Report Number: WUCS-79-6

1979-07-01

Interval Maintenance of Dynamically Changing Flow Graphs

Will D. Gillett

Many compilers incorporate an optimization phase during the development of the final object code. Most optimization phases utilize a flow graph and partition it into disjoint single entry regions to perform various global flow analyses. The interval is a commonly used single entry region and interval analyses have been developed for performing standard analyses such as live variable analysis, etc. This paper presents a technique for maintaining the interval structure as the underlying flow graph is dynamically modified. The techniques presented preserve the interval order required by many interval analyses and do not require that the underlying flow graph... Read complete abstract on page 2.

Interval Maintenance of Dynamically Changing Flow Graphs

Will D. Gillett

Complete Abstract:

Many compilers incorporate an optimization phase during the development of the final object code. Most optimization phases utilize a flow graph and partition it into disjoint single entry regions to perform various global flow analyses. The interval is a commonly used single entry region and interval analyses have been developed for performing standard analyses such as live variable analysis, etc. This paper presents a technique for maintaining the interval structure as the underlying flow graph is dynamically modified. The techniques presented preserve the interval order required by many interval analyses and do not require that the underlying flow graph be reducible.

INTERVAL MAINTENANCE OF DYNAMICALLY CHANGING FLOW GRAPHS

Will D. Gillett

WUCS-79-6

Department of Computer Science

Washington University

St. Louis, Missouri 63130

July 1979

ABSTRACT

Many compilers incorporate an optimization phase during the development of the final object code. Most optimization phases utilize a flow graph and partition it into disjoint single entry regions to perform various global flow analyses. The interval is a commonly used single entry region and interval analyses have been developed for performing standard analyses such as live variable analysis, common subexpression detection, uninitialized variable analysis, etc. This paper presents a technique for maintaining the interval structure as the underlying flow graph is dynamically modified. The techniques presented preserve the interval order required by many interval analyses and do not require that the underlying flow graph be reducible.

Key Words: flow graph, derived graph, interval, reducible

CR Categories: 4.12, 4.40

1. INTRODUCTION

The techniques presented here were developed for use in an interactive FORTRAN compiler [WIL73] used as a teaching tool for beginning programmers. (Several projects have already been completed to help the student detect and correct syntax errors [TIN75] and execution errors [DAV75] and help the student produce correct solutions to problems [DAN75].) The purpose of this project [GIL76] was to detect program anomalies such as "unreferenced data", uninitialized variables, local variables in a subroutine parameter list, etc. and to help the student understand global aspects of his program. The general system approach is:

1) Create a flow graph corresponding to the student's source program.
2) Determine the interval structure [ALL70, COC70] of the flow graph.
3) Perform various interval analyses to detect program anomalies.
4) Present information to the student with possible suggestions on how to resolve a specific anomaly.
5) Allow the student to edit his already existing program based on this information.

As the student edits his program, the flow graph corresponding to his source program dynamically changes. This, of course, affects the interval structure. In a batch environment such an editing change requires submitting a completely new job to the system; thus, all 5 steps would be repeated for each submission. However, in an interactive environment, the source program (and associated data structures) is resident in the system during the entire session; thus, the previous flow graph and interval structure can be used (i.e. modified) to determine the new structure. Clearly, the flow graph of the student's program can be modified as he dynamically deletes, inserts, or modifies lines of his program (so step 1 need not be completely repeated). Recalculation of the interval

structure of the flow graph from scratch could be performed, but this can be costly ($O(n^2)$, where n is the number of nodes in the flow graph). A system design in which the interval structure is dynamically modified as the underlying flow graph changes is preferable. Thus, after the student has edited his program, the system would return to step 3 to perform analyses on the student's updated program.

The usefulness of the ability to modify an existing interval structure as the underlying flow graph is changed is not restricted to interactive systems. For instance, consider a batch system in which large utility programs (such as the accounting system) are resident to the system. When modules or components of these programs are changed, it is not necessary to reanalyse the entire program if the old interval structure has been retained (by the operating system). The new interval structure can be determined by combining the previous interval structure, with that of the newly modified components. Section 2 presents a brief background on flow graphs and intervals. Section 3 presents an overview of how structural changes in the underlying flow graph are propagated to different levels of the interval structure. Section 4 analyses in detail what happens as the underlying flow graph is modified. Section 5 presents the maintenance algorithms in detail. Section 6 gives a brief analysis of the maintenance algorithms and discusses other applications for this technique and further work to be done.

Appendix I describes the notation of the algorithmic language used to present the formation and maintenance algorithms. Appendix II presents specific implementations of algorithms that originally analyze and form the interval structure of a flow graph (some of these are used as components of the maintenance algorithms). Appendix III describes a modification to be applied to the maintenance algorithms if the editing functions that use them allow intermediate directed graphs which are not flow graphs.

2. BACKGROUND

Graph Background

A directed graph is a pair D = (N, E) where N is a set of nodes and E is a set of edges which constitute a relation on N; i.e., $E \subset (N \times N)$. An edge, (a, b), is said to leave node a and enter node b. Also, a is a predecessor of b and b is a successor of a; a is the source of the edge and b is the sink of the edge.

A path from node c to node d in a graph D = (N,E) is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ $k \geq 0$ such that $\{(n_{j-1}, n_j), 1 \leq j \leq k\} \subset E$ and $c = n_0$, $d = n_k$. A cycle is a path $(n_0, n_1, \ldots, n_k)$ such that $n_0 = n_k$, $k \geq 1$. A node, a, is said to be an ancestor of b if there is a path from a to b; in such a case, b is said to be a descendant of a.

A flow graph is a triple G = (N, E, e) such that (N, E) is a directed graph and the initial or entry node, $e \in N$, has the property that for every $n \in N$ there exists a path from e to n.

A program may be partially represented by use of a flow graph where the nodes of the graph represent statements of the program and the edges represent possible flow of control. In the remainder of this document, it is assumed that the source program has been transformed into such a flow graph. Thus, the text will reference the nodes of the flow graph instead of the statements of the program.

Interval Background

In a flow graph G = (N,E, e), an interval, $I \subset N$, is a set of nodes satisfying the conditions:

I1) There exists $h \in I$, the head of the interval, such that $\{pred(I-\{h\})\} \subset I$.

Thus, only h may possibly have predecessors outside of I.

I2) ($\forall$ i $\in$ I) (there exists $(n_0, n_1, \ldots, n_k) k \geq 0$ such that $(n_j, n_{j+1}) \subset E$ and $n_0 = h$ and $n_k = i$). In other words, every element of I is a descendant of h.

I3) I - {h} contains no cycles.

The above is an abstract definition of an interval. However, we are interested in a procedural definition for producing a maximal interval (i.e., one for which there is no interval J such that $I \subset J$ and $I \neq J$) given a head node h. The algorithm MAX_INTERVAL (see Appendix II) provides such a definition. The order of the nodes in the LIST is known as interval order. This order is not unique since the method for picking node m at S5 (see MAX_INTERVAL in Appendix II) is not well defined. However, a moments reflection reveals that:

P1) a node (except for h) will not be placed into LIST until all its predecessors are in LIST, and

P2) a node preceeds all its successors in LIST.

These two properties are useful in many forms of interval analysis and must be preserved by the maintenance algorithms.

The concept of maximal interval can be used to partition the entire flow graph into disjoint single entry subgraphs; i.e., $I_1 \cup I_2 \cup \ldots \cup I_n = N$ and $I_j \cap I_k = \emptyset$ for $j \neq k$. The general strategy employed is the following:

Whenever a newly formed interval, I, is completed, any successor of any node in I not contained in some interval already formed becomes the head of a new interval to be subsequently formed.

PARTITION (see Appendix II) presents a procedural method for such a construction.

Let the original flow graph, G = (N,E,e), be denoted by G_0. Then a new flow graph $G_1 = (N_1, E_1, e_1)$, called the derived graph of G_0, can be formed where:

- N_1 = {intervals formed by PARTITION},

- e_1 is the interval containing e, the entry node, of G_0, and
- E_1 is a set of edges defined by:

  $(m,n) \in E_1$ iff there exists a node in interval m with a successor in interval n. (The successor must, of course, be the head of interval n.) Duplicate edges are permitted and one such edge will be present for each predecessor (of the head of interval n) which exists in interval m.

G_1 satisfies the definition of a flow graph. Since G_1 is a flow graph, this interval partition process can be applied to G_1 to produce its derived graph, G_2. This process can be continued, ad infinitum; however, it normally terminates when $G_{n+1} = G_n$. G_0, G_1, ..., G_n is known as the derived sequence. $G = G_0$ is reducible [HEC72,HEC74] if G_n is a single node, and nonreducible otherwise.

A procedural method for creating the sequence of derived graphs is presented in REDUCE (see Appendix II). This algorithm assumes the existance of an external data structure capable of accepting newly created derived graphs and assumes that $G = G_0$ has been previously placed in the structure. The derived sequence is obtained by invoking REDUCE(0). It is this sequence which is used in interval analyses. Some forms of interval analysis require reducibility; however, this restriction is required of none of the algorithms presented in this paper.

3. OVERVIEW

The sequence of derived flow graphs can be viewed as a two-dimensional data structure where:

- dimension 1 (the horizontal dimension in Figure 3.1) represents the flow graph structure of each G_i, and
- dimension 2 (the vertical dimension) represents the tree structure produced by making each node of graph G_i the son of its corresponding interval node in G_{i+1}.

Although the dimensions are orthogonal in some senses, they are, of course, not independent.

Assume a specific modification is performed on G_0 ; i.e., dimension 1 at level G_0 is changed. It must be determined how this affects the rest of the data structure. This can be done by looking at only one level of the data structure at a time. If the modification to G_0 is "local enough" that dimension 1 of G_1 is not modified, then no further action is required. However, if G_1 is modified (an edge may be added or deleted, or a node may be split into two or more nodes or vice versa), then the effect of this modification (to G_1) on G_2 must be determined. This process continues up the levels of the data structure until a level, G_i , is finally found at which no modification is performed.

4. MODIFYING G_i

Consider G0(A) (i.e., G_0 of Figure 4.1 A). Assuming G0(A) corresponds to a source program, the addition of one source statement (node d) can produce a flow graph such as G0(B). Note the significant change in complexity of the interval structure as node d (with associated edges) is inserted. This change in interval structure complexity caused by simple editing funcitons is one of the reasons why the algorithms to be presented:

- perform editing by considering insertions and deletions of only a single edge at a time, and
- use recursion to propagate the effect of modification (at a given level) to higher levels of the derived graphs (as explained in Section 3).

Adding Edges To G_i

Adding a new edge (m,n) to G_i may cause one or more intervals to "split". Let j = int (m) (see appendix I), k = int(n). The insertion of (m,n) into G_i :

- will not produce a split if n is the head of interval k,
- will split interval k if $j \neq k$ and n is not the head of interval k, and
- may or may not split the interval if j = k and n is not the head of the interval.

No interval will be combined with another interval due to the addition of a new edge.

Example 4.1:

Assume we wish to add node d to G0(A) to obtain G0(B) (with edges manipulated in the order in which they are presented). The following table presents specific actions that occur as each edge is inserted or deleted one at a time. Only modifications to G_1 are presented.

Edge Change	Change In Interval Structure
Add(c,d)	None. d becomes part of interval z. d had no successors and only one predecessor (c).
Add(d,e)	None.
Add(d,f)	None.
Add(d,b)	b becomes the head of a new interval (x).
Add(d,c)	c becomes head of a new interval (w). Another new interval (v) is formed with head f caused by the fact that node b (a predecessor of f) is no longer in the same interval as node f.
Delete(c,e)	None.

General Analysis

The following maintenance actions are required leave edge (m,n) is added to G_i. Let j = int(m), k = int(n).

Case 1: j = k = NULL (i.e., no interval structure above nodes n and m).

Before insertion, the graph G_i was not further reducible and no interval structure existed above the level at which nodes m and n reside. The addition of edge (m,n) may allow further reduction; such a reduction should be attempted (This is done by invoking REDUCE, see appendix II) .

Case 2: j ≠ k.

Case 2.1: n is the head of interval k.

No action is required.

Case 2.2: n is not the head of interval k.

Interval k will be split into at least two intervals; otherwise I1 is violated. No intervals other than k are affected.

Case 3: j = k ≠ NULL.

Case 3.1: n is the head of interval k.
No action is required.

Case 3.2: n is not the head of interval k.

Interval k may or may not be split (see addition of edges (d,e) and (d,b) in Example 4.1). An analysis must be performed to determine if a split occurs. □

The analysis mentioned in Case 3.2 above is left unspecified because several different analyses might be applied. The analysis method used by the algorithms in this paper is to perform PARTITION on the nodes of

interval j considered as a subgraph of G_i. This method is used because the mechanism is already available and requires no auxiliary data structures.

One might suspect that some linear ordering of the nodes in the interval might contain enough information to determine if such a split occurs in Case 3.2, assuming that the addition of edge (m,n) will:

- split the interval if n precedes m in the linear ordering, and
- cause no split if m precedes n in the linear ordering. However, by counter example, such a linear ordering can be shown not to exist. Even knowing the ancestral relationship between the two nodes is not sufficient.

Other data structures, such as depth first spanning tree [HEC73], may be useful. However, this introduces a new data structure and algorithms for maintaining it.

Deleting Edges From G_1

Deleting an edge (m,n) from G_i may cause one or more intervals to combine. The deletion of an edge will never cause an interval to split.

Example 4.2:

Assume we wish to delete node d from G0(B) to obtain G0(A). The following table presents specific actions that occur as each edge is inserted or deleted one at a time (with edges manipulated in the order in which they are presented). Again, only modifications to G_1 are presented.

Edge Change	Change In Interval Structure
Add(c,e)	None.
Delete(d,b)	Node b now has only one predecessor so intervals x and y combine into one.
Delete(d,c)	Node c now has only one predecessor so all of interval w combines with the predecessor interval (newly combined x and y from the previous deletion). The three edges (b,f) (e,f) and (d,f) now eminate from the same interval subsuming the interval of which f used to be the head (v).
Delete(d,f)	None.
Delete(d,e)	None.
Delete(c,d)	None.

□

When deleting an edge, the resulting configuration cannot violate I1 or I3. Thus, besides the possibility that deleting an edge may produce a graph which is not a flow graph, only I2 and maximality need to be considered in determining how intervals combine. The discussion below assumes that the editing functions which use these maintenance algorithms do not allow a non-flow graph to be produced. For a discussion of modifications to apply when this assumption does not hold, see Appendix III.

I2

Assume that edge (m,n) is to be deleted from G_i; let j = int(m), k = int(n). If $j \neq k$, then I2 cannot be violated for either interval (j or k).

Thus, assume j = k. If n is the head of j, I2 must still be satisfied after deletion. Thus, assume n is not the head of j. I1 implies that every path from e (the entry node) to n includes h ($\neq$ n), the head of j. If I2 is not satisfied upon deletion of (m,n) (i.e., there is no path from h to n), then there is no path from e to n. In other words, the directed graph is not a flow graph. Assuming only editing functions which preserve the flow graph property are permitted, the above situation cannot occur; thus, I2 cannot be violated.

MAXIMALITY

When applying PARTITION, a node (except the entry node) becomes an interval head because it has a predecessor in more than one interval. In each interval (except the entry interval) of the final partition, one of the following two conditions on the interval head must be true:

A: The interval head has no predecessors inside itself and has predecessors in two or more other intervals, or

B: the interval has one or more predecessors inside itself and predecessors in one or more other intervals.

Thus, if deleting an edge causes A and B to be false for some interval, i, then the old interval head can no longer head a maximal interval; interval i combines with its single interval predecessor.

General Analysis

The following maintenance actions are required when an edge (m,n) is deleted from G_i. Let j = int(m), k = int(n).

Case 1: n is the head of interval k.

If n has no predecessors inside k and predecessors eminating from only one other interval, then interval k should be combined with its single interval predecessor. If such a combination occurs, then the newly formed interval and all its successors must be checked for the same condition. Since edge (m,n) enters the head of k, I2 remains satisfied.

Case 2: n is not head of interval k (in this case we must have j = k). No action is required.

5. ALGORITHMS

The variable "level", which corresponds to the level of derived graph being processed, appears several places in the algorithms; it is assumed to be implicitly known at its points of use. This could be accomplished by explicitly passing "level" through the parameter list (this was not done here because it is not necessary for understanding the algorithms and adds unneeded complexity) or by holding the information in the nodes of the data structure.

For examples of how these algorithms are used to modify a glow graph, see Figure 4.1 (A and B).

INSERT_NODE is invoked to insert a node n with list of predecessors P and list of successors S. Each edge is inserted independently of all others by invoking ADD_EDGE. The condition "j = null at S1 (Figure 5.2) corresponds to there being no interval structure above node n. The addition of edge (n,s) may allow further reduction to take place. ADD_EDGE inserts the new edge into the structure of the appropriate G_{level} and recursively modifies the affected region of $G_{level+1}$.

The condition "k = null" at S2 corresponds to the fact that node s is being newly inserted into the flow graph and is not currently contained in any interval. The statement "int(s)← j" has the side affect of appending node s to the end of the list of nodes in interval j. This preserves properties P1 and P2 of interval order.

Bookkeeping problems could arise if the old interval structure was deleted before the new structure was inserted. However, since ADD_EDGE applies these operations in the opposite order, the edges of the old interval structure supply the "glue" required to maintain the flow graph structure as the new interval structure is inserted one component at a time.

DELETE_NODE is invoked to delete all edges associated with node n. Each edge is deleted independently of all others by invoking DELETE_EDGE. The condition "k = null" at S3 (Figure 5.4) has the same meaning as that at S1.

As an edge is deleted, two separate intervals may combine into one; such a combination may cause more intervals to combine. COMBINE is invoked to perform the required "check" and combine operations. The members of the set CHECK are intervals which are candidates for being combined with their predecessors. At S4 (Figure 5.5), a new interval is formed combining two previously existing intervals; the lists of members of the component intervals are concatenated in such a way that properties p1 and p2 of interval order are preserved.

6. DISCUSSION

The order of complexity for these maintenance algorithms is at least as great as that of the basic formation algorithm since PARTITION is used as an algorithm component and a change in G_i can cause a completely new

G_{i+1} structure. However, in practice, a change in G_i will normally cause only a segment of G_{i+1} to change. These maintenance algorithms restrict their attention to the smallest segment of the interval structure affected by a modification of the underlying flow graph. It is this property of localizing the effects of a modification that makes this technique more efficient than starting anew to form the interval structure each time a modification is performed.

Although these maintenance algorithms could be used to build the interval structure as the nodes of the underlying flow graph are entered, this is not suggested. This is because of the overhead involved in splitting intervals multiple times as new edges are inserted. Instead, it is suggested that the original flow graph be entirely collected and its initial interval structure determined (by invoking PARTITION). These maintenance algorithms are best used in applications where a small number of modifications are made to an already existing flow graph and interval structure.

The methods used by the maintenance algorithms cause the following properties to hold:

- The flow graph property of all graphs is preserved assuming the editing functions themselves do not destroy the flow graph property. This is accomplished by adding and deleting edges in the "correct" order within the algorithms.
- "Interval thrashing" is prevented in the sense that a given interval will not be split and then be recombined (or vice versa) during the processing of a given node. However, it is possible for, say, a sequence of edge insertions to successively split the same interval several times.
- The two properties, p1 and p2, of interval order are preserved.

There are two undesirable aspects of the maintenance algorithms. The first is that recursion is used to simplify control structure and propagate

information to higher levels of derived graphs; this produces undesirable overhead. The second is that since edge insertions and deletions are performed one at a time, intervals may successively split or combine during the processing of a given node. Thus, interval maintenance is not performed in an optimal manner.

A reasonable approach to the original problem which eliminates these undesirable properties is the following:

> At each graph G_i, collect information on <u>all</u> insertions and deletions. Once all modifications have been completed on G_i, propagate the newly determined interval information to G_{i+1} where it is collected before being propagated to the next higher level.

However, such an approach also has undesirable properties. First, some mechanism must be used to collect the modifications made to G_i. This complicates the algorithms and either introduces new data structures or complicates the existing ones. Second, the maintenance algorithms become much more complicated using this approach. One of the reasons why the maintenance algorithms presented in this paper are reasonable simply is that just before performing any modification to G_i, the interval structure above G_i is the correct previously updated interval structure. In the approach described above, once more than one modification has been collected at G_i, none of the old interval structure information can be assumed applicable.

Once these maintenance algorithms have been implemented, the driver routines for editing functions are very easy to generate. An example of the code necessary to add and delete a node (i.e., a statement of the source program) is shown in Figure 4.1. The driver routines for moving and modifying statements are equally simple.

7. SUMMARY

Algorithms have been presented which maintain the interval structure associated with an underlying flow graph as the flow graph is dynamically modified. The algorithms localize the effect of the modification to the smallest segment of the flow graph and interval structure possible. They also maintain the properties of interval order used by many interval analyses. They do not require that the underlying flow graph be reducible and never produce a non-flow graph as an intermediate structure.

APPENDIX I

Algorithm Language

This appendix describes the language used to present the interval formation and maintenance algorithms. The notation and special built-in functions are described in some detail because they are not found in general purpose languages.

Statements are separated by semicolons. Scalar variables, keywords (underlined), and built-in functions are represented by strings of lower case letters. Algorithm names and data structures are represented by strings of upper case letters.

Delimiters

• /* • */	comment
• (•)	delimit parameter list; change order of operation
• [•]	perform operation in an unspecified manner
• { • }	delimit elements of a set

Functions and Operators

- interval_node(R)

  Arguments: R - list of nodes

  Value: a node, i

  Semantics: node i is allocated and list R is associated with it. The "int" function of every r ϵ R is set to i

- delete_list(R,S)

  Arguments: R - list of nodes

  S - set of nodes

  Value: list of nodes

  Semantics: returns a copy of R with all occurances of every s ϵ S deleted

- R1 || R2

  Arguments: R1, R2 - list of nodes

  Value: list of nodes

  Semantics: concatenate R2 onto the end of R1

- r <= R

  Arguments: R - list of nodes

  Semantics: assign to r the first element of R (do not delete the element from the list)

- C ⌣ D -- C D

  Arguments: C, D - set of nodes

  Value: set of nodes

  Semantics: standard union and intersection

- C-D

  Arguments: C, D - either sets or lists of nodes

  Value: a set or list of nodes.

  Semantics: for sets - standard meaning

  for lists - delete from a copy of C the first occurance of each d ϵ D, if it exists in C

- <c>

  Arguments: c - a node

  Value: list of nodes

  Semantics: consider c to be a one element list

- |C|

  Arguments: C - set, list or graph of nodes

  Value: an integer

  Semantics: the number of elements in C

- /R/

  Arguments: R - list or graph of nodes

  Value: collection of nodes

  Semantics: consider the elements of R as a collection (i.e., "strip" the structure from the elements)

- head(i)

  Arguments: i - an interval node

  Value: a node

  Semantics: return the head of interval i

Psuedo-functions

- succ(n)

  Arguments: n - a node

  Value: list of nodes

  Semantics: Expression: returns the successors of n

  Object: modify edges such that assigned information can be retrieved (this, of course, changes the value of the "pred" function also)

- pred(n)

  Arguments: n - a node

  Value: list of nodes

  Semantics: similar to succ(n)

- list(i)

  Arguments: i - an interval node

  Value: list of nodes

Semantics: Expression: returns the list of nodes contained in interval i

Object: modify the list of nodes associated with interval i

- int(n)

Arguments: n - a node

Value: interval node

Semantics: Expression: returns the interval node in which n is contained: null if not in an interval

Object: set node n to be contained in the interval; also append n to the end of the interval list

The psuedo-functions defined above (when used as functions) may also take arguments which are sets or lists of node returning the expected results, i.e.,

int(A) = (∪ int(a) ∀ a ∈ A) if A is a set, and

(|| int(a) ∀ a ∈ A) if A is a list.

Predicates

- header(n)

Arguments: n - a node

Value: Boolean

Semantics: TRUE if n is the head of an interval;

FALSE otherwise

APPENDIX II

Interval Formation Algorithms

This appendix presents a set of 3 algorithms used to produce the sequence of derived graphs used in interval analysis. The algorithms are modularized

in this way so that they can be used as components of the maintenance algorithms.

Given a graph, G, and "candidate" set of nodes, N, MAX_INTERVAL (Figure II.1) forms a maximal interval (nodes returned in LIST) with head h. M is used to maintain the set of current "candidate" nodes as nodes are extracted and placed into the interval.

PARTITION (Figure II.2) is used to partition a subgraph, S, of G into disjoint intervals. The entry node of S is e; the set of intervals produced by the partitioning is returned in list R.

REDUCE (Figure II.3) is used to produce the sequence of derived graphs generated by the underlying flow graph, G. Note that REDUCE assumes a data structure capable of storing the newly determined derived graphs and their entry nodes and that "some" of these derived graphs have already been placed in the data structure. The entire sequence of derived graphs is generated by invoking REDUCE (0) (where $G = G_0$ has already been placed in the data structure).

APPENDIX III

Modified Approach When Non-flow Graphs Are Permitted

This appendix presents a modified approach to be applied when the editing functions that use these maintenance algorithms allow non-flow graphs to be produced. The reason this subject is relegated to this appendix instead of being addressed in the body of the paper is that the modifications to be presented are best implemented at a very "low level"; i.e., within the underlying routines that actually perform the edge insertions and deletions. If they are implemented at such a level, the maintenance algorithms presented in the body of the paper require only small modification. The remainder of the discussion assumes implementation at this low level. The three specific questions

that must be addressed are:

- how to determine when a non-flow graph has been produced,
- what part of the old flow graph should be "extracted" to produce the new flow graph, and
- how do these extracted portions recombine with the flow graph when the appropriate edges are added.

Since the derived graph of the a flow graph is also a flow graph, non-flow graph consideration need only be applied to $G = G_0$. (In other words, by extracting a portion of G to maintain its flow graph property, the higher levels of derived graphs will naturally inherit their flow graph property.) The extracted nodes will be placed in a psuedo-interval, *, which utilizes the data structures associated with intervals but does not possess the logical structure of intervals as presented in Section 2. Because of these properties the underlying routines must be able to determine which graph they are manipulating and must be able to identify the "distinguished" interval, *. (For instance, the manipulation of edges completely contained within * should not cause any interval re-structuring.) The psuedo-interval, *, will contain all nodes which are not reachable from e, the entry node.

Note that such a framework allows the edges which eminate from nodes of * to affect the interval structure of the remaining nodes of G. Of course, no node in * will have an edge with a source node outside of *.

Deleting an Edge

Extraction of nodes from the directed graph will only occur upon deletion of an edge. Such an extraction of nodes should occur whenever a

subgraph of G is produced which cannot be reached from the entry node, e. This occurs upon the deletion of edge (n,s) if either:

1) s is the header of its interval and has no predecessors outside its interval (or interval *), or

2) s has no predecessors at all

In case 1, the entire interval should be extracted and placed in *. In case 2, that region of the interval to be extracted must be determined; this can be done by executing MAX_INTERVAL with s as the head of the interval to be formed. In either case, the extraction of nodes may produce some regions of the graph that must be extracted. The inclusion of the code in Figure III.1 within the underlying edge deletion routine just after the edge (n,s) has been deleted produces the desired result.

Adding an Edge

The addition of an edge, (n,s), where int(s) = *, will, of course, introduce nodes from * back into G. As such an edge is added, that region of * reachable from e (the entry node) must be determined. This can be done by executing PARTITION with s assumed to be the entry node of * considered as a subgraph of G. As the newly formed intervals are inserted, they may recombine with already existing intervals. The inclusion of the code in Figure III.2 within the underlying edge insertion routine just after edge (n,s) has been added produces the desired results.

```
MAX_INTERVAL(h,N,G;LIST,M)
   LIST ← <h>
   M ← N;
S5:
   do while ([there exists m ∈ M ∩ succ(LIST)
        and {/pred(m)/} ⊂ {/LIST/}])
      LIST ← LIST || <m>;
      M ← M - {m};
   od;
   return;
END MAX_INTERVAL
```

Figure II.1
SPECIFICATION OF MAX_INTERVAL

```
PARTITION(e,S,G;R)
   H ← {e};
   N ← S - H;
   R ← null;
   do while (|H| ≠ 0)
      [pick an arbitrary h ∈ H];
      H ← H - {h};
      MAX_INTERVAL(h,N,G;LIST,M);
      R ← R || interval_node(LIST);
      H ← H ∪ ({/succ(LIST)/} ∩ M);
      N ← M - {/succ(LIST)/};
   od;
   return;
END PARTITION
```

Figure II.2
SPECIFICATION OF PARTITION

```
REDUCE(lev)
   do
      PARTITION(e_lev,{/G_lev/}, G_lev; R);
      for every r ∈ R
         Q ← list(r);
         S ← delete_list(succ(Q), {/Q/});
         succ(r) ← int(S);
      rof;
      n <= R;
      G ← (R, [edges as added],n);
      [e_lev+1 ← n; G_lev+1 ← G;]
      while ([G ≠ G_lev])
      REDUCE(lev+1);
   od;
   return;
END REDUCE
```

Figure II.3
SPECIFICATION OF REDUCE

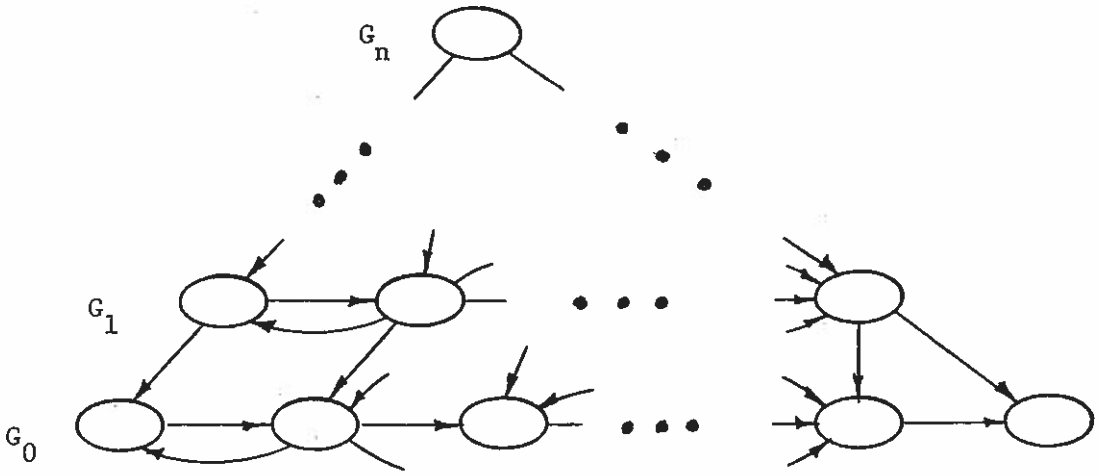
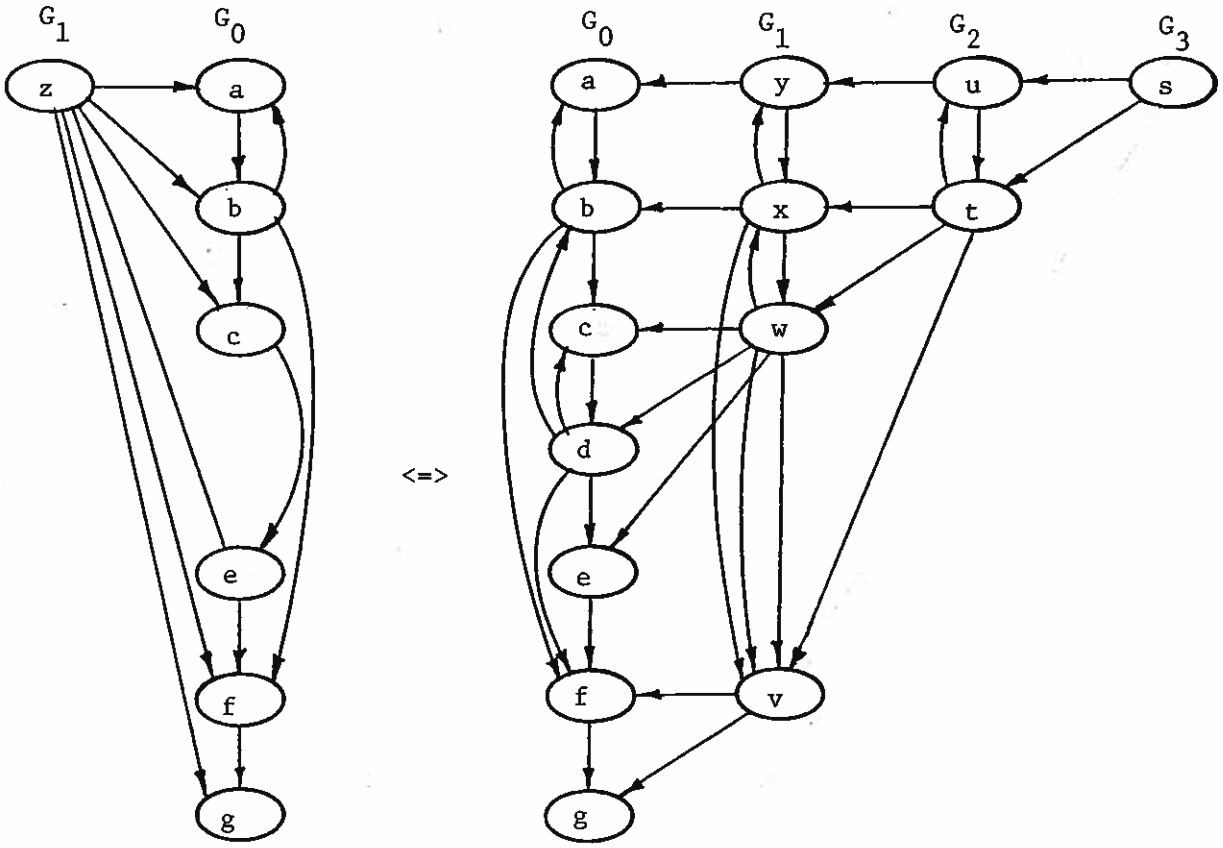


Figure 3.1
EXAMPLE DERIVED GRAPH DATA STRUCTURE



INSERT_NODE(d,<c>,<e,f,e,c>)
DELETE_EDGE(c,e)

(A)

ADD_EDGE(c,e)
DELETE_NODE(d)

(B)

(for a description of INSERT_NODE, DELETE_EDGE, ADD_EDGE and DELETE_NODE, see Section 5)

Figure 4.1
INTERVAL STRUCTURE CHANGES DUE TO
INSERTIONS/DELETIONS

```
S ← {s};
do while (S ≠ ∅)
   [pick s ∈ S];
   S ← S - {s};
   k ← int(s);
   Q ← list(k);
   if header(s) then
      if |pred(s) - (list(*) || Q) | = 0 then
         S ← S ∪ ({/succ(Q)/} - {/Q/});
         list(*) ← list(*) || Q;
         DELETE_NODE(k);
      fi;
   else
      if |pred(s)| = 0 then
         MAX_INTERVAL(s, {/Q/}, G;LIST,M);
         list(k) ← Q - LIST;
         S ← S ∪ ({/succ(LIST)/} - {/LIST/});
         list(*) ← list(*) || LIST;
      fi;
   fi;
od;
```

Figure III.1

```
if int(s) = * then
   PARTITION(s, {/list(*)/}, G;R);
   for every r ∈ R
      Q ← list(r);
      P ← delete_list(pred(head(r)), {/Q/});
      S ← delete_list(succ(Q), {/Q/});
      INSERT_NODE(r, int(P), int(S));
   rof;
   r <= R;
   COMBINE({r});
fi;
```

Figure III.2

```
INSERT_NODE (n,P,S)
   if P = null then /* new graph entry node */
      [make n new entry node for corresponding derived graph];
      if S = null then /* whole graph is just n */
         [delete all higher level derived graphs ];
         return;
      fi;
   /* add predecessors edges */
   else for every p ∈ P
            ADD_EDGE(p,n);
        rof;
   fi;
   /* add successor edges */
  for every s ∈ S
     ADD_EDGE(n,s)
  for;
  return;
END INSERT_NOTE
```

Figure 5.1
SPECIFICATION OF INSERT_NODE

```
ADD_EDGE(n,s)
   j ← int(n); k ← int(s);
   succ(n) ← succ(n) || <s> ;
S1:    /* if further reduction possible, try to reduce */
   if j = null then REDUCE(level); return; fi;
S2:    /* s is a newly inserted node; put it into interval */
   if k = null then int(s) ← j; return; fi;
   if ¬header(s) then
   /* get a new partition of int k */
   PARTITION (head (k), {/list(k)/}, G_level ; R);
   /* insert new structure */
   for every r ∈ R
      Q ← list(r);
      P ← delete_list(pred(head(r)), {/Q/});
      S ← delete_list(succ(Q), {/Q/});
      INSERT_NODE(r,int(P), int(S));
   rof;
   /* delete old structure */
   DELETE_NODE(j);
   fi;
   return;
END ADD_EDGE
```

Figure 5.2
SPECIFICATION OF ADD_EDGE

```
DELETE_NODE(n)
    /* delete successor edges */
    for every s ∈ succ(n)
       DELETE_EDGE(n,s);
    rof;
    /* delete predecessor edges */
    for every p ∈ pred(n)
       DELETE_EDGE(p,n);
    rof;
    return;
END DELETE_NOTE
```

Figure 5.3
SPECIFICATION OF DELETE_NODE

```
DELETE_EDGE(n,s)
   succ(n) ← succ(n) - <s> ;
   k ← int(s);
S3:
   if k = null then REDUCE(level);
   else COMBINE({k});
   fi;
   return;
END DELETE_EDGE
```

Figure 5.4
SPECIFICATION OF DELETE_EDGE

```
COMBINE(CHECK)
   do while ([there exists m ∈ CHECK such that
      |{/pred(m)/}| = 1
      and {/pred(list(m))/} ∩ {/list(m)/} = ∅])
      p ← /pred(m)/; /* know there can only be one predecessor */
      /*  find combined successors */
      S ← delete_list(succ(p) || succ(m), {m,p});
      /* form new combined interval */
S4:   t ←  interval_node(list(p) || list(m));
      /* insert new structure */
      INSERT_NODE(t,pred(p), S);
      CHECK ← (CHECK - {m,p}) ∪ {t} ∪ {/S/};
      /* delete old structure */
      DELETE_NODE(m);
      DELETE_NODE(p);
   od;
   return;
END COMBINE
```

Figure 5.5
SPECIFICATION OF COMBINE

REFERENCES

[ALL70] Allen, F.E. Control Flow Analysis. SIGPLAN Notices 5, 7 (Jul. 1970) 1-19.

[ALL76] Allen, F.E. and Cocke, J. A Program Data Flow Analysis Procedure. CACM 19, 3 (Mar. 1976) 137-147.

[COC70] Cocke, J. Global Common Subexpression Eliminating. SIGPLAN Notices 5, 7 (Jul. 1970) 20-24.

[DAN75] Danielson, Ronald L. PATTIE: An Automated Tutor for Top-down Programming. Ph.D. Thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-695 (Jan. 1975).

[DAV75] Davis, A. An Interactive Analysis System for Execution-time Errors. Ph.D. Thesis, Department of Computer Science, University of Illinois, Report UIUCDCS-R-75-695 (Jan. 1975).

[GIL76] Gillett, Will D. An Interactive Program Advising System. Proceedings of the ACM Conference on Computer Science and Education (Feb. 1976) 335-341.

[HEC72] Hecht, Mathew S. and Ullman, Jeffrey D. Flow Graph Reducibility. SIAM Journal of Computing 1, 2 (Jun. 1972) 188-202.

[HEC73] Hecht, Mathew S. and Ullman, Jeffrey D. Analysis of a Simple Algorithm for Global Data Flow Problems. SIGAC-SIGPLAN (Oct. 1973) 207-217.

[HEC74] Hecht, Mathew S. and Ullman, Jeffrey D. Characterization of Reducible Flow Graphs. JACM 21, 2 (Jul. 1974) 367-375.

[TIN75] Tindall, Michael H. An Interactive Compile-time Diagnostic System. Ph.D. Thesis, Department of Computer Science, University of Illinois, Report UIUDCDS-R-75-748 (Oct. 1975).

[WIL73] Wilcox, T.R. The Interactive Compiler as a Consultant in the Computer Aided Instruction of Programming. Proceedings of the Seventh Princeton Conference in Information Sciences and Systems (Mar. 1973).