Dominators, Super Blocks, and Program Coverage

Hiralal Agrawal
Bellcore
445 South Street
Morristown, NJ 07960
hira@bellcore.com

Abstract

In this paper we present techniques to find subsets of nodes of a flowgraph that satisfy the following property: A test set that exercises all nodes in a subset exercises all nodes in the flowgraph. Analogous techniques to find subsets of edges are also proposed. These techniques may be used to significantly reduce the cost of coverage testing of programs. A notion of a super block consisting of one or more basic blocks is developed. If any basic block in a super block is exercised by an input then all basic blocks in that super block must be exercised by the same input. Dominator relationships among super blocks are used to identify a subset of the super blocks whose coverage implies that of all super blocks and, in turn, that of all basic blocks. Experiments with eight systems in the range of 1-75K lines of code show that, on the average, test cases targeted to cover just 29% of the basic blocks and 32% of the branches ensure 100% block and branch coverage, respectively.

1 Introduction

Common sense dictates that a program be tested on enough inputs that exercise each of its statements at least once, as a statement must be exercised before a fault in it may be exposed [5, 14]. In other words, a program should be executed on enough inputs during testing so each node in its flowgraph gets visited. A tester, therefore, is faced with the task of creating test cases targeted to "cover" all nodes in the flowgraph of the program being tested. In this paper, we present a technique to find a small subset of these nodes with the property that if the subset is covered, the remaining nodes are automatically covered. Thus the tester only needs to develop test cases targeted to cover the nodes in the subset rather than the entire set.

The problem that all nodes in a flowgraph be covered is referred to as the *block coverage problem* as nodes in a flowgraph represent basic blocks in a program[1]. A more stringent problem than the block coverage problem is the *branch coverage problem* which requires all control transfers—or branches—among basic blocks to be exercised. In other words, it requires that the program should be executed on enough inputs so all edges in its flowgraph get visited at least once. The techniques presented in this paper apply equally well to the branch coverage problem. They may be used to identify a small subset of edges such that if the subset is covered then other edges are automatically covered.

Besides saving the user time, these techniques may also be used to reduce the space and time overhead imposed by the coverage testing tools by reducing the number of probes that need to be placed in a program. Many optimal program profiling techniques are available in the literature [4, 7, 11, 15, 17] that may also be used to reduce this overhead (see Section 8, Related Work). Unfortunately none of these techniques help reduce the number of test cases the user must develop to achieve the desired coverage. These techniques help us find a small subset of nodes/edges in a flowgraph with the property that if we know how many times the nodes/edges in the subset are visited, then we may infer how many times the remaining nodes/edges are visited. If all nodes/edges in the subset are visited, however, we may not conclude that the remaining nodes/edges are visited too. The subsets of nodes/edges determined by the techniques presented in this paper, on the other hand, enable the latter inference to be made.

Pre- and postdominator relationships among flow-

This paper was published in the *Conference Record of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 16–19, 1994, pages 25–34.

[1]It is same as the *statement coverage problem* as covering all basic blocks implies covering all statements, and vice versa.

```
e1;
while (e2) {
   switch (e3) {
      case 1: e4;
                break;
      case 2: e5;
                while (e6) e7;
      default:
         if (e8) {
            e9;
            continue;
         }
         do e10; while (e11);
         e12;
   }
   e13;
}
e14;
```

Figure 1: An example C program

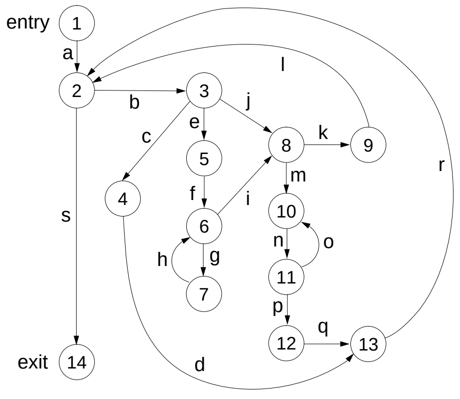

Figure 2: Control flowgraph

graph nodes are used to partition the set of nodes into "super blocks" with the property that if any node in a super block is covered then all nodes in that super block are covered. Super blocks are different from basic blocks in that a super block may contain several basic blocks. A dominator relationship among super blocks is defined and used to identify a subset of the super blocks such that covering the subset implies covering all super blocks and, in turn, covering all basic blocks. Analogous techniques are developed to partition the set of edges and determine a subset of the partitions such that covering the latter implies covering the former.

Experiments using these techniques on eight systems varying in size from a 1,000 to 75,000 lines of C code indicate that these techniques are quite promising (see Section 7, Experimental Results). It was found that, on the average, test cases targeted to cover just 29% of the basic blocks and 32% of the branches chosen by these techniques ensure 100% block and branch coverage, respectively. Moreover, these techniques also provide an ordering among the targeted basic blocks so covering the first one third of the targeted blocks, on the average, implies covering more than two thirds of all blocks.

In the next section, we briefly review the terminology used in the this paper. In Section 3, we discuss techniques to identify a subset of the basic blocks whose coverage implies that of all basic blocks. Then, in Section 4, we describe how to find the order in which the basic blocks identified should be covered. Section 5 discusses how these techniques may be used to reduce the time and space overhead involved in computing the coverage attained by a given test set. In Section 6, we present how the analogous techniques may be used to expedite branch coverage. Section 7 includes the results of our preliminary experiments and Section 8 compares our work with the related work in the literature.

2 Background

A control flowgraph, or simply, a flowgraph, of a program is a four-tuple (N, E, *entry*, *exit*) where N is the set of nodes that correspond to basic blocks in the program, E is the set of directed edges between nodes, and *entry* and *exit* are two distinguished nodes in N. Every node in N is reachable from the *entry* node and the *exit* node is reachable from every node by following edges in E. Figure 2 shows the flowgraph of an example C program, shown in Figure 1, where $e1, e2, \ldots, e14$ are blocks whose contents are not relevant for our purposes.

A node, u, *predominates* a node, v, denoted as $u \xrightarrow{pre} v$, if every path from the *entry* node to v contains u[2]. A node, w, *postdominates* a node, v, denoted as $w \xrightarrow{post} v$, if every path from v to the *exit* node contains w. For example, in Figure 2, nodes 1, 2, and 3 predominate node 8 and nodes 2 and 14 postdominate it. Pre- and postdominator relationships can be expressed in the form of pre- and postdominator trees, respectively. $u \xrightarrow{pre} v$ *iff* there is s path from u to v in the predominator tree. Similarly, $w \xrightarrow{post} v$ *iff* there is path from w to v in the postdominator tree. Figures 3 and 4 show the pre- and postdominator trees of the flowgraph in Figure 2.

Many algorithms have been developed to find predominator relationships among nodes of a flowgraph [3, 9, 12, 16]. As the postdominator relationship is the same as the predominator relationship in the reverse flowgraph, the same algorithms may also be used to find postdominator relationships. The predominator tree of a flowgraph may be built in $O(N + E)$ time, where N and E denote the number of nodes and edges in the

[2]The predominator relationship is also referred to as the dominator relationship at other places in the literature. In this paper, we refer to it as the predominator relationship to clearly distinguish it from the postdominator relationship. We use the term dominator relationship here to mean either pre- or postdominator relationships.

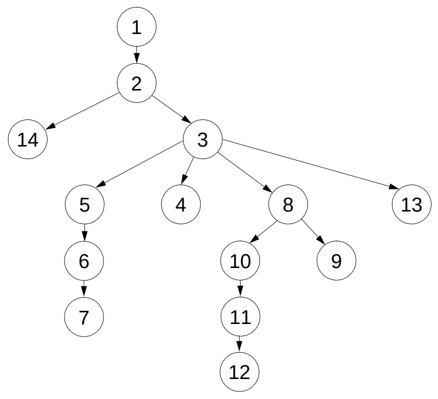
Figure 3: Predominator tree

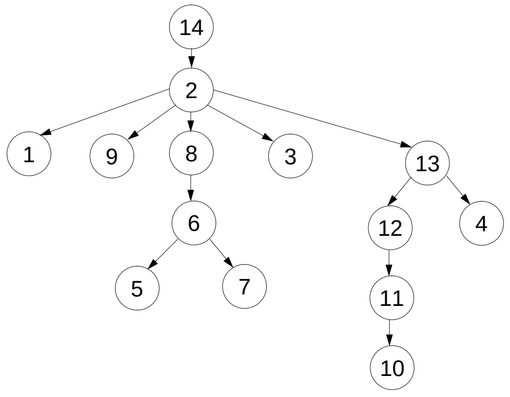
Figure 4: Postdominator tree

flowgraph, respectively [9]. The same applies for the postdominator tree.

3 Expediting Block Coverage

3.1 Pre- and Postdominators

A node, u, in a flowgraph is said to be *covered* by a test case, t, denoted as *covered(u, t)*, if the control reaches the corresponding basic block at least once when the program is executed on t. If $u \xrightarrow{pre} v$ and v is covered by t then u must also be covered by t, as every execution path from the entry node to v contains u. In other words:

$$if\ u \xrightarrow{pre} v\ then\ covered(v,t) \Longrightarrow covered(u,t) \tag{1}$$

Equivalently, whenever a node in a flowgraph is covered then all its ancestors in the predominator tree are also covered. Thus, if all leaves in the predominator tree are covered then all other nodes in the flowgraph are automatically covered. A tester, therefore, only needs to develop test cases meant to exercise the leaves of the predominator tree.

The same arguments hold if we substitute the postdominator relationship wherever the predominator relationship was used in the previous paragraph, *provided* the program execution terminates normally on all test cases supplied. Thus we also have the following, provided the termination condition holds:

$$if\ u \xrightarrow{post} v\ then\ covered(v,t) \Longrightarrow covered(u,t) \tag{2}$$

The condition that the execution normally terminate on all test cases supplied does not cause any adverse problems for our purposes here. The goal of testing is to detect faults in a program. If the tester supplies a test case on which the execution does not terminate, or terminates abnormally, it is indicative of a fault in the program. The fault should then be fixed so the corrected program terminates normally on the same test case[3]. Therefore, in the remainder of this paper we assume that the program execution terminates normally on all test cases supplied by the tester.

A coverage testing tool may build both the pre- and postdominator trees of the program under test, compare the number of leaves in the two trees, and have the tester cover the smaller of the two sets. For example, for the flowgraph in Figure 2, the predominator tree in Figure 3 has six leaves whereas the postdominator tree in Figure 4 has seven leaves. Thus, in this case, a tester only needs to develop test cases targeted to cover the six leaves in the predominator tree.

3.2 Super Block Dominators

We say that a node, u, *dominates* a node, v, denoted as $u \longrightarrow v$, if every path from *entry* to *exit*, via v, contains u. Clearly, u dominates v *iff* u pre- or postdominates v, for if this were not true, we would have a path from the *entry* to the *exit* node via v bypassing u altogether. In other words:

$$u \longrightarrow v\ iff\ (u \xrightarrow{pre} v\ or\ u \xrightarrow{post} v) \tag{3}$$

Thus, the dominator relationship among flowgraph nodes is the union of pre- and postdominator relationships. It may be represented graphically by merging the pre- and postdominator trees of the flowgraph. We call the union of the pre- and postdominator trees as the *basic block dominator graph* of the corresponding flowgraph. Figure 5 shows the basic block dominator graph of the flowgraph in Figure 2, obtained by merging the pre- and postdominator trees of Figures 3 and 4, respectively. A node, u, dominates another node, v, in a flowgraph *iff* there is a path from u to v in its basic block dominator graph. As the name suggests, a basic

[3]Changes to a program may require that its flowgraph as well as the pre- and postdominator trees be rebuilt.

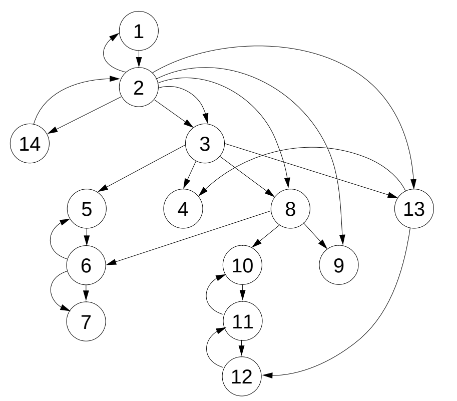

Figure 5: Basic block dominator graph

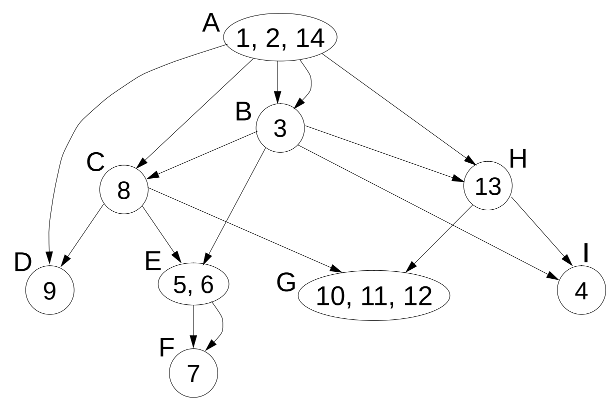

Figure 6: The graph obtained by merging the strongly connected components of the basic block dominator graph

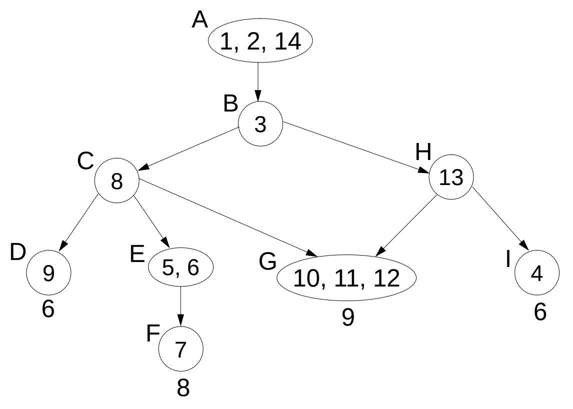

Figure 7: Super block dominator graph with the initial weights associated with the leaves.

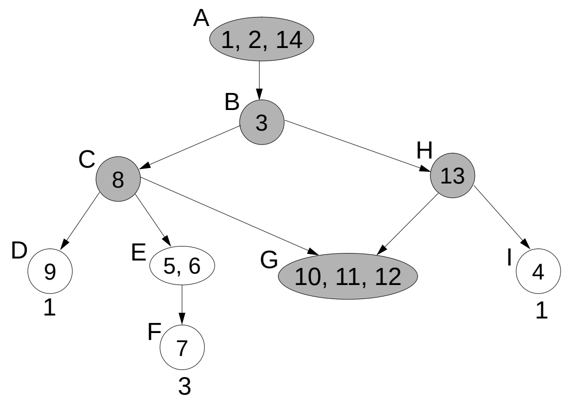

Figure 8: Highlighted blocks show the covered blocks after the initially heaviest leaf is covered. New weights of the remaining leaves are also shown.

block dominator graph may not necessarily be a tree. Moreover, it may not be acyclic either.

A strongly connected component of a basic block dominator graph has the property that every node in the component dominates all other nodes in that component. For example, nodes $\{1, 2, 14\}$ form a strongly connected component of the basic block dominator graph in Figure 5. Strongly connected components of a graph may be found in $O(N + E)$ time where N and E denote the number of nodes and edges in the graph, respectively [2].

Assertions (1), (2), and (3), in turn, imply:

$$\textit{if } u \longrightarrow v \textit{ then } covered(v,t) \Longrightarrow covered(u,t) \qquad (4)$$

This, coupled with the fact that each node in a strongly connected component dominates all other nodes in that component, implies that whenever any node in a component is covered by a test case then all other nodes in that component must be covered by the same test case. We refer to the strongly connected components of a basic block dominator graph as *super blocks*, as they are made up of one or more basic blocks. Any one basic block in a super block may be designated as the *representative* of that super block.

Unlike basic blocks super blocks need not be contiguous. They may have multiple entry and multiple exit points. If a program instruction inside a basic block is executed, all other instructions in that basic block must be executed before the control leaves that basic block. On the other hand, if a program instruction in a super block is executed, all instructions in that super block must be executed before the exit node is reached.

Just as with basic blocks, we also define dominator relationships among super blocks. A super block, U, *dominates* another super block, V, denoted as $U \longrightarrow V$, if every path from the entry to the exit node via V in the flowgraph also contains U. Thus we also have:

$$\textit{if } U \longrightarrow V \textit{ then } covered(V,t) \Longrightarrow covered(U,t) \qquad (5)$$

Recall that if a path contains any basic block in a su-

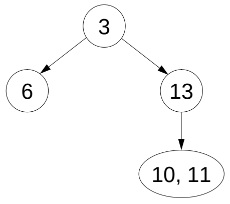

Figure 9: Super block dominator graph involving a subset of basic blocks.

per block then it contains all basic blocks in that super block.

Dominator relationships among super blocks may be represented graphically in the form of a *super block dominator graph*. It is obtained by merging the nodes in the strongly connected components of the corresponding basic block dominator graph and removing the composite edges from the resulting graph[4]. An edge, e, from a node, u, to a node, v, is said to be a *composite edge* if v is also reachable from u without going through e. In case there are multiple edges from one node to another, all but one of them are considered composite edges. In other words, a composite edge represents a composition of one or more other edges.

Figure 6 shows the graph obtained by merging the strongly connected components of the graph in Figure 5. It has several composite edges, e.g., the edges from node A to nodes C, D, and H are all composite edges. One of the two edges from node A to node B is also a composite edge. Figure 7 shows the super block dominator graph of the basic block dominator graph in Figure 5, obtained by removing the composite edges in the graph in Figure 6. Note that unlike a basic block dominator graph a super block dominator graph is an acyclic graph. But unlike pre- and postdominator trees it need not be a tree. It can be shown that each basic block dominator graph has a unique super block dominator graph.

Assertion (5) implies that covering all the leaves in the super block dominator graph implies covering all other super blocks as well. A tester, therefore, only needs to develop test cases aimed at covering one basic block from each leaf in the super block dominator graph. For example, for the flowgraph in Figure 2, one only needs to create test cases that cover basic blocks 4, 7, 9, and 10—one each from the leaf nodes in its super block dominator graph in Figure 7.

[4]Equivalently, the super block dominator graph of a flowgraph may be obtained by finding the condensed graph of its basic block dominator graph and obtaining the minimum equivalent graph of the condensed graph [13].

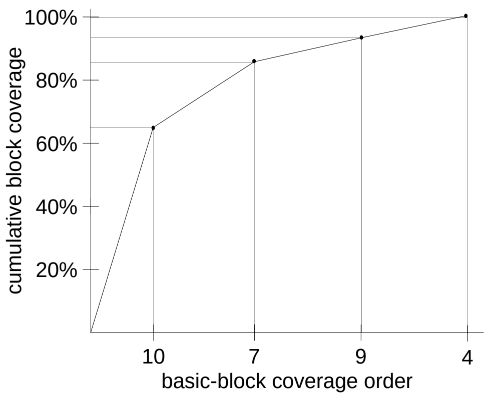

Figure 10: The order in which the targeted basic blocks are covered and the corresponding cumulative coverages achieved.

3.3 Selective Coverage

Sometimes a tester may need to cover a set of specific basic blocks rather than the set of all basic blocks. In this case a new dominator graph may be obtained by projecting the program's super block dominator graph over the given basic blocks. The tester then only needs to develop test cases that cover the leaves of the projected dominator graph.

For example, suppose we need to develop test cases that cover nodes 3, 6, 10, 11, and 13 in the flowgraph in Figure 2. Figure 9 shows the super block dominator graph obtained by projecting that in Figure 7 over the above five blocks. As the projected graph has only two leaves, we only need to develop test cases that cover the corresponding two basic blocks—6 and 10, or alternatively, 6 and 11.

4 Deciding the Right Order

Oftentimes, due to the lack of time and other resources, testers are content with attaining some desired fraction of 100% coverage, e.g., 70% or 80%. In such situations, how does one find the smallest subset of the leaves of the super block dominator graph whose coverage implies the desired overall coverage?

The above problem, in general, is an NP-complete problem [6]. But we can use the greedy approach to get good approximate answers. We can associate weights with the leaves of the super block dominator graph. *Weight* of a leaf is defined to be the total number of basic blocks in the leaf and all its *uncovered* ancestors in the super block dominator graph. Initially, all super blocks are marked as uncovered. Figure 7 also shows the initial weights associated with the leaves of the super block dominator graph.

We select the leaf with the largest weight to be cov-

```
Probe(U)
{
    If U has fewer than two children in the super block dominator graph
    Then return true;

    /* check if there exists a path via U that bypasses all super blocks dominated by U */
    Mark all nodes in the flowgraph as unvisited;
    Mark a representative basic block, r, of U as visited;
    Mark representatives of the children of U in the super block dominator graph as visited;

    Visit_predecessors(r);
    Visit_successors(r);

    If both the entry and the exit nodes of the flowgraph are marked as visited
    Then return true;
    Else return false;
}
```

```
Visit_predecessors(n)
{
    For each immediate predecessor, p,
    of node n in the flowgraph Do
        If p is not marked as visited Then
        {
            Mark p as visited;
            Visit_predecessors(p);
        }
}
```

```
Visit_successors(n)
{
    For each immediate successor, s,
    of node n in the flowgraph Do
        If s is not marked as visited Then
        {
            Mark s as visited;
            Visit_successors(s);
        }
}
```

Figure 11: An algorithm to check if a super block should be probed

ered first as covering it implies maximum number of basic blocks to be covered. If there are more than one heaviest leaves any one of them may be selected. Then the selected leaf and all its ancestors are marked as covered. Next, the weights of the remaining uncovered leaves are recomputed and the heaviest leaf among these is selected to be covered. This process continues until the desired coverage level has been achieved. Figure 8 shows the super block dominator graph with covered blocks highlighted after the initially heaviest leaf is covered. It also shows the new weights of the remaining leaves.

The graph in Figure 10 shows, along its horizontal axis, the order in which the desired basic blocks of the flowgraph in Figure 2 should be covered[5]. Along the vertical axis, it shows the cumulative overall coverage after each of these blocks is covered. Note that covering just two (14%) of the basic blocks, viz., 10 and 7, gives us more than 85% coverage and covering two more (29%) ensures that all fourteen basic blocks are covered. This means *at most* four test cases need to be developed to cover all basic blocks. In practice, however, even fewer test cases may be needed as a test case developed to cover a leaf may cover other blocks besides the leaf and its ancestors. In particular, techniques described in [8] may be used to further reduce the number of test cases needed to cover the basic blocks identified (see Section 8). For example, for the flowgraph in Figure 2, it may be possible to create a single test case that covers all basic blocks. Thus, the curve in Figure 10 gives us a *lower bound* on the cumulative coverage levels achieved by following the above technique.

[5] Alternatively, the order—10, 7, 4, 9—may be used. Multiple orderings are possible whenever at any step in the process more than one leaves have the largest weight.

5 Selecting the Right Probes

In order to determine which basic blocks are covered by a test set, coverage tools insert probes at various program locations, usually one per basic block (see, however, Section 8) [10]. Probes obviously increase the size of the program's object code as well as its execution time. If we reduce the number of probes inserted in a program we also reduce its object code size and the runtime overhead.

In Section 3.2, we saw that a basic block in a super block is covered *iff* all basic blocks in that super block are covered. Thus, it is sufficient to place one probe per super block instead of placing one probe per basic block. It is, however, not necessary to probe all super blocks. A super block, U, need not be probed if it satisfies the following condition: A test case that covers U must also cover one of its children in the super block dominator

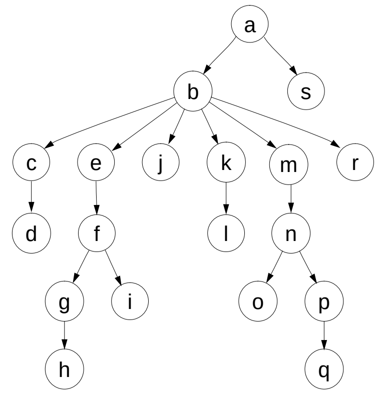
Figure 12: Edge predominator tree

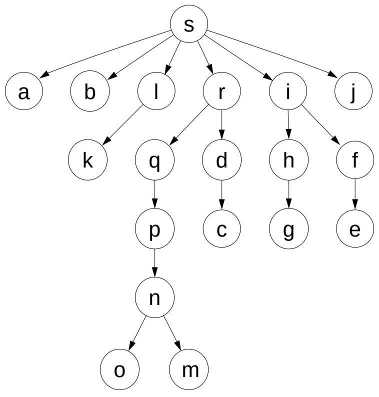
Figure 13: Edge postdominator tree

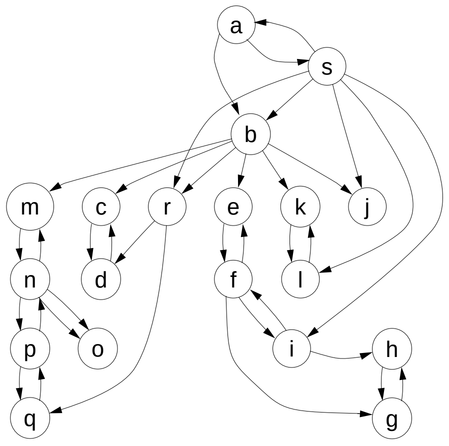
Figure 14: Edge dominator graph

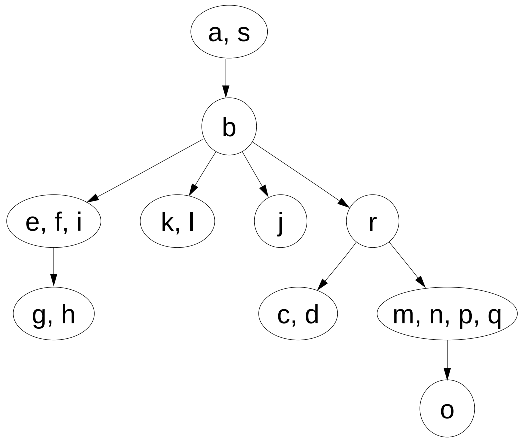
Figure 15: Edge partition dominator graph

graph, i.e.:

$$covered(U,t) \implies \exists V \ni U \longrightarrow V, covered(V,t) \quad (6)$$

We need not probe U above because its coverage may be inferred from that of V. Super blocks that do not satisfy the above condition, however, must be probed. As a leaf in a super block dominator graph has no children, it may never satisfy the above condition. Thus it must be probed. An internal node, on the other hand, may or may not satisfy the above condition. For example, super block B in Figure 7 satisfies the above condition as it is impossible to cover it without also covering one of its two children—super blocks C and H. Thus super block B need not be probed. Super block E, on the other hand, must be probed as it is possible to cover it without covering any of its children—we may have a test case that covers basic blocks 5 and 6 but not 7[6]. The flowgraph in Figure 2 shows that that this is indeed possible.

[6]Note that although it is desirable to develop test cases that cover one or more leaves in the super block dominator graph, the tester may supply test cases that do not cover any. They should, nevertheless, be able to find the true overall coverage irrespective of whether or not the test cases cover any leaves in the dominator graph.

As in the case of a leaf node, an internal node that has just one child node may never satisfy Condition (6). If it did then every time it is covered its unique child node would be covered as well. This would imply that the child node dominates the parent node. As a parent node always dominates a child node, it would mean both the parent and the child node are in the same super block—a contradiction. Thus all nodes that have just one child node must be probed.

Most nodes with two or more children do satisfy Condition (6) and thus they need not be probed. There may, however, be exceptions. Fortunately, they are relatively few in number (see Section 7) and we can identify them. Figure 11 includes a simple algorithm to check if a node in a super block dominator graph should be probed. It takes $O(N+E)$ time where N and E denote the number of nodes and edges in the graph.

The super block dominator graph in Figure 7 does not have any nodes with two or more children that need to be probed. It does, however, have two internal nodes

program	basic blocks	super-block leaves		block probes required		edge-partition leaves		edge probes required	
sort	455	138	30%	152	33%	195	31%	197	32%
spiff	1266	361	29%	404	32%	458	29%	462	29%
mgr	3848	1043	27%	1233	32%	1579	31%	1602	31%
ion	4886	1280	26%	1507	31%	1697	27%	1740	28%
atac	8737	2574	29%	2971	34%	3553	30%	3645	31%
odin	9870	2344	24%	2944	30%	3575	29%	3623	30%
xlib	15580	5111	33%	6016	39%	7435	36%	7525	36%
tvo	17680	6267	35%	8150	46%	11783	45%	11796	45%

Table 1: Experimental results

with one child node each that must be probed—super blocks A and E. Thus, in this example, we only need to probe six basic blocks—1, 4, 5, 7, 9, and 10—one each from the four leaves and the two internal nodes mentioned above.

Whenever a probe is reached during the program execution, all basic blocks in the corresponding super block and all its ancestors in the super block dominator graph are marked as covered. The coverage achieved at any given time, then, may be computed by counting the number of basic blocks marked as covered at that time. Consider, again, our example flowgraph in Figure 2. Suppose the very first test case developed causes the path <1, 2, 14> to be executed[7]. This causes the probe at basic block 1 to be marked as covered, which, in turn, causes basic blocks 2 and 14 to be marked as covered as they belong to the same super block as basic block 1. Thus coverage at this time may be correctly computed as 3/14, or 21%. Next, suppose another test case causes the execution path <1, 2, 3, 8, 10, 11, 12, 13, 2, 14> to be executed. As the probe at basic block 1 was already marked covered by the previous test case, this test case causes the probe at basic block 10 to be marked as covered. This, in turn, causes five new basic blocks to be marked as covered—3, 8, 11, 12, and 13—raising the count of covered basic blocks to 9. Therefore, the coverage at this time is again computed correctly as 9/14, or 64%.

6 Expediting Branch Coverage

The techniques described so far may also be used to expedite branch coverage—we only need to find pre- and postdominator relationships among edges instead of nodes. Figures 12 and 13 show the edge pre- and postdominator trees, respectively, of the flowgraph in Figure 2. Figure 14 shows the edge dominator graph obtained by merging the two trees. As in the case of a basic block dominator graph, we may also find the strongly connected components of an edge dominator graph. We refer to these components as *edge partitions*. An edge in an edge partition is covered by a test case *iff* all edges in that partition are covered by that test case. Figure 15 shows the edge partition dominator graph obtained by merging the strongly connected components of the edge dominator graph in Figure 14 and removing the composite edges from the resulting graph. In this example, the edge partition dominator graph is a tree but, in general, it may be a directed acyclic graph. Note that a tester only needs to develop test cases targeted to cover five edges, one each from the five leaves in Figure 15. The remaining fourteen edges are automatically covered by the same test cases.

As in the case of block coverage, space and time overhead of measuring branch coverage may also be reduced by placing one probe per leaf and certain internal nodes where covering the corresponding edge partition does not imply covering at least one of its children.

[7]Note that this test case does not exercise any leaves in the super block dominator graph. A tester should be able to find out the correct coverage achieved at any time regardless of whether or not any leaves have been covered.

7 Experimental Results

To evaluate the effectiveness of the techniques proposed here, we incorporated them into a prototype tool, Spyder [1], that provides dynamic program slicing facilities for C programs. We used it to find the number of leaves and internal nodes in the super block dominator graph and the edge partition dominator graph that need to be probed for eight diverse systems and library packages whose source codes were locally available. These systems varied in size from about 1,000 to 75,000 lines of C code. Table 1 lists the results. Note that the number of leaves in a super block dominator graph, on the average, is about 29% of the number of basic blocks. This means, in order to achieve 100% block coverage, testers testing these systems only need to develop test cases targeted to cover, on the average, 29% of the basic blocks. In case an automatic test case generator is used, it only needs to generate test cases to cover 29% of the basic blocks. Also note that the number of probes required is, on the average, 35% of the number of basic blocks. Thus, only 6% internal nodes need to be probed in addition to the leaf nodes.

Similarly, the number of leaves in an edge partition dominator graph, on the average, is about 32% of the number of edges. This means, in order to achieve 100% branch coverage, testers testing these systems only need to develop test cases targeted to cover, on the average, 32% of the branches. Also, the number of edge probes required is, on the average, 33% of the number of edges. Thus only 1% additional branches, besides the leaves in the edge partition dominator graph, need to be proved.

Figure 16 shows the analogue of Figure 10 for the sort program. Note that the first one third of the 30% targeted blocks provide more than 75% coverage. Recall that the curve in this graph gives us a lower bound on the cumulative coverages achievable. The actual curve, represented by the dotted curve, depends on the particular test cases developed but it always lies above the solid curve. Thus the actual savings in the number of test cases that need to be developed are always higher than that implied by the solid curve.

As coverage testing is generally performed during the unit testing phase, the numbers presented above are computed assuming that each program unit is tested in isolation. In other words, they are computed based on intraprocedural control flow analysis. But oftentimes, multiple units are tested together even during unit testing. In these situations, we may further increase the savings by performing interprocedural analysis, as covering a block in one unit may imply covering several blocks in other units as well.

8 Related Work

We are not aware of any reference in the literature that addresses the problem discussed here although several references are available on the related problem of optimal program profiling [4, 7, 11, 15, 17]. Program profiling involves determining the *frequency counts* of flowgraph nodes and edges when it is executed on a given test set. Clearly, if we have the frequency counts of all nodes and edges we know which of them are covered and which not.

Optimal profiling techniques are aimed at finding small subsets of nodes/edges in a flowgraph such that if the frequency counts of the nodes/edges in the subset are known, the frequency counts of other nodes/edges may be inferred from them. Note, however, that covering the nodes/edges in a subset identified by these tech-

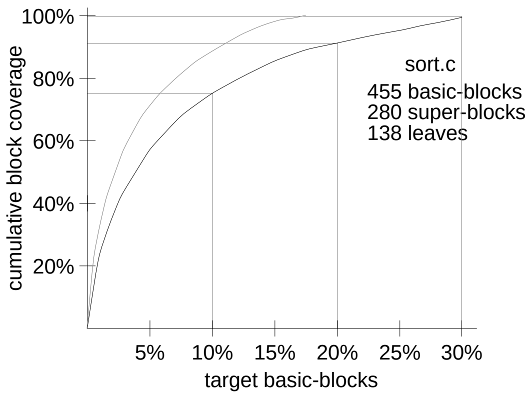

Figure 16: Coverage rate for sort

niques does *not* imply covering all other nodes/edges. It simply means we can find out whether or not other nodes/edges are covered if know the frequency counts of those in the subset. Thus we may not use these techniques to reduce the time spent developing test cases.

Also note that with optimal profiling techniques, simply knowing whether or not nodes/edges in the subset are covered is not sufficient to infer whether or not others are covered—we must capture how many times the nodes/edges in the subset are executed. With the techniques presented in this paper, on the other hand, simply knowing whether or not the nodes/edges in the subset are covered is enough to find out whether or not others are covered. Therefore, with our techniques, a probe may be removed after the first time it is reached. It need not be executed every time the control reaches the corresponding program location.

We know of only one other reference, [8], that also addresses the selective coverage problem discussed in Section 3.3 although in the context of automatic test case generation. It uses a generalization of the postdominator relationship between nodes to that between sets of nodes to guide the generation of test cases that cover a given set of nodes. A sequence $V_1, V_2, \ldots, V_n$ of sets of nodes is determined such that V_{i+1} postdominates V_i, $1 \leq i \leq n-1$, and each V_i contains at least one node from the relevant set of nodes to be covered. First a test case that covers one or more relevant nodes in V_1 is generated. Then, if possible, it is extended so it also covers one or more relevant nodes in V_2; and so on. The process is continued until all relevant nodes have been covered.

The above approach is complimentary to the approach taken in this paper. The latter may be used first to find a subset of the relevant nodes such that covering the subset would imply covering all the relevant nodes. Then, the above approach may be used to find as few test cases as possible to cover this subset. For instance, in the case of the example in Section 3.3, instead of applying the above approach to the relevant set of five nodes, $\{3, 6, 10, 11, 13\}$, we only need to apply it over a relevant set of two nodes, $\{6, 10\}$.

9 Summary

In this paper we have presented some techniques that may help expedite both block and branch testing of programs. For large programs, the number of basic blocks and branches that must be covered may be overwhelming. The techniques presented here help us identify a significantly smaller subset of relatively "high efficiency" blocks and branches such that covering them implies that other blocks and branches are automatically covered. The same techniques may also be used to reduce the number of probes placed in a program in order to find out which blocks and branches have been covered. This helps reduce the object code size and the runtime overhead imposed by coverage testing tools. Preliminary experiments have shown that these techniques may indeed be very effective in saving both the user and the system time spent during coverage testing of programs. More experimentation is required to evaluate how well testers are able to exploit these techniques in reducing the number of test cases they develop. Also, it would be interesting to determine how well these techniques perform compared to a heuristic approach, such as covering the most deeply nested blocks first.

Acknowledgements

We would like to thank Bob Horgan and Norman Ramsey for their helpful comments on an earlier draft of this paper. We would also like to thank the anonymous referees for their useful suggestions on improving its presentation.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 59–70. ACM Press, Jan. 1992.

[5] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings Publishing Co., 1987.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Giude to the Theory of NP-Completeness*. Freeman, 1979.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13:671–685, 1983.

[8] R. Gupta. Generalized dominators and postdominators. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 246–257. ACM Press, Jan. 1992.

[9] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[10] J. R. Horgan and S. L. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium on Assessment of Quality Software Development Tools*, pages 2–10. IEEE Computer Society Press, May 1992.

[11] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[12] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[13] D. M. Moyles and G. L. Thompson. An algorithm for finding a minium equivalent graph of a digraph. *Journal of the ACM*, 16(3):455–460, July 1969.

[14] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[15] R. L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, SE-8(1):34–42, Jan. 1982.

[16] P. W. Purdom, Jr. and E. F. Moore. Immediate predominators in directed graphs. *Communications of the ACM*, 15(8):777–778, Aug. 1972.

[17] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, SE-1(4):403–411, Dec. 1975.