Graph Services for Program Understanding Tools

Patrick Brown
David W. Stafford

IBM Sterling Forest
PO Box 700
Suffern, NY 10901

Abstract

Many types of problems are more easily understood when represented with graphs. This paper presents an organization of graph theoretical functions for use by an application program. The discussion covers a unified set of generalized routines and data structures for graph reductions, path analysis, and data flow problems. Some suggestions for user interfaces are provided. Examples of problems in which these functions might be used are program understanding, software testing, software design, scheduling, and network management.

Background Material

All of these functions assume an arbitrary directed graph with a root and sink (exit) vertex. In practice, many control flowgraphs will have multiple entry points and multiple exits. Therefore, an imaginary root and sink \((r,s)\) may be created for the purpose of analysis. The imaginary root and sink are not usually displayed to the user.

Given a directed graph \(G(V,E,r)\):

- \(V\) is the set of vertices in \(G\)
- \(E\) is the set of edges \((u,v)\) in \(G\). \((u,v)\) represents the edge between vertex \(u\) and vertex \(v\).
- \(r\) is the root or source vertex of graph \(G\).

The term "flowgraph" covers both control flow and call graphs. A flowgraph has a source or root vertex \(r\), which has no predecessors, and one or more exit or sink vertices, each of which has no successors.

To build a control flowgraph of a program, we partition its statements into blocks, sequences of consecutive statements having a single entry and a single exit at the end. Each basic block is represented by a vertex in the graph. Each edge represents a possible transfer of control between basic blocks.

There is an edge \((u,v)\) in the graph if, during execution, control can transfer from basic block \(U\) to basic block \(V\). If \((u,v)\) is an edge then we call \(v\) an immediate successor of \(U\), and \(U\) an immediate predecessor of \(v\). Although each basic block has only one entry point, it can have more than one immediate predecessor.

Each edge \(e\) has a head, \(h(e)\), and a tail, \(t(e)\). We say edge \(e\) leads from \(h(e)\) to \(t(e)\).

A path sequence for a directed graph is a sequence \((P_1,v_1,w_1), (P_2,v_2,w_2), \ldots, (P_k,v_k,w_k)\), containing vertices \(v_1, v_2, \ldots, v_k\) and avoiding all other edges and vertices. The length of a path, \(P(x_1, x_2, \ldots, x_k)\), containing vertices \(x_1, x_2, \ldots, x_k\) is \(k - 1\).

A simple path \(P\) is a path containing no vertex twice.

There is an empty path of no edges from any vertex to itself. A cycle is a non-empty path from a vertex to itself.

If there is a path from a vertex \(u\) to a vertex \(v\), then vertex \(v\) is reachable from \(u\).

A path sequence for a directed graph is a sequence \((P_1,v_1,w_1), (P_2,v_2,w_2), \ldots, (P_k,v_k,w_k)\). For any non-empty path, \(P\), in \(G\), there is unique partition of \(P\) into non-empty paths \(P = P_1,P_2,P_3,\ldots,P_k\).
A reducible flowgraph, G = (V, E, r), is a flowgraph that can be reduced to a single vertex r and no edges. Intuitively, a flowgraph is reducible if every cycle has a single entry from the start vertex.

The following functions are described in subsequent sections:

Given a directed graph G(V, E, r):

- Generate a spanning tree for G.
- Develop forward and backward dominator trees for G.
- Identify the flow dependencies (upward and downward) in graph.
- Identify self-contained sub-programs (proper subgraphs).
- Reduce a graph to a specified target size.
- For a reduced graph, expand selected nodes to present abstracted detail to the user.
- Identify all paths between any two nodes.
- For a given path:
  - Identify all data required to execute the path.
  - Identify all data updated in the path.
  - Determine values of data updated in a path.

### Graph Services

#### Spanning tree

Generating a spanning tree is the first step in many graph functions. A spanning tree is generated by using a depth-first search to visit all vertices. The vertices will be numbered from 1 to n as they are reached during the search. Variables used in other processes will be initialized in the spanning tree algorithm.

### Data Structures

The following data sets will be used:

**Input**

- **suc(v)**: The set of vertices w, such that (v, w) is an edge of the graph.

**Computed**

- **parent(w)**: The vertex which is the parent of vertex w in the spanning tree.
- **pred(w)**: The set of vertices v, such that (v, w) is an edge of the graph.
- **semi(w)**: A number defined as follows:
  - Before vertex w is numbered, semi(v) = 0.
  - After w is numbered but before its semidominator is computed, semi(w) is the number of w.
  - After the semidominator of w is computed, semi(w) is the number of the semidominator of w.

**vertex(r)**: The vertex whose number is r.

**bucket(w)**: A set of vertices whose semidominator is w.

### Algorithm

The algorithm maintains a data structure which represents a forest of nodes contained in the depth-first spanning tree (DFST). The forest consists of vertex set V, and edge set {(parent(w), w) | vertex w has been placed in the spanning tree}.

For each vertex (v) in the graph, starting with the root node (r):

1. Vertex(i) = v, i = i + 1.
2. For each node (w), where w is a successor of v.
   - Set parent(w) = v
   - Repeat the step for node(w)

#### Forward dominator tree

A dominator tree identifies important structural information in a graph. Forward dominators are vertices which "control" or "give control" to other vertices in the forward flow of the graph. A dominator tree defines the single entry regions for a graph.

If v and w are two distinct nodes in a flowgraph G, then v dominates w, if every path in G from the initial node to w contains v. Vertex v is the immediate dominator of w, denoted v = idom(w), if v dominates w and every other dominator of w dominates v.

We wish to construct the dominator tree of an arbitrary flowgraph G. If G represents the flow of control of a computer program, then the dominator tree provides information about control flow dependencies.

### Data Structures

The following data sets will be used:

**Input**

- **suc(v)**: The set of vertices w, such that (v, w) is an edge of the graph.

**Computed**

- **parent(w)**: The vertex which is the parent of vertex w in the spanning tree.
- **pred(w)**: The set of vertices v, such that (v, w) is an edge of the graph.
- **semi(w)**: A number defined as follows:
  - Before vertex w is numbered, semi(v) = 0.
  - After w is numbered but before its semidominator is computed, semi(w) is the number of w.
  - After the semidominator of w is computed, semi(w) is the number of the semidominator of w.
vertex(v)
The vertex whose number is i.

bucket (w)
A set of vertices whose semidominator is w.

dom(w)
A vertex defined as follows:
1. After step 3, if the semidominator of w is its immediate dominator, then dom(w) is the immediate dominator of w. Otherwise dom(w) is a vertex v whose number is smaller than w and whose immediate dominator is also w’s immediate dominator.
2. After step 4, dom(w) is the immediate dominator of w.

Ancestor(v)
An array of predecessors. Initially ancestor(v) = 0. In general ancestor(v) = 0 only if v is a tree root in the forest; otherwise ancestor(v) is an ancestor in the forest.

Label(v)
An array such that Label(v) = v for each vertex v.

Algorithm
The algorithm consists of the following four steps:
1. Carry out a depth-first search of the problem graph. Number the vertices from 1 to n as they are encountered, top to bottom, left to right. Initialize variables used in steps 2-4.
2. Compute the semidominators of all vertices.
3. Implicitly define the immediate dominator of each vertex.
4. Explicitly define the immediate dominator of each vertex, carrying out the computation, vertex by vertex, in increasing order number.

Step 1 uses the recursive procedure DFST, defined above.
Steps 2 and 3 are carried out simultaneously, processing the vertices w ≠ r (not the root) in decreasing order by number. A data set is maintained which represents a forest contained in the depth-first spanning tree. The forest consists of a set of vertices V and a set of edges E = { parent(w),w } | vertex w has been placed in the spanning tree).

Two procedures are used; one to construct the forest and another to extract data from the forest.

Procedure
LINK(v,w) Add edge (v,w) to the forest.
EVAL(v) If v is the root of a tree in the forest, return v. Otherwise, let r be the root of the tree in the forest which contains v. Return any vertex u ≠ r of minimum semi(u) on the path r–v.

The algorithm assigns semi(w) = min{ semi(EVAL(v)) | (v,w) are in edge set E}. After this assignment, semi(w) is the number of the semidominator of w.

After computing semi(w), the algorithm adds w to bucket(vertex(semi(w))) and adds a new edge to the forest using LINK(parent(w),w). This completes step 2 for w. The algorithm then empties bucket(parent(w)), carrying out step 3 for each vertex in the bucket.

Step 4 examines the vertices in increasing order by number, filling in immediate dominators not explicitly computed in step 3.

Two arrays are used to represent the forest built by the LINK instructions:

Array
Ancestor Initially ancestor(v) = 0. In general ancestor(v) = 0 only if v is a tree root in the forest; otherwise ancestor(v) is an ancestor in the forest.

Label(v) Label(v) = v for each vertex v.

To carry out LINK(v,w), the algorithm assigns ancestor(w) = v.

To carry out EVAL(v), the algorithm follows ancestor pointers to determine the path from v to r. If v = r, then v is returned. Otherwise, the algorithm performs a path compression by assigning ancestor(vi) for i from 2 to k, simultaneously updating labels as follows:
If semi(label(vi-1)) < semi(label(vi))
then
label(vi) = label(vi-1);
The label(v) is returned.

Backward dominator tree
The backward dominator tree identifies those nodes which ‘receive’ control in the graph. The backward dominator tree is built by reversing the direction of edges in the original control flowgraph and applying the algorithm.

Proper subgraphs
Proper sub-programs are those which have one entry point and one exit point. These sub-programs have several interesting characteristics. They are good candidates to be made into sub-routines because they have a single entry and exit point. When examined with data flow information, they also provide information about the degree of data encapsulation present in the program.

A proper sub-program can be identified in a graph as a collection of paths which are forward dominated by one node and backward dominated by another node. That is, a proper sub-program is a set of paths such that all
paths starting from a given node also pass through a second node and there are no branches into or out of the set of paths.

For proper subgraph, the set $\text{dom}(u)$, the nodes dominated by node $u$, and the set $\text{dom}(vr)$, the set of nodes dominated by node $v$ in the reverse graph, will contain the same nodes.

End User Interface

One use for this function is in helping programmers break a program into smaller programs. For this purpose, it will be useful to highlight the proper subgraphs contained in the original graph. This will identify the possible sub-routines to the user.

Another use is to examine a program for the presence or absence of data encapsulation. Data encapsulation occurs when the only usage of a data element occurs within a proper sub-program. This means that the only usage of the data is within a single entry, single exit routine. In this example, both data usage and control structure are highlighted. Let us assume that the proper subgraphs are already highlighted in the original graph.

We determine whether or not the data elements used in any proper-subgraph are used in any other nodes outside of the proper subgraph. Those data elements which are used only within a single subgraph may be said to be encapsulated.

The user interface might work as follows:

1. Subgraphs which encapsulate one or more data elements will be highlighted in reverse video. By selecting the highlighted subgraphs with the mouse, the user can see a list of the encapsulated data.
2. Users may bring up list of data elements used, either those used in a particular sub-program or those used across the entire program. They may assign colors to individual data elements or clusters of data elements. Each node which references or sets one of the data elements will be shaded with the color associated with the data element. The result will allow the user to visually determine where encapsulation occurs and which nodes cause violations.

Graph reduction

Diagrams representing logic flow and calling relationships are often quite large. The size and complexity of the diagrams pose problems for the tool developer who is trying to produce the diagram, as well as the end user who tries to use it.

Control flow and calling structure diagrams differ in structural characteristics as well as intended use. Different kinds of abstraction and layout schemes can be used to assist the user perceive the underlying structure of the data being presented. An abstraction of control flow will mimic the top-down refinement of program logic. Abstraction of calling diagrams involves grouping of function, rather than analysis of instruction sequences.
A simple and useful approach is to assign a default abstraction and layout style for each kind of graph being produced. A more sophisticated approach allows a user to invoke different abstractions or layouts in order to examine the characteristics of the graph being viewed. A further enhancement might allow the user to specify the parameters which control the abstraction or layout.

Two abstractions are presented in the following sections. The first describes the reduction of a control flow graph. Most control flow graphs contain a large number single-entry regions corresponding to structured programming constructs. The algorithm uses these regions and a loop analysis routine to reduce structured constructs to a single node.

The second algorithm describes a clustering technique intended for use on calling structure graphs. Particularly in operating systems or applications with many utility/service modules, there is a high degree of interrelationships between many of the modules. The graphs are characterized by large numbers of nodes with a high in-degree and out-degree. Clustering organizes and simplifies the graph by collecting nodes into groups of nodes or concentrations of edges, which helps users perceive patterns of relationships.

Figure 1 on page iv shows an example of a graph abstraction.

Reducing control flow graphs

Two transformations and interval reductions will be used to reduce large graphs to a more comprehensible size.

The first transform deals with cycles. Let G be a flowgraph and let (x,y) be an edge of G. Transformation T1 is the removal of this edge. If T1 is applied to delete an edge e, then vertex v(e) = h(e) in the reduced graph represents the union of what h(e) and e represent.

Let y not be the root node and have a single predecessor x. Transformation T2 is the replacement of node x and y and edge (x,y) by a single node z. Predecessors of x become predecessors of z. Successors of y become successors of z. There is an edge (z,x) if and only if there was formerly an edge (y,x) or (x,x). Whenever T2 is applied, we say that x consumes y. If T2 is applied to compress vertex y into vertex x, then x in the reduced graph represents the union of what x, y, and all the deleted edges e with h(e)=x and (e)=y represent.

This algorithm uses single entry and strongly connected subgraphs (called intervals) to direct the reduction. The interval approach partitions the flowgraph into subgraphs, replacing each interval with a single node, and continuing to define such interval partitions until the graph cannot be further reduced. In an interval I, there is one vertex, the interval head h, such that all edges from outside the interval to nodes inside the interval flow through h. The collection of intervals for a program closely models the loop structure of the program control flowgraph.

Head(v) for v \neq r is the vertex into which v is eventually contracted. Cycle(v) for any vertex v is the set of edges in G which were deleted when applying T1. Non-cycle(v) for v \neq r is the set of edges deleted when applying T2 to delete v.

The interval analysis defines an order of nodes, reduction order, closely related to the depth-first spanning tree construction. By definition, the source node is the interval head node for the outermost interval, which may not be strongly connected.

Data Structures

Input

Depth-First Spanning Tree (DFST) A tree rooted at r, with vertices numbered in pre-order, such that (v,w) is an edge in the spanning tree.

Back-edge Targets The set of vertices w, such that (v,w) is a backward edge in the original graph. Edge (v,w) is a backward edge if and only if v=w or w dominates v.

Computed

Intervals I(h) The set of vertices v, such that (v,w) is an edge in a strongly connected subgraph of G. Vertex h is the head or root of the interval.

reachunder(x) An interval for which the head is the target of a back edge.

HIGHPT(y) For each vertex, HIGHPT(y) is the head node for all nodes y contained in a reachunder set. In reachunder set Ix, all nodes y will have HIGHPT(y) = x.

SNUMBER(y) The nodes are numbered by an ancestor first, right to left traversal of the DFST.

Cycle(v) The set of edges which were deleted when applying transform T1.

Non-Cycle(v) The set of edges which were deleted when applying transform T2.

Algorithm

1. We first form a depth-first spanning tree (DFST) on the flowgraph G, rooted at the source node of G, and number the nodes by preorder traversal.

2. We obtain the set of all back edges in G. The back edge targets become interval head nodes. Edge (x,y) is a back edge if and only if x=y or y dominates x.

3. For each back edge target, in reverse node order, we perform the following procedure:
a. Calculate the set $\text{reachunder} (x)$, where node $n$ is a member of the set if there is a simple path from $n$ to $x$ for which the final edge is a back edge. $\text{reachunder}(x)$ is the interval $I_x$.

b. All the nodes in $I_x$ are removed from $G$ and represented by a node $x$ in the newly derived flowgraph.

c. We set $\text{HIGHPT}(y) = x$ for all nodes $y$ in $\text{reachunder}(x)$ and continue to form $\text{reachunder}$ sets using the newly derived graph.

If the original graph is reducible, the graph transformation process results in a final graph consisting of one interval $I_{\text{source}}$. This final interval contains all nodes that are not within any strongly connected subgraph of the flowgraph, and some nodes representing intervals not nested within any other intervals.

4. Number all the nodes according to an ancestor-first, right-most first traversal of the DFST, calling this number $\text{SNUMBER}$.

5. We associate the tuple $(\text{HIGHPOINT}(y), \text{SNUMBER}(y)) = (y_1, y_2)$ with every node $y$. Sort the tuples so that $x$ precedes $y$ in reduction order if and only if $x_1 > y_1$ or $x_1 = y_1$ and $x_2 < y_2$. $\text{HIGHPOINT}$ values within an interval are its interval head node.

The selection on $\text{HIGHPOINT}$ ensures that inner, nested intervals are reduced before the outer, surrounding intervals. Nestings of intervals can be traced by following reverse sequences of $\text{HIGHPOINT}$ values corresponding to interval head nodes. The $\text{HIGHPOINT}$ function yields loop-nesting information for the program represented by the flowgraph, since the intervals correspond to loops.

Figure 2 shows some examples of reduction.

**Figure 2. Examples of interval reduction.**

**Example 1a** Shows a graph prior to reduction. Transformation $T_1$ on arc B-B results in the graph shown in 1b. Transformation $T_2$ is applied to node B in Example 1b to collapse node B into node A. Applying the transformation again reduces the graph to a single node, shown in 1d.

**Example 2** Shows two disconnected graphs. Each graph may be reduced into a single node resulting in two disconnected nodes, A and D.

**Example 3** Shows a graph containing a cycle, BCD. Nodes B and C are entry points into the cycle. Applying $T_2$ collapses node D into node C. However, no further transformations can be applied, and the result is the graph shown in 3b.

**Example 4** Cannot be reduced because node C cannot be collapsed into both A and B.
**End User Interface:** The desired effect is that of abstracting the control flow from the lowest levels of detail to increasingly simpler structures at higher levels. It is very confusing to see intervals reduced at several nesting levels. For example, if some reductions were to be done at nesting level 2 and other reductions on structures at nesting level 10, the user may be confused by the activity at several levels of detail.

The reduction starts by using interval reduction to remove the lowest level of nested intervals, then the next lowest level, and so on. This retains the significant structural properties of the graph by eliminating the small "leaves" of the graph first. The user perceives the graph to be simplified, from the bottom upward.

Our initial implementation gave users a slide bar, which was calibrated in percentages ranging from 0% to 100% (of nodes and arcs in the graph). The graph was reduced until the number of nodes and arcs remaining was equal to or less than the target number represented by the selected percentage. Figure 1 on page iv illustrates an example of this.

The concept of percentage is useful when thinking about the size of the graph. However, most programmers seem to be more comfortable with the idea of nesting level and this appears to be a more appropriate scale for reductions. We are investigating the approach of letting users pick a nesting level, rather than a percentage.

There is also a need to allow the user to selectively reduce nodes in a one-at-a-time fashion. Although the graph services will support this activity, there is not a user interface.

**Clustering for call graphs**

The goal of a clustering algorithm is to identify a set of nodes and/or edges that are more strongly connected with each other than with the rest of the graph.

Clustering is a form of abstraction that is especially useful for structure specification. It involves grouping entities that have some important common properties into a cluster, associating those properties with the cluster itself, and regarding the cluster as an atomic entity whenever possible. Interactions between members of clusters are specified in terms of the clusters themselves.

Packaging the clusters into subgraphs and separating the subgraphs geographically provides additional help in summarizing complex structures and understand modularity. A characteristic of clustered graphs is a reduction in the total number of edges and, often, a significant reduction in the number of edge crossings. Since the clustering is done prior to the layout process, many different types of clustering algorithms may be developed and used for specific purposes, without reducing the generality of the approach. Figure 3 on page viii shows an example of the grouping and simplification achieved by a clustering algorithm.

Clustering techniques have been described as foundations for transforming or respecifying existing code into new technologies such as identifying parallelism in existing programs, and objectifying or identifying software objects in existing programs. Both of these are of interest to many users of software reverse engineering or program understanding tools, and the potential to share technology among tools is attractive.
Figure 3. Simplification of a call graph with clusters

Data Structures

Input
Successor list For each node $x$, $\text{Succ}(x) = y$ is a list, such that $(x,y)$ is an edge in the graph.

Computed
Intersection An intersection consists of a set of source nodes (callers) and target (called) nodes. An intersection is formed by a pair of source nodes and the intersection of their successor nodes.

Concentration A concentration is a set of source and target nodes such that all the source nodes in the set are connected to all the target nodes in the set.

Output
Successor list Successor list updated with nodes representing edge concentrations. New nodes representing edge concentrations are added and successors of existing nodes are modified to reflect the insertion of the new nodes.

Algorithm: In the following description, $\text{SRC}(I)$ and $\text{TGT}(I)$ denote the source nodes and target nodes of intersection $I$, respectively. A minimum size, $M$, for the subgraphs may be specified. The size is initially set to $((\text{number of target nodes}) -1) \times (\text{number of source nodes})$. This ensures that concentrations having only a single target node will not be created.

1. Objective: Create a list of independent concentrations that are at least of size $M$.
   
   a. Set initial concentration list to empty list
   b. Sort successors of each node. This will make the calculation of intersections more efficient in a later step.
   c. For each pair of source nodes, calculate the intersection of target nodes and maintain a list of intersections. Sort the list of intersections into increasing order of target nodes.
   d. For each intersection $I$ in the intersection list

   Compare the intersection $I$ with each concentration $C$ in the concentration list.
   
   1) If size $I < M$ then
      Discard $I$
   2) If $\text{TGT}(I) = \text{TGT}(C)$ then
      Add $\text{SRC}(I)$ to the nodes in $\text{SRC}(C)$
   3) If $\text{TGT}(C)$ is contained in $\text{TGT}(I)$ then
      Split $C$ into $I$ and $C'$
      Add $I$ and $C'$ to the concentration list
   4) If $\text{TGT}(I)$ is contained in $\text{TGT}(C)$ then
      Split $I$ into $I'$ and $C$
      Add $I'$ and $C$ to the concentration list
   5) If $I$ has no intersection with any $C$ then

   Algorithm:

   1. Objective: Create a list of independent concentrations that are at least of size $M$.
   2. Sort successors of each node. This will make the calculation of intersections more efficient in a later step.
   3. For each pair of source nodes, calculate the intersection of target nodes and maintain a list of intersections. Sort the list of intersections into increasing order of target nodes.
   4. For each intersection $I$ in the intersection list

   Compare the intersection $I$ with each concentration $C$ in the concentration list.
   
   1) If size $I < M$ then
      Discard $I$
   2) If $\text{TGT}(I) = \text{TGT}(C)$ then
      Add $\text{SRC}(I)$ to the nodes in $\text{SRC}(C)$
   3) If $\text{TGT}(C)$ is contained in $\text{TGT}(I)$ then
      Split $C$ into $I$ and $C'$
      Add $I$ and $C'$ to the concentration list
   4) If $\text{TGT}(I)$ is contained in $\text{TGT}(C)$ then
      Split $I$ into $I'$ and $C$
      Add $I'$ and $C$ to the concentration list
   5) If $I$ has no intersection with any $C$ then
Create a new concentration
e. Merge edge concentrations that have the same set of target nodes. Discard those where size < M.

2. Objective: Update the successor list with information from the concentrations.
a. For each concentration C in the concentration list
   1) Find each successor list for the source nodes SRC(C)
      a) Delete the successors TGT(C) for each node SRC(C)
      b) Add successors EC for each node SRC(C)
   2) Create new successor list EC and add successors for each node TGT(C)

The algorithm works by grouping programs (source programs) that call the same program (target programs). The algorithm tries to find a set of subgraphs such that the target nodes of a subgraph are not allowed to be a subset of the target nodes of any other subgraph. The algorithm works with two lists: a list of intersections and a list of concentrations. An intersection is the subgraph formed by a pair of source nodes and the intersection of their successors. A concentration is a subgraph such that all source nodes are connected to all target nodes.

The algorithm can easily be generalized by replacing the attribute of "calls program X" with any other attribute.

End User Interface

This algorithm operates on a set of source (calling) nodes and target (callee) nodes. Its performance characteristics make it unsuitable for indiscriminate use. Therefore, it should be applied carefully. An approach might be to allow the user to select the target row and apply the algorithm against the selected target row and all of the callers associated with the nodes in the row.

Path analysis

Assume that we are discussing vertices contained in a graph. There are three general questions which can be asked:

1. For a single source (s) and an arbitrary vertex (v), compute an unambiguous path expression P(s,v) which contains all paths from s to v.
2. For a single sink/exit (t) and an arbitrary vertex (v), compute an unambiguous path expression P(v,t) which contains all paths from v to t.
3. For all pairs of vertices v,w, compute an unambiguous path expression which contains all paths from v to w.

This algorithm uses both the dominator tree and another structure called a derived graph. The derived graph G' contains edges defined as follows:

- For any edge e, let e be an edge, such that t(e)=t(e) and h(e)=h(e) if h(e) is the immediate dominator of t(e).
- If t(e) is not the immediate dominator of h(e) then h(e)=u where the immediate dominator of t(e),idom(t(e)), is a parent of u, and u is an ancestor of h(e) in the dominator tree.

There are three kinds of edges in the derived graph G':

1. If t(e) is an immediate dominator of h(e) then e = e.
2. If t(e) is an ancestor of h(e) in the dominator tree then e is in a loop.
3. If t(e) is not an immediate dominator or an ancestor of h(e) then e connects two siblings in the dominator tree. This means e connects two different logic flows in the program.

Data Structures

The algorithm use the following data structures created in the dominators algorithm:

Computed

- dom(u) is the immediate dominator of vertex u.
- Children(u) For each vertex u in the dominator tree, the set of vertices v, such that idom(v) = u.
- Tree(u) For a vertex u in the dominator tree, the set of edges, such that t(e) = u and h(e) = idom(u). This is sometimes called tree(u).
- Non-tree(u) For a vertex u in the dominator tree, the set of edges, such that t(e) = u and h(e) ≠ idom(u).
- e For each edge e in the dominator tree, the corresponding edge e in the derived graph G'.

Algorithm

The algorithm assumes that the dominator tree of G is known and the vertices are numbered from 1 to n so that the dominator of V has a greater number than V.

The idea is to break the problem graph into subgraphs, apply a procedure called ELIMINATE to construct a path sequence for each subgraph, and combine these path sequences into a path sequence for the original graph. The dominator tree defines the single entry regions of the graph G. The derived graph, G', is used in computing path sequences for each single entry.
section of the graph G, called dominator strong components of G', and to combine these path sequences to form a total path sequence for G.

The algorithm groups together vertices with a common parent and processes these sibling sets in increasing order by parent. The algorithm processes the set of siblings, dom(u), for each vertex as follows:

- For each edge e such that h(e) is a child of u in the dominator tree, the algorithm uses a procedure EVAL_AND_SEQUENCE to compute a path expression P(e) representing all paths in G from h(e) to t(e) which end with edge e and contain only proper descendants of h(e) as intermediate vertices.
- The algorithm computes a path sequence Xu for the graph described by dom(u).
- Substituting P(e) for each edge e appearing in these path sequences produces a sequence Yu that represents every path in G starting and ending at a child of u and containing only proper descendants of u as intermediate vertices.

The algorithm concatenates Yu onto the end of the path sequences. By applying procedure SOLVE to Yu, it computes for each child (v) of u a path sequence which represents all paths in G from u to v. The final steps are to invoke procedures UPDATE(u,v) and LINK(u,v) to add the sequences to the set of path expressions for the graph.

**End User Interface**

The simplest example of using this function is finding all paths which contain a given node. In this example, the cursor is used to select a node in the graph. All paths from the root to the selected node and all paths from the selected node to each sink are highlighted on the original graph display.

A single path in the original graph may be highlighted by selecting the desired nodes with the cursor. If the selected nodes are not contiguous, then all paths between the two nodes will be highlighted.

A text window may be created to display the source code contained in the nodes of the selected path(s).

**Determine data flow in a path**

We are interested in determining, for each basic block of the program, facts which must be true on entry to the block, regardless of the actual paths of program execution. We can define a dependency graph corresponding to the interdependencies of variables in the system. Each node represents one or more variables; each directed edge (m,n) represents the dependence of Xn on Xm. For forward data flow problems, the dependency graph is the flowgraph of the program.

For the code fragment and control flowgraph:

```
K: y = 99999;
if x > z
  then y = a;
else
  if z > 3 * w
    then y = b;
L: q = 2 * y; /* can y = 9999 here? */
```

**Figure 4. program fragment**

```
1 y=9999
2 x>z -> 3 y=a ->
4 z>3*w -> 5 y=b ->
6 q=2 <-
```

**Figure 5. control flow for program fragment**

The definition of y at statement K: in node 1 can be propagated to statement L: along the path of nodes (1,2,4,6) but not along path (1,2,3,6) because the definition at node 3 blocks the earlier definition.

The problem of reaching definitions can be expressed concisely using equations. Let Pj be the set of all definitions of y in the program, if y is not defined at node j, and the empty set otherwise. Let dj be the set of all definitions of y created at node j, if there are definitions of y at node j, and the empty set otherwise. Then Xi, the set of definitions of y that reach node i, can be expressed by:

```
X_1 = 0
X_2 = p_2 \cap X_1 \cup d_1
X_3 = X_4 = p_2 \cap X_2 \cup d_2
X_4 = p_4 \cap X_4 \cup d_4
X_5 = (p_2 \cap X_3 \cup d_3) \cup (p_4 \cap X_4 \cup d_4)
    \cup (p_5 \cap X_5 \cup d_5)
```

**Figure 6. equations for code fragment and control flowgraph**

**General Algorithm**

A general algorithm for data flow is
\[
X_m = \{pj \cap X_j U dj\} \ j \text{ is an element of pred}(m)
\]

Where:
1. \(\text{pred}(m)\) is the set of all immediate predecessors of \(m\);
2. \(X_j\) is the set of all variable definitions reaching the entry to \(j\);
3. \(pj\) is the set of all variable definitions that may be preserved through node \(j\) (the set of all definitions not redefined at \(j\));
4. \(dj\) is the set of locally exposed definitions at node \(j\), that is, the set of last definitions of each variable defined at node \(j\).

Available Expressions

An expression such as \(A + B\) is available at point \(p\) in a flowgraph if every sequence of branches which the program may take to \(p\) causes \(A + B\) to have been computed after the last computation of \(A\) or \(B\).

Let \(E\) be the set of expressions computed in the flowgraph.

Let \(K\) be the set of expressions killed in node \(x\). Expression \(A*B\) is killed if either \(A\) or \(B\) is assigned within node \(x\). (The symbol \(*\) indicates a genic binary operator.)

Let \(G\) be the set of expressions generated in node \(x\). If expression \(r = A*B\) is in \(G(x)\) and neither \(A\) nor \(B\) are subsequently assigned within \(x\), then \(r\) is generated in \(x\).

Let \(AEIN(x)\) and \(AEOU(x)\), for each node \(x\), be the set of expressions available at entrance to and at exit from node \(x\).

There are some fundamental relationships which enable us to compute \(AEIN(x)\) for each node \(x\):

\begin{align*}
\text{AE1} &: \text{AEIN}(r) = \text{null} \\
\text{AE2} &: \text{For } x \neq r, \text{AEIN}(x) \text{ is the intersection of } AEOU(y) \text{ over all predecessors } y \text{ of } x. \\
\text{AE3} &: AEOU(x) = (\text{AEIN}(x) \cdot (K(x)^*) \cup G(x)^*) \text{ for each node } x.
\end{align*}

That is, \(AEOU(x)\) is the set of expressions available at the entry of the node minus those expressions killed in the node, plus those expressions generated in the node.

AE4 Since \(AE1, AE2, AE3\) do not necessarily have a unique solution for \(AEIN(x)\), we want the largest solution.

Data Structures

Input

\(\text{Pred}(y)\) For each node \(y\), \(\text{Pred}(y)\) is a predecessor list such that \(\text{Pred}(y) = (z|z(x,y)\text{ is an edge in the graph. The nodes are referred to by the number generated in the DFST algorithm}.\)

\begin{align*}
\text{NOTKILL}(x) & : \text{NOTKILL} \text{ is a bit vector where the } i\text{th bit of of NOTKILL is } 1 \text{ if and only if the } i\text{th expression is not killed in node } x. \text{ All bit vectors have length } p, \text{ where } p \text{ is the number of expressions.} \\
\text{GEN}(x) & : \text{GEN} \text{ is a bit vector where the } i\text{th bit of of GEN is } 1 \text{ if and only if the } i\text{th expression is not generated in node } x. \text{ All bit vectors have length } p, \text{ where } p \text{ is the number of expressions.}
\end{align*}

Output

\(\text{AEIN}(x)\) The set of expressions available at the entry to node \(x\).

Algorithm

\begin{align*}
\text{DCL previous} & : \text{bit}(p), /*p \text{ is the shr of expressions}*/ \\
\text{DCL AEIN(n)} & : \text{bit}(p), /*n \text{ is the shr of nodes}*/ \\
\text{AE}(i) = \text{all 0} & : /* \text{nothing available at the root} */ \\
\text{Do } j = 2 \text{ to } n & : /* \text{numbering from root to bottom} */ \\
\text{previous} & = \text{AEIN}(j) \text{ initialze other to all 1's} \\
\text{end} & : \\
\text{change} & = \text{true} \\
\text{do while } (\text{change} = \text{true}) & : \\
\text{change} & = \text{false} \\
\text{end} & : \\
\text{Do} & : j = 2 \text{ to } n \\
\text{previous} & = \text{AEIN}(j) \text{ numbering from root to bottom} \\
\text{AEIN}(j) & = \text{previous} \text{ previous } (\text{AEIN}(k) \wedge \text{NOTKILL}(k)) \text{ } \lor \text{GEN}(k) \text{ } \text{If previous } \neq \text{AEIN}(j) \\
\text{then} & : \text{change} = \text{true} \\
\text{else} & : \\
\text{end} & : \\
\text{and} & : \wedge \text{ is intended to be the AND function (bitwise product)} \\
\text{or} & : \lor \text{ is intended to be the OR function (bitwise sum)}
\end{align*}

Live Variables

A path in a flowgraph is called definition-clear with respect to a variable \(v\) if there is no definition for \(v\) (by assignment or input) on the path. A variable \(v\) is live at a node \(x\) in flowgraph if there is a definition-clear path for \(v\) from node \(x\) to a use of \(v\) in some other node \(y\). That is, \(v\) is live if there is a potential use of \(v\) before \(v\) is redefined. The definition of \(v\) reaches node \(p\) if there is a definition-clear path from node \(n\) to node \(p\).

Let \(V\) be the set of live variables occurring in a flowgraph \(G = (V,E,r)\)

Let \(C(x)\) be the clear set of node \(x\). The clear set is the set of variables which are not defined in node \(x\).

\(U(x)\) is the set of variables which have exposed uses in node \(x\), that is, those variables with a definition-clear path from the entry of node \(x\) to a use within node \(x\).
Let \( \text{LVOUT}(x) \) and \( \text{LVIN}(x) \) for each node \( x \) be the set of variables live at exit from and on entrance to node \( x \).

Some fundamental relationships:

**LV1**
If \( x \) has no successors, then \( \text{LVOUT}(x) = \text{null} \).

**LV2**
Otherwise, \( \text{LVOUT}(x) \) is the union of \( \text{LVIN}(y) \) over all successors \( y \) of \( x \).

**LV3**
\[
\text{LVIN}(x) = \text{LVOUT}(x) \cap \text{"C}(x)"
\]
\[
\cup \text{"U}(X)" \quad \text{for each node} \quad x
\]

**LV4**
Since \( \text{LV1}, \text{LV2}, \text{LV3} \) do not necessarily have a unique solution for \( \text{LVOUT}(x) \), we want the smallest solution.

\( \text{AEIN} \) and \( \text{LVOUT} \) are not the same algorithms. \( \text{LVOUT} \) propagates information opposite to the direction of control flow. \( \text{AEIN} \) propagates information in the direction of control flow. It is important to note that the nodes are processed in reverse order in \( \text{LVOUT} \).

**Data Structures**

**Input**
- For each node \( y \), \( S(y) \) is a successor list such that \( P(y) = \{ z(x,y) \mid \text{is an edge in the graph} \} \). The nodes are referred to by the number generated in the DFS algorithm.
- \( \text{CLEAR}(x) \) is a bit vector where the \( t \)th bit of \( \text{CLEAR}(x) \) is 1, if and only if the corresponding variable is not defined in node \( x \). All bit vectors have length \( q \), where \( q \) is the number of variables.

**Output**
- \( \text{XUSE}(x) \) is a bit vector where the \( t \)th bit of \( \text{XUSE}(x) \) is 1, if and only if the corresponding variable has a definition-clear path from the entry of node \( x \) to a use in node \( x \). All bit vectors have length \( q \), where \( q \) is the number of variables.

**Algorithm**
The algorithm consists of \( \text{LV3} \) 'plugged into' \( \text{LV2} \) with an appropriate initialization for \( \text{LVOUT} \) and a different node ordering from \( \text{AEIN} \).

```
DCL previous bit(p); /*p is the nbr of expressions*/
DCL LVOUT(n) bit(q); /*n is the nbr of nodes*/

Do j = 1 to n
    LVOUT(j) = 0's; /*initialize other to all 0's*/
end;

Change = true;

do while (change = true);
    change = false;
    Do j = n to 1, by -1 /*reverse numbering*/
        previous = LVOUT(j);
        LVOUT(j) =
            \text{for all} \quad k \text{ in Succ}(j) \quad ((\text{LVOUT}(k) \text{ AND } \text{CLEAR}(k))
            \text{OR } \text{XUSE}(k));
        If previous \# \text{LVOUT}(j)
            then
                change = true;
            else;
            end;
    end;
end;
```

\( \text{AND} \) is intended to be the AND function (bitwise product)
\( \text{OR} \) is intended to be the OR function (bitwise sum)

**End User Interface**

One example of this function is reaching analysis. The graph will be highlighted to show the paths in which a data element is not changed. The user selects a node in a graph and all data elements used in the node are listed. The user may associate colors with each data element or cluster of data elements. Those nodes which use the selected data elements without changing the values are shaded with the color(s) associated with the data element(s) selected.

If the user selects a path from the original graph, the data values for each node in the path can be displayed. Sometimes the user will be asked to provide some values which were determined outside of the scope of the selected path.

Display the data flow for a data element(s). The user selects a data element(s) and associates color(s) with the data. The graph is displayed with all uses and transformations of the data highlighted in color. This means that if data element ZIP is selected and at some point in the program the value of ZIP is moved into ZCODE then all subsequent uses of ZCODE will also be highlighted.

Data flow can be very computationally expensive. The algorithm given above is intended for illustration purposes. It can be used for prototypes and for small exercises. A robust or commercial implementation of a data flow application will probably require a specialized parsing and analysis techniques rather than the general purpose algorithm.
Appendix A. Data Structures

Data about Nodes

Nodes
AEIN(x) The set of expressions available at the entry to node x.
AEOUT(x) The set of expressions available at the exit from node x.
Ancestor(v) An array used by the LINK procedure. Initially Ancestor(v)=0. In general Ancestor(v)=0 only if v is a tree root in the forest; otherwise Ancestor(v) is an ancestor in the forest.
bucket (w) A set of vertices whose semidominator is w.
Children(u) For each vertex u in the dominator tree, the set of vertices v such that idom(v) = u.
CLEAR(x) CLEAR is a bit vector where the zth bit of CLEAR(x) is 1 if and only if the corresponding variable is not defined in node x. All bit vectors have length q, where q is the number of variables.
Cycle(v) The set of edges which were deleted when applying transform Ti.
dom(w) dom(w) is the immediate dominator of vertex w.
GEN(x) GEN is a bit vector where the zth bit of of GEN is 1, if and only if the zth expression is not in 'G(x)'. All bit vectors have length p, where p is the number of expressions.
HIGHPT(y) For each vertex, HIGHPT(y) is the head node for all nodes y contained in a reachunder set. In reachunder set Ix all nodes y will have HIGHPT(y) = x.
Label(v) An array used in the LINK procedure. Label(v) = v for each vertex v.
LVOUT(x) The set of variables live at exit from node x.
LVIN(x) The set of variables live at entrance to node x.
Non-Cycle(v) The set of edges which were deleted when applying transform T2.
Non-tree(u) For a vertex u in the dominator tree, the set of edges such that t(e) = u and h(e) ≠ idom(u).
NOTKILL(x) NOTKILL is a bit vector where the zth bit of of NOTKILL is 1, if and only if the zth expression is not in 'K(x)'. All bit vectors have length p, where p is the number of expressions.
parent(w) The vertex which is the parent of vertex w in the spanning tree.
pred(y) For each node y, Pred(y) is a predecessor list such that Pred(y) = x and (x,y) is an edge in the graph. The nodes are referred to by the number generated in the DFST algorithm.
semi(w) A number defined as follows:
1. Before vertex w is numbered, semi(v) = 0.
2. After w is numbered but before its semidominator is computed, semi(w) is the number of w.
3. After the semidominator of w is computed, semi(w) is the number of the semidominator of w.
SNUMBER(y) The nodes are numbered by an ancestor first, right to left traversal of the DFST.
succ(x) For each node x, Succ(x) = y is a successor list, such that (x,y) is an edge in the graph. The nodes are referred to by the number generated in the DFST algorithm.
Tree(u) For a vertex u in the dominator tree, the set of edges, such that t(e) = u and h(e) = idom(u). This is sometimes called tree(u).
vertex(t) The vertex whose number is t.
XUSE(x) XUSE is a bit vector where the zth bit of of XUSE(x) is 1, if and only if the corresponding variable has a definition-clear path from the entry of node x to a use in node x. All bit vectors have length q, where q is the number of variables.

Data about Edges

Edges
Concentration A concentration is a set of source and target nodes such that all the source nodes in the set are connected to all the target nodes in the set.
\( \epsilon \) For each edge e in graph G, the corresponding edges \( \epsilon \) in the derived graph \( G' \).
Intersection

An intersection consists of a set of source nodes (callers) and target (called) nodes. An intersection is formed by a pair of source nodes and the intersection of their successor nodes.

Tree Structures

Trees

Depth-First Spanning Tree (DFST) A tree T rooted at r, with vertices numbered in pre-order.

Intervals I(h) The set of vertices v, such that (v,w) is an edge in a strongly connected subgraph of G. Vertex h is the head or root of the interval.

reachunder(x) An interval for which the head is the target of a back edge.

Paths

Paths

P(é) A path expression representing the paths in G from h(é) to t(é) which end with edge e.

Q Represents all paths in an interval from v to v which do not contain v as an intermediate vertex.

(Q*) Represents all paths in an interval from v to v.

R(v) A path expression representing the paths in G from idom(v) to v.

S(v) A path expression representing all paths from ancestor(v) to v, all of whose intermediate vertices are smaller than ancestor(v).

Sequence A path sequence for G containing the triples (S(v),ancestor(v),v).

Xu A path sequence for the subgraph Gu of G, derived from children(u).

Yu For each vertex u in G, Yu is a sequence(P1,v1,w1),(P2,v2,w2),...,(Pn,vx,wx).

Bibliography


