresponding solutions, and iterate this process until solutions for all variables are obtained.

6.3. Algorithm Statement

Graham/Wegman analysis consists of three phases: S-set finding, elimination and propagation. Forming S-sets and establishing a num node order require the use of a DFST (see Section 6.1). The elimination phase performs coefficient/constant substitutions in the equations of variables in num order within S-sets considered in reverse num order of their entry variables. The propagation phase obtains a solution for the source variable and performs the back substitutions in the reduced equations.

The Graham/Wegman algorithm can be transformed to handle irreducible dependency graphs also. The irreducibility is discernible during S-set construction [Wegman 81]. If, while we are performing a reverse traversal of all paths ending in a back arc to \( h \), we encounter a node \( x \) such that num(x)<num(h), then we know we have a multiple entry loop by the properties of num, and therefore an irreducible dependency graph [Hecht 77]. This situation can be handled by generalizing the definition of S-set to allow multiple entry regions and the transformations \{S1,S2,S3\} to \{GS1,GS2,GS3\} which handle nodes with multiple parents.

To form the generalized S-sets use the constructive definition of S-sets given in Section 6.1. While performing a reverse traversal of all paths ending in a back arc to \( h \), if a node \( x \) such that num(x)<num(h) is encountered, \( x \) is simply not added into the S-set being constructed. If the node immediately previous to \( x \) on the reverse traversal is \( z \), then during elimination, when node \( z \) is processed it will have 2 parent nodes, one within the S-set and one not. We must use the generalized transformations shown in Figure 28, to accommodate the possibility of multiple parents for S-set nodes. The meaning of these transformations \{GS1,GS2,GS3\} in terms of the corresponding equation manipulations is
similar to that of \( \{S1,S2,S3\} \) (see Section 6.1).

Application of \( \text{GS1}(P,v,(v,v)) \) is a loop breaking rule for the equation for \( X_v \). Thus using the nodes in Figure 28:

\[
X_v = a \cap X_v \cup b \cap X_u \cup c \cap X_w \cup d
\]

becomes:

\[
X_v = b \cap X_u \cup c \cap X_w \cup d
\]

\( \text{GS2}(P,v,r,(v,v)) \) is a substitution transformation which corresponds to the substitution of the right-hand side of the equation for \( X_v \), which is a linear function of the \( \{X_p\}, p \in P \), for the \( X_v \) term in the equation of \( X_r \). That is, if:

\[
X_v = a \cap X_u \cup b \cap X_w \cup c
\]

then:

\[
X_r = a \cap X_v \cup d
\]

becomes:

\[
X_r = (a \cap a \cap X_u) \cup (e \cap b \cap X_w) \cup (e \cap c) \cup d
\]

If, as in Figure 28(iii), \( X_v \) appears only in the \( X_r \) equation then this transformation removes \( X_v \) from the system of equations. \( \text{GS3} \) follows similarly from \( S3 \).

Thus, the Graham/Wegman algorithm can easily be adapted to handle irreducible dependency graphs. Since the irreducibility is discovered during the first phase of the algorithm, there is no question as to which set of transformations is appropriate.
Figure 28: Graham/Wegman GS1, GS2, GS3 Transformations

(i.) GS1(P, v, (v, v)), P = \{u, w\}

(ii.) GS2(P, v, r, (v, r)), P = \{u, w\}

(iii.) GS2(P, v, z, (v, z)), P = \{u, w\}

(iv.) GS3(P, v, (u, v)), P = \{u, w\}
Model of Graham/Wegman Analysis

S-set Finding:

i. Using a DFST construction, identify back arcs of the dependency graph of the system of equations. Form S-sets. Establish the num node order on the DFST (see Section 6.1).

Elimination Phase:

ii. Process S-sets in reverse num order of their entry nodes. Within an S-set process each node n in num order.
   a. If necessary, perform S1(n,n,n). Apply a loop breaking rule to the equation for $X_n$.
   b. For each descendant z of n in the S-set, perform the substitution transformation S2(m,n,z,n,z). Substitute the right-hand side of the reduced equation for $X_n$ for the $X_n$ term in the equation for $X_z$. (If $X_n$ appears in only one place in the system, this eliminates $X_n$.)

iii. If necessary, use S3 transformations to reduce the final graph to one node.

Propagation Phase:

iv. Determine the solution of $X_{source}$. (Note: if $X_{source}$ is in a loop, apply a loop breaking rule to the reduced equation for $X_{source}$.) Let $S = \{X_{source}\}$.

v. Iterate until all solutions are obtained:

For each unsolved variable $X_n$ whose reduced equation is a linear function of $X_k \in S$, substitute the value of the solution of $X_k$ into that equation, obtaining the value of $X_n$. Add $X_n$ to $S$. 

6.4. Comparisons to the Allen/Cocke, Hecht/Ullman, and Tarjan Algorithms

The Graham/Wegman algorithm is closely related to Tarjan interval analysis. Common substitution sequences in the system of equations are recognized and used to avoid duplicate calculations. The substitution sequences are shown explicitly in the Graham/Wegman algorithm through the node listing, rather than as a node order as in the Tarjan algorithm. The delayed substitutions are represented explicitly in the flow graph by not collapsing S-sets to one node. Substitutions are remembered in a transformed version of the original flow graph rather than the 2-3 tree of the Hecht/Ullman algorithm or the path compressed tree of the Tarjan algorithm. The Graham/Wegman algorithm is the only one of the three improvements on Allen/Cocke interval analysis which handles irreducible flow graphs gracefully by accommodating such graphs rather than transforming them to eliminate the irreducibility or not handling them at all.

In Figure 29, as with the other algorithms, the Graham/Wegman algorithm is applied to the example of Figure 18. We show the four S-sets corresponding to the flow graph and the S1 and S2 transformations of them. In Figure 30 we define a REACH problem on this flow graph with equations of the form of equation 6, which facilitates comparison with Figure 26. The solid lines are edges within the S-sets; the dashed lines are flow graph edges not within the S-set. We give the equation transformations corresponding to the S1 and S2 transformations on the S-sets. Because of the ordering of the substitutions, calculations along the inter-regional substitution path <2 3 4> are only performed once and then used in the equations of $X_8$, $X_9$, and $X_{10}$.

The Tarjan and Graham/Wegman variable substitution orders are similar; in the equation transformations in Figures 26 and 30, the order is the same. The Tarjan algorithm elimination is described in terms of a linear node order (reduction order), delayed calculations, and a path compressed tree for keeping track of delayed substitutions. In the Graham/Wegman algorithm, the node listing is longer, with multiple appearances of variables representing substitutions for individual terms in that variable; the equation calculations are kept in the transformed version of the original flow graph. The Graham/Wegman algorithm makes explicit the delayed calculations of the Tarjan algorithm. In Figure 30 the Graham/Wegman node listing is:

---

No S3 transformation is needed here because the source node is the entry node of the outermost cycle in the flow graph.
Figure 29: S-sets Obtained in Example of Figure 16

First S-set

Second S-set

Third S-set

Fourth S-set
Figure 30: Graham/Wegman Algorithm on REACH Problem from Figure 16

Initially

\[ X_1 = p_{10} \cap X_{10} \cup d_{10} \]
\[ X_2 = p_1 \cap X_1 \cup p_9 \cap X_9 \cup d_1 \cup d_9 \]
\[ X_3 = p_2 \cap X_2 \cup p_7 \cap X_7 \cup d_2 \cup d_7 \]
\[ X_4 = p_3 \cap X_3 \cup p_6 \cap X_6 \cup d_3 \cup d_6 \]
\[ X_5 = p_4 \cap X_4 \cup d_4 \]
\[ X_6 = X_7 = p_5 \cap X_5 \cup d_5 \]
\[ X_8 = p_4 \cap X_4 \cup d_4 \]
\[ X_9 = X_{10} = p_8 \cap X_8 \cup d_8 \]

Let \( d_{i-1} \cdots i-k = p_{n-\cdots np_{k-1}k-1} \cap d_{i-1} \cdots i-k-1 \)

\[ p_{i-1} \cdots i_k = p_{n-\cdots np_{k-1}k-1} \]

\[ S-set = \{4,5,6\} \]

After S2(4,5,6,(5,6))
\[ X_6 = p_{5,4} \cap X_4 \cup d_{5,4} \]

After S2(4,6,4,(6,4)), eliminating \( X_6 \)
\[ X_4 = p_3 \cap X_3 \cup p_{6,5} \cap X_5 \cup d_{6,5} \cup d_3 \]

\[ S-set = \{3,4,5,7\} \]

After S1(4,(4,4))
\[ X_4 = p_3 \cap X_3 \cup d_{6,5} \cup d_3 \]

After S2(3,4,5,(4,5))
\[ X_5 = p_{4,3} \cap X_3 \cup d_{4,6} \cup d_{4,3} \]

After S2(3,5,7,(5,7)), eliminating \( X_5 \)
\[ X_7 = p_{5,4} \cap X_3 \cup d_{5,4} \cup d_{5,4} \]

After S2(3,7,3,(7,3)), eliminating \( X_7 \)
\[ X_3 = p_2 \cap X_2 \cup p_{7,5} \cap X_5 \cup d_{7,5} \cup d_{7,5} \cup d_{7,5} \cup d_2 \]
Figure 30, continued

$S$-set = \{2, 3, 4, 8, 9\}

After $S1(3, (3,3))$

$X_3 = p_2 \cap X_2 \cup d_7 \cup 4 \cup 3 \cup d_5 \cup 6 \cup d_2$

After $S2(2,3,4,(3,4)), \text{ eliminating } X_3$

$X_4 = p_3 \cap X_2 \cup d_3 \cup 2 \cup d_3 \cup 7 \cup 5 \cup 4 \cup d_6 \cup 5 \cup 4$

After $S2(2,4,8,(4,8)), \text{ eliminating } X_4$

$X_8 = p_4 \cap X_2 \cup d_4 \cup 3 \cup 2 \cup d_4 \cup 3 \cup 7 \cup 5 \cup d_4 \cup 8 \cup 5$

After $S2(2,8,3,(8,9))$

$X_9 = p_8 \cap X_2 \cup d_8 \cup 4 \cup 3 \cup 2 \cup d_8 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_8 \cup 4 \cup 5 \cup 5$

After $S2(2,9,2,(9,2)), \text{ eliminating } X_9$

$X_2 = p_1 \cap X_1 \cup p_9 \cup 8 \cup 4 \cup 3 \cup 2 \cap X_2 \cup d_1 \cup d_9 \cup 8 \cup 4 \cup 3 \cup 2 \cup d_9 \cup 8 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_9 \cup 8 \cup 4 \cup 6 \cup 5$

$S$-set = \{1, 2, 8, 10\}

After $S1(2, (2,2))$

$X_2 = p_1 \cap X_1 \cup d_1 \cup d_9 \cup 8 \cup 4 \cup 3 \cup 2 \cup d_9 \cup 8 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_9 \cup 8 \cup 4 \cup 6 \cup 5$

After $S2(1,2,8,(2,8)), \text{ eliminating } X_2$

$X_8 = p_4 \cap X_1 \cup d_4 \cup 3 \cup 2 \cup 9 \cup 8 \cup d_4 \cup 3 \cup 7 \cup 5 \cup d_4 \cup 8 \cup 5 \cup d_4 \cup 3 \cup 2 \cup 1$

After $S2(1,8,10,(8,10)), \text{ eliminating } X_8$

$X_{10} = p_9 \cap X_1 \cup d_9 \cup 4 \cup 3 \cup 2 \cup 9 \cup d_9 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_9 \cup 8 \cup 4 \cup 6 \cup 5 \cup d_9 \cup 4 \cup 3 \cup 2 \cup 1$

After $S2(1,10,1,(10,1)), \text{ eliminating } X_{10}$

$X_1 = p_{10} \cap X_1 \cup d_{10} \cup 8 \cup 4 \cup 3 \cup 2 \cup 9 \cup d_{10} \cup 8 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_{10} \cup 8 \cup 4 \cup 6 \cup 5 \cup d_{10} \cup 8 \cup 4 \cup 3 \cup 2 \cup 1$

After $S1(1, (1,1))$

$X_1 = d_{10} \cup 8 \cup 4 \cup 3 \cup 2 \cup 9 \cup d_{10} \cup 8 \cup 4 \cup 3 \cup 7 \cup 5 \cup d_{10} \cup 8 \cup 4 \cup 6 \cup 5 \cup d_{10} \cup 8 \cup 4 \cup 3 \cup 2 \cup 1$
{5,6,4,5,7,3,4,8,9,2,10,1}

whereas reduction order is:

{5,6,4,7,3,8,9,2,10,1}

7. SUMMARY

Our systems of equations models of elimination algorithms enable us to compare them and contrast their sources of worst case complexity improvement. The original algorithm. Allen/Cocke interval analysis, establishes a natural partition of the variables and a variable order on each of a sequence of systems which, when used to order the equations, results in a highly structured coefficient matrix facilitating the equation reduction process. The other three algorithms: Hecht/Ullman T1–T2 analysis, Tarjan interval analysis, and Graham/Wegman analysis, avoid repeated calculations of common substitution sequences in the equations by delaying certain computations. Hecht/Ullman T1–T2 analysis uses a nondeterministic substitution order for terms in the equations; substitutions performed are recorded in a height-balanced 2–3 tree to take advantage of possible common factors in subsequent calculations. Tarjan interval analysis establishes a linear variable order and eliminates variables from the system of equations in that order, delaying some calculations; a path compressed tree is used to remember sequences of reduced equations for these delayed calculations. Graham/Wegman analysis establishes an order for substitution for each term in the system. This order avoids duplication of common substitution sequence calculations. The Graham/Wegman algorithm uses a transformed version of the original flow graph to remember previous substitutions.

The best elimination algorithm in terms of worst case complexity is the Tarjan almost linear interval analysis algorithm in which the path compressed tree is balanced in a preprocessing operation. This algorithm is the best algorithm for doing a sequence of unions and finds but is not used for data flow analysis in practice [Tarjan 79]. Tarjan suggests the use of path compressed trees for ease of calculation; they insure a bound of $O(n \log n)$ [Tarjan 81a].

The four algorithms presented in this tutorial vary in their worst case complexity
bounds for reducible flow graphs as shown.\textsuperscript{17}

<table>
<thead>
<tr>
<th>Allen/Cocks</th>
<th>Hecht/Ullman</th>
<th>Tarjan</th>
<th>Graham/Wegman</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{O}(n)</td>
<td>\text{O}(n \log n)</td>
<td>\text{O}(n \alpha(n))</td>
<td>\text{O}(n \log n)</td>
</tr>
</tbody>
</table>

\text{N} is the total number of nodes in the derived sequence of graphs and is bounded by \text{n}^2. Recall from Theorem 1 that this \text{O}(n^2) bound is \text{O}(n) for many reasonable programs. The \alpha function is related to the inverse of Ackermann's function; \alpha(n)\leq 3 for all practical purposes (see Section 5.1).

All of these algorithms represent a savings on reducible flow graphs over straightforward Gaussian elimination, an \text{O}(n^3) algorithm. Both the Tarjan and Graham/Wegman algorithms identify an irreducible system of equations using an \text{O}(n \log n) algorithm for a flow graph. The Allen/Cocks or Graham/Wegman algorithms are applicable in this eventuality although their performance cannot be guaranteed to be better than the Gaussian elimination-like technique. All the algorithms can be used on a reducible flow graph.

In presenting a uniform model of these algorithms, we are able to explain their similarities and differences. All are applicable to general systems of equations with coefficient structures similar to those described here. The reducibility of the dependency graph is necessary to partition the problem into smaller, more easily solved problems. We are interested in discerning related structural properties of systems of equations which may aid in their solution; the models described here hopefully will provide intuition for improving algorithms in other problem domains.

8. ACKNOWLEDGMENTS

We owe thanks to Tom Marlowe for his diligent readings of this paper and his help in improving the presentation, and to the reviewers whose constructive criticisms aided us in our revision.

\textsuperscript{17}Recall that for a flow graph, \text{e} is \text{O}(n).
REFERENCES

[Aho 76] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley. 1976.

[Aho 77] Aho, A. V. and Ullman, J. D.
Principles of Compiler Design.
Addison-Wesley. 1977.

[Allen 71] Allen, F. E.
A Basis for Program Optimization.
in Proceedings of 1971 IFIP Congress. pages 385–390. Institute of
Electrical and Electronics Engineers, Inc., North Holland Publishing

[Allen 72] Allen, F. E. and Cocke, J.
Graph Theoretic Constructs for Program Control Flow Analysis.
Computer Science Technical Report RC 3923, IBM Thomas J. Watson

[Allen 74] Allen, F. E.
Interprocedural Data Flow Analysis.
in Proceedings of 1974 IFIP Congress. pages 399–402. Institute of
Electrical and Electronics Engineers, Inc., North Holland Publishing

[Allen 77] Allen, F. E. and Cocke, J.
A Program Data Flow Analysis Procedure.

[Allen 79] Allen, F. E.
private communication.

[Backus 57] Backus, J. W., Beener, R. J., Best, S., Goldberg, R., Halbist, L. M., Herrick,
H. L., Nilsen, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes,
R. A., and Nutt, R.
The FORTRAN Automatic Coding System
1957. Los Angeles.

[Banning 79] Banning, J.
An Efficient Way to Find the Side Effects of Procedure Calls and the
Aliases of Variables.
in Conference Record of the Sixth Annual ACM Symposium on
Principles of Programming Languages, pages 29–41. Association

[Barth 78] Barth, J. M.
A Practical Interprocedural Data Flow Analysis Algorithm.

[Burke 84] Burke, M.
An Interval Analysis Approach Toward Interprocedural Data Flow
Analysis.
Computer Science Technical Report RC 10840, IBM Thomas J. Watson
[Cocke 70] Cocks, J.
Global Common Subexpression Elimination.
In Proceedings of ACM SIGPLAN Symposium on Compiler

[Cooper 84] Cooper, K. and Kennedy, K.
Efficient Computation of Flow Insensitive Interprocedural Summary
Information.
In Proceedings of SIGPLAN ’84 Symposium on Compiler Construction,

[Farrow 75] Farrow, R., Kennedy, K. and Zucconi, L.
Graph Grammars and Global Program Data Flow Analysis.
In Proceedings of Seventeenth Annual IEEE Symposium on the
Foundations of Computer Science, pages 42–56. Institute of

[Graham 76] Graham, S. and Wegman, M.
Fast and Usually Linear Algorithm for Global Flow Analysis.

[Hecht 77] Hecht, M. S.
Flow Analysis of Computer Programs.

[Hopcroft 72] Hopcroft, J. E. and Ullman, J. D.
A n log n Algorithm for Reduction of Flow Graphs.
In Sixth Annual Princeton Conference on Information Sciences and

[Isaacson 66] Isaacson, E. and Keller, H. B.
Analysis of Numerical Methods.

[Kennedy 71] Kennedy, K.
A Global Flow Analysis Algorithm.
December, 1971.

[Kennedy 74] Kennedy, K.
Schaeffer’s Node Splitting Algorithm.
SETL Newsletter #125, February 6, 1974, Courant Institute of
Mathematical Sciences, New York University.

[Kennedy 77] Kennedy, K. and Zucconi, L.
Application of a Graph Grammar for Program Control Flow Analysis.
In Conference Record of the Fourth Annual ACM Symposium on
Principles of Programming Languages, pages 72–85. Association

[Kennedy 79] Kennedy, K.
A Survey of Data Flow Analysis Techniques.
In Muchnick, S. and Jones, N. (editor), Program Flow Analysis: Theory
[Kildall 73] Kildall, G.
A Unified Approach to Global Program Optimization.
In Conference Record of the ACM Symposium on the Principles of
Programming Languages, pages 194-206. Association for

[Knuth 68] Knuth, D. E.
Addison Wesley, Reading, Ma, 1968.

[Knuth 71] Knuth, D. E.
An Empirical Study of FORTRAN Programs.

[Paul 85] Paul, M. C.
Introduction to Algorithm Design Principles.
pre-publication manuscript.

[Reingold 77] Reingold, E. M., Nievergelt, J. and Deo, N.
Combinatorial Algorithms: Theory and Practice.

[Robinson 79] Robinson, S. K. and Torsun, I. S.
An Empirical Analysis of FORTRAN Programs.

[Ryder 74] Ryder, B. G.
The PFORT Verifier.

[Ryder 79] Ryder, B. G.
Constructing the Call Graph of a Program.
IEEE Transactions on Software Engineering SE-5(3):216-225, May,
1979.

[Ryder 82a] Ryder, B. G.
Incremental Data Flow Analysis.
In Conference Record of the Tenth Annual ACM Symposium on
Principles of Programming Languages, pages 167-176.
Association for Computing Machinery-SIGPLAN, January, 1982.

[Ryder 82b] Ryder, B. G.
Incremental Data Flow Analysis Based on a Unified Model of
Elimination Algorithms.
PhD thesis, Department of Computer Science, Rutgers University, 1982.

[Ryder 83] Ryder, B. G. and Paul, M. C.
Incremental Data Flow Analysis Algorithms.
1983.
being reviewed for publication in ACM Transactions on Programming
Languages and Systems.

[Ryder 85] Ryder, B. G.
Incremental Algorithms for Software Systems
Department of Computer Science Technical Report DCS-TR-158,
Rutgers University, July, 1985.
[Schwartz 78] Schwartz, J. T. and Sharir, M.
Tarjan's Fast Interval Finding Algorithm.
SETL Newsletter No. 204, March 3, 1978, Courant Institute of
Mathematical Sciences, New York University.

[Schwartz 79] Schwartz, J. T. and Sharir, M.
A Design for Optimizations of the Bitvectorsing Class.
Computer Science Technical Report 17, Courant Institute of
Mathematical Sciences, New York University, September, 1979.

[Sharir 77] Sharir, M.
On interprocedural Flow Analysis.
Institute of Mathematical Sciences, New York University.

[Tarjan 72] Tarjan, R. E.
Depth-first Search and Linear Graph Algorithms.

[Tarjan 74] Tarjan, R. E.
Testing Flow Graph Reducibility.

[Tarjan 79] Tarjan, R. E.
Applications of Path Compression on Balanced Trees.

[Tarjan 81a] Tarjan, R. E.
Fast Algorithms for Solving Path Problems.

[Tarjan 81b] Tarjan, R. E.
A Unified Approach to Path Problems.

[Ullman 73] Ullman, J. D.
Fast Algorithms for the Elimination of Common Subexpressions.

[Wegman 81] Wegman, M.
General and Efficient Methods for Global Code Improvement.
PhD thesis, Department of Computer Science, University of California at
Berkeley, 1981.

[Zadeck 84] Zadeck, F. K.
Incremental Data Flow Analysis in a Structured Program Editor.
In Proceedings of SIGPLAN '84 Symposium on Compiler Construction,