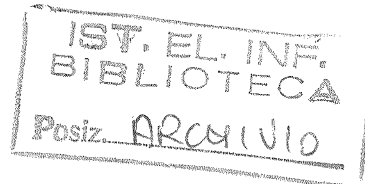
Consiglio Nazionale delle Ricerche



ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

How many paths are needed for branch testing?

Antonia Bertolino and Martina Marré

Nota Interna B4 - 41
November 1994

How many paths are needed for branch testing?

Antonia Bertolino

Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. *

Martina Marré

Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.†

Abstract

The number of test cases needed to achieve branch coverage is important to evaluate the effort needed to test a given program. However, the bounds proposed so far in the literature are not effective to measure testing effort. In this paper, we introduce a new, meaningful lower bound on the number

*Corresponding author's address: Antonia Bertolino, IEI-CNR, via S. Maria, 46, 56126 Pisa, Italy. Phone: +39 50 593478 E-mail: bertolino@iei.pi.cnr.it

†Research performed while this author was at Dipartimento di Informatica, Università di Pisa, Italy, and at North Carolina State University, USA.

of test cases needed to achieve branch coverage. We first identify the set of unconstrained arcs in a ddgraph. This is the minimum set of arcs such that a set of paths that exercises these arcs covers all the branches in the program. In general, a path may cover more than one unconstrained arc: the strategy we use to combine more unconstrained arcs into one path determines the cardinality of the set of test paths thus obtained, i.e., the bound we are looking for. It is now commonly accepted that the real problem in branch testing is to derive an executable set of test paths. Therefore, we will consider those control flow paths containing a low number of decisions as being meaningful, since they are more likely to be feasible. We formalize this notion by introducing the weak incomparability relation between ddgraph arcs. We then define the new, meaningful bound as the maximum number of unconstrained arcs in a ddgraph that are mutually weakly incomparable. Furthermore, we discuss interesting properties of this new bound. We analyze the bound with respect to Weyuker's axioms for complexity measures (Weyuker, 1988), and we show that the bound fits into the testability model of Bache and Müllerburg (Bache and Müllerburg, 1990).

1 Introduction

Testing a software system consists of executing it over a suitable sample of input data and then checking if the output produced matches what was expected. Testing is widely used to enhance software quality (Deutsch and Willis, 1988), and, specifically, to uncover bugs that are inevitably introduced during the software development process.

Testing activities consume a considerable fraction of the time and resources spent to produce a software product (Beizer, 1990), and therefore it would be useful to have a way to estimate this *testing effort*. Knowing in advance how much effort will be needed to test a given program is essential to the manager to plan the software process. Ultimately, this would allow to predict a measure, expressed in dollars, for example, which quantifies the overall cost of the testing phase. Of course, this is very hard to achieve, due to the multiplicity of factors involved in the testing process, many of which, e.g tester's expertise, or even tester's commitment, are not objectively measurable. What can be done instead is to derive and to refer to a range of measures that capture significant, quantifiable attributes (Fenton, 1991) of the testing process, such as the number of tests to be performed, or the number of errors expected per line of code. The *number of tests* to be executed is an important and useful attribute of the entity testing effort.

In order to select a suitable and finite subset of tests from the potentially infinite execution domain, different strategies can be followed, based on program specifica-

tion or on program structure. Whatever strategy is selected, measures of structural coverage (Miller, 1984; Rapps and Weyuker, 1985) can be used to predict the number of test cases needed to guarantee a particular coverage, and to determine how thorough the executed test cases have been. Therefore, the number of tests to be executed on a given program can be evaluated by measuring the minimum number of test paths to be exercised to achieve a certain structural coverage.

In particular, branch coverage, i.e. exercising each edge in the program flowgraph at least once, is commonly accepted as a "minimum mandatory testing requirement" (Beizer, 1990). Thus, the minimum number of test paths needed to achieve branch coverage can be regarded in a sense as a measure of the *minimum mandatory effort* to test a given program, regardless of the particular strategy used in the selection of test data.

The computation of a lower bound on the number of test cases needed to achieve branch coverage has already been addressed in the testing literature. McCabe's number (McCabe, 1976) is often (mis)used as such a bound. This widely used approach is however criticizable (Beizer, 1990). In fact, McCabe's number properly provides the number of tests needed to satisfy his "structured testing" strategy (McCabe, 1982) and it is not related to any other structural testing strategy.

Ntafos and Hakimi (Ntafos and Hakimi, 1979) tackled this problem by taking a network-theory approach. By generalizing Dilworth's theorem (Dilworth, 1950) to an arbitrary flowgraph, they state that the number of paths in a minimum path cover is equal to the maximum number of mutually incomparable edges in the

flowgraph.

Ntafos (Ntafos, 1988) established also an upper bound on the number of tests required by several structural strategies: in particular for branch testing, this bound is a function of the number of segments in the program. However, the number of test cases needed in practice is usually considerably less than what is implied by this bound.

More recently, Bache and Müllerburg used the hierarchical metrics approach to solve this problem (Bache and Müllerburg, 1990). By using prime decomposition (Fenton and Whitty, 1986), they show how to calculate the *minimum* number of test paths for a family of control flow testing strategies, including branch testing. However, when the minimum number of test paths is used as a lower bound on the branch testing strategy, the presence of loops in a program reduces the number of test cases required. This goes against our intuition: the introduction of a loop would increase the control flow complexity and hence should require a higher testing effort. This is not the case if we use the minimum number of test paths. For example, if a program consists of a loop with a very complex body, the theoretical minimum number of paths needed to branch-cover it is always one. Hence, Bache and Müllerburg conclude that branch testing is "insufficient" (Bache and Müllerburg, 1990). While this unquestionably follows from the use of the minimum number of test cases as a measure of the testability of a program, branch testing is in practice among the more widely used strategies (Yates and Malevris, 1989). Hence, the use of this bound implies that one of the most used strategy is

insufficient.

In conclusion, the earlier development of measures for the number of test paths required to achieve branch coverage has not taken into account the expected use of this number in practice. Thus, a different, more useful approach should be taken to count the number of test cases needed for branch testing a program. We believe that a clear specification not only of what attribute is being measured, but also of *why* it is being measured must be stated prior to the definition of such a measure. Correspondingly, in this paper we introduce a new method to evaluate a new, *meaningful* lower bound on the number of test cases needed to achieve branch coverage.

We argue that the set of test paths that should be planned to satisfy the branch coverage criterion is not the mathematical minimum set, whose cardinality would be obtained as in (Bache and Müllerburg, 1990) or in (Ntafos and Hakimi, 1979), but it is a *meaningful* set. By meaningful we mean both corresponding to a presumed usage of the program and, most importantly, feasible. In fact, it is now commonly accepted that the real problem in branch testing is to derive an *executable* set of test paths (Hedley and Hennell, 1985): some recent work (Yates and Malevris, 1989) has given statistical evidence to the very intuitive notion that paths with a low number of predicates are more likely to be feasible. This agrees with the belief that it is better to test many simple paths than a few complicated ones (Beizer, 1990).

We introduce here the following notions. We use a particular flowgraph called *ddgraph* to represent a program's control flow structure. Given a ddgraph $G =$

(V, E), there exists a minimum set of arcs, the set $UE \subseteq E$ of *unconstrained arcs* (Bertolino, 1993), with the property that a set of paths which exercises them covers all the branches in the program. Hence, of course, the number of test cases needed to achieve branch coverage is $\leq |UE|$. However, for an arbitrary ddgraph, an entry-exit path may, in general, cover more than one unconstrained arc: the strategy by which we combine together more unconstrained arcs into one path affects both the properties and the cardinality of the set of test paths thus obtained. We can minimize the number of test paths by combining into each path the largest number of unconstrained arcs. However, this may not be meaningful from a practical point of view. Instead, we shall consider meaningful those paths containing a low number of decisions. In particular, such paths do not combine together unconstrained arcs requiring to enter different loops (obviously excluding the case of nested loops), and if a loop is entered, then it will be iterated just once. In our approach, we capture this intuitive notion of combining unconstrained arcs to form meaningful paths within a rigorous, mathematical framework by introducing the notion of *weak incomparability.* The latter, enlarging the notion of incomparability between arcs, used in (Ntafos and Hakimi, 1979), allows us to formally state the notion of meaningfulness of test paths introduced above.

In the next Section, we set some preliminary background. In Section 3 we introduce our bound by intuitively defining what we count and discussing its relationship with earlier work. In Section 4 we formally define the β_{branch} bound and outline a computation methodology. In Section 5 we evaluate the metric according

to axiomatic properties provided in the literature (Weyuker, 1988) and we show the relation between our approach and Bache and Müllerburg general method. Finally, we give conclusions and suggest future developments in Section 6.

2 Preliminary Background

In this section, some basic notions used through the paper are briefly introduced. More material on flowgraphs can be found in (Hecht, 1977).

2.1 Ddgraphs

A program structure is conveniently analyzed by means of a directed graph, called *flowgraph*, that gives a graphical representation of the program control flow. A *directed graph* or *digraph* $G = (V, E)$ consists of a set V of *nodes* or *vertices*, and a set E of *directed edges* or *arcs*, where an arc $e = (T(e), H(e))$ is an ordered pair of *adjacent* nodes, called *Tail* and *Head* of e, respectively. We say that e *leaves* $T(e)$ and *enters* $H(e)$. If $H(e) = T(e')$, e and e' are called *adjacent* arcs. For a node n in V, *indegree*(n) is the number of arcs entering and *outdegree*(n) the number of arcs leaving it. The digraph $G' = (V', E')$ is a *subgraph* of a digraph $G = (V, E)$ if $E' \subseteq E$ and $H(e), T(e) \in V' \subseteq V$ for each arc $e \in E$.

A *path* P of length q in a digraph G is a sequence $P = e_1, e_2, ..., e_q$; where $T(e_{i+1}) = H(e_i)$ for $i = 1, ..., q-1$. P is said to be a path from e_1 to e_q, or from $T(e_1)$ to $H(e_q)$. A *subpath* P' of P is a sequence $P' = e_{i_1}, ..., e_{i_r}$, where $\{i_1, ..., i_r\} \subseteq \{1, ..., q\}$. The path of length 0 (empty sequence of arcs) is the *empty*

path. An arc e *reaches* an arc e' (a node n *reaches* a node n') if there exists a path in G from e to e' (from n to n'). On the contrary, two arcs (two nodes) are *incomparable* if there does not exist a path in the ddgraph containing both of them, i.e., if each of these arcs (nodes) does not reach the other. A set S of arcs (nodes) is said an *incomparable arc (node) set* if any two distinct arcs (nodes) in S are incomparable.

A path P in a digraph G is *simple* if all its nodes are distinct. A path $P = e_1, e_2..., e_q$ is a *cycle* if $T(e_1) = H(e_q)$. A *simple cycle* is a cycle in which all nodes, except the first and the last, are distinct. An *acyclic* digraph is a digraph that has no cycles.

A program control flow may be mapped onto a flowgraph in different ways. In this paper, we use a flowgraph representation called *ddgraph* (for decision-to-decision graph), which is particularly suitable for the purposes of branch testing. In fact, each arc in a ddgraph directly corresponds to a program branch; thus, program branch coverage is immediately measured in terms of ddgraph arc coverage. The following is a formal definition of ddgraph.

Definition 1 *Ddgraph*

A ddgraph is a digraph $G = (V, E)$ with two distinguished arcs e_0 and e_k (the unique entry arc and the unique exit arc, respectively), such that any other arc in E is reached by e_0 and reaches e_k, and such that for each node $n \in V$, $n \neq T(e_0)$, $n \neq H(e_k)$, $(indegree(n) + outdegree(n)) > 2$, while $indegree(T(e_0)) = 0$ and $outdegree(T(e_0)) = 1$, $indegree(H(e_k)) = 1$ and $outdegree(H(e_k)) = 0$.

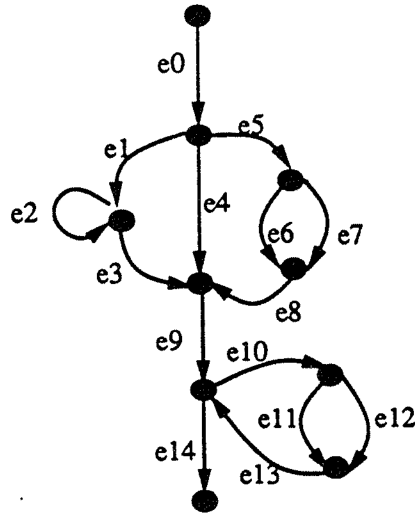


Figure 1: G_{ex}

In Figure 1 we present the ddgraph G_{ex} with distinguished arcs e_0 and e_{14}.

A ddgraph node can be associated to a *decision* (a program point at which the control flow diverges) or a *junction* (a program point at which the control flow merges). A ddgraph arc is associated with a strictly sequential set of program statements uninterrupted by either decisions or junctions, i.e., a sequence of program statements not containing alterations of the control flow. However, in some cases, an arc is introduced that does not correspond to a program segment, but nevertheless represents a possible course of the program control flow (e.g., the implicit ELSE part of an IF statement).

2.2 Unconstrained Arcs

We use the relations of dominance and implication (elsewhere called inverse dominance or post-dominance) between the arcs of a ddgraph. Intuitively, an arc e_i dominates another arc e_j if any path from the entry arc to e_j must pass through arc e_i.

Definition 2 *Dominance*

Let $G = (V, E)$ be a ddgraph with distinguished arcs e_0 and e_k. An arc e_i dominates an arc e_j if every path P from entry arc e_0 to e_j contains e_i.

By applying the dominance relation between the arcs of a ddgraph G, we obtain a tree (whose nodes represent the ddgraph arcs) rooted at e_0. This is called the *dominator tree $DT(G)$*. For each pair (e_i, e_j) of adjacent nodes in the dominator tree, $e_i = Parent(e_j)$ is the *immediate dominator* of e_j. The immediate dominator e_i of an arc e_j is a dominator of e_j with the property that any other dominator of e_j also dominates e_i. Note that each arc (different from e_0) has just one immediate dominator.

Next, we introduce the "dual" relation of *implication* between two arcs in a ddgraph. Intuitively, an arc e_i implies another arc e_j if each path from arc e_i to the exit arc must pass through arc e_j.

Definition 3 *Implication*

Let $G = (V, E)$ be a ddgraph with distinguished arcs e_0 and e_k. An arc e_i implies an arc e_j if every path P from e_i to exit arc e_k contains e_j.

By applying the implication relation between the arcs of a ddgraph G, we obtain a tree (whose nodes represent the ddgraph arcs) rooted at e_k. This is called the *implied tree* $IT(G)$. For each pair (e_j, e_i) of adjacent nodes in the implied tree, $e_j = Parent(e_i)$ is the arc *immediately implied* by e_i. An arc e_j is immediately implied by an arc e_i if e_i implies e_j and any other arc that is implied by e_i is also implied by e_j. Note that each arc (different from e_k) is immediately implied by just one arc.

Dominance and implication allow us to identify a subset of ddgraph arcs which is very useful for branch testing: the set of *unconstrained arcs* (Bertolino, 1993). The *fundamental property of unconstrained arcs* is that a path set that covers all the unconstrained arcs of a ddgraph also covers all the arcs in the ddgraph, and the unconstrained arcs form the minimum set of arcs with that property. This property is proved in (Bertolino, 1993). Let us introduce the definition of unconstrained arc.

Definition 4 *Unconstrained Arcs*

An arc e_u is unconstrained if e_u dominates no other arc and is implied by no other arc in G.

In other words, an arc e in a ddgraph G is unconstrained if for any other arc e' in G there is at least one path from e_0 to e_k containing e' and not containing e.

We can immediately find the set of unconstrained arcs by using the dominator tree and the implied tree of a ddgraph G (Bertolino, 1993). In fact, by definition, we can obtain the set $UE(G)$ of unconstrained arcs of G as $DTL(G) \cap ITL(G)$,

where $DTL(G)$ is the set of leaves of $DT(G)$ and $ITL(G)$ is the set of leaves of $IT(G)$.

Thus, for the ddgraph G_{ex} we have:

$DTL(G_{ex}) = \{e_2, e_3, e_4, e_6, e_7, e_8, e_{11}, e_{12}, e_{13}, e_{14}\}$,

$ITL(G_{ex}) = \{e_0, e_1, e_2, e_4, e_5, e_6, e_7, e_{10}, e_{11}, e_{12}\}$,

and then the set of unconstrained arcs of G_{ex} is:

$$UE(G_{ex}) = DTL(G_{ex}) \cap ITL(G_{ex}) = \{e_2, e_4, e_6, e_7, e_{11}, e_{12}\}.$$

2.3 Path Covers

Branch testing, which requires to exercise every program branch at least once, is commonly accepted as the "minimum mandatory testing requirement" (Beizer, 1990). A set of paths such that each branch in the program is covered by at least one path in the set is called a *path cover.*

Definition 5 *Path Cover*

Let $G = (V, E)$ be a ddgraph. A set of paths $\wp = \{P_1, ..., P_n\}$ is a path cover for G if for each arc $e \in E$ there exists at least one path in $\wp$ containing e.

In particular, a path cover $\wp$ for a ddgraph G is called a *minimum path cover* if there is no path cover $\wp'$ for G with[1] $|\wp'| < |\wp|$. A path cover $\wp = \{P_1, ..., P_n\}$ is called a *simple path cover* if for $i \in \{1, .., n\}$, P_i is a simple path.

[1]The cardinality of a set S, denoted by $|S|$, gives the number of elements in S.

By definition, a path cover satisfies the branch testing criterion (of course, provided that the paths in the path cover are executable). And, by the fundamental property of unconstrained arcs, a set of paths which covers the unconstrained arcs of a ddgraph is a path cover.

3 The Problem and Related Work

How many test cases should be planned to achieve branch coverage in practical software development? This question, addressed in this paper, is an important one, because managers and testers need such a bound to plan their work. In fact, very often are measures of coverage used to evaluate testing thoroughness, and therefore the number of test cases needed to achieve branch coverage is useful to estimate the testing effort needed, as an important part of the development effort.

In this section, we first discuss earlier work in the evaluation of a lower bound on the number of test cases needed for branch coverage. After showing why proposed measures are not useful from a practitioner's viewpoint, we introduce our new, meaningful bound at an intuitive level.

Several years ago, McCabe (McCabe, 1976) proposed the use of the cyclomatic number $v(G)$ of the program flowgraph $G = (V, E)$, with

$$v(G) = |E| - |V| + 2,$$

not only to measure the structural complexity of the program considered, but also as the basis for a testing methodology. Indeed, $v(G)$ gives the number of maximal linearly independent paths in a flowgraph G, i.e., every path in G can be obtained by a linear combination of $v(G)$ independent paths in a basis set. Specifically, McCabe's number gives a lower bound on the number of test cases needed for his *structured testing* strategy (McCabe, 1982). However, this number is often (mis)used (Beizer, 1990) as a lower bound on the number of test cases needed for branch-testing a program.

This is not exactly the number we are looking for. Consider for example a simple program (Figure 2A) consisting of a number r of cascading *ifs*. Through direct inspection, we can see that a total of two test cases are needed to achieve branch coverage, but $v(G)$ is $r+1$. In his paper (McCabe, 1976), McCabe considered two cascading *ifs* and explained why the cyclomatic number (yielding 3) and the "actual complexity" [2] (yielding 2) differ. This happens because the program we actually test by exercising only two paths can be represented by a less complex flowgraph. This flowgraph is obtained by removing one decision, for example as in Figure 2B. However, this argument is not sufficient to convince us that $r + 1$ test cases should be planned in practice to branch-test the program in Figure 2A. Branch testing should cover all of the possible branches in the program, and this can be done by planning just two test cases.

[2]McCabe defines *actual complexity* as the number of paths tested to achieve branch coverage.

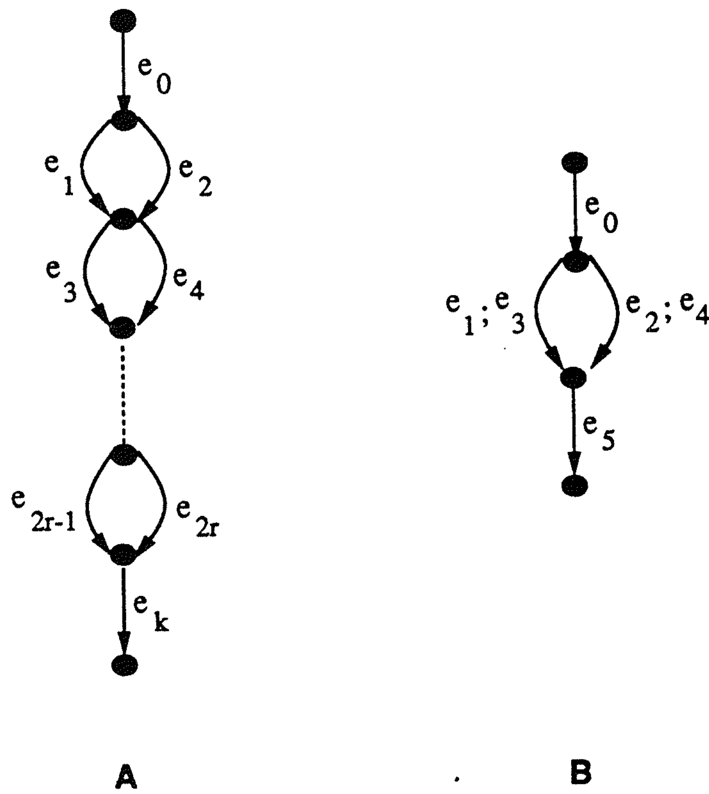


Figure 2: Number of test cases: Cascading ifs

Some years later, the problem of finding a path cover with the minimum number of paths has been addressed by Ntafos and Hakimi (Ntafos and Hakimi, 1979), using a network-theory approach. In their paper, they generalized Dilworth's theorem (Dilworth, 1950) for acyclic digraphs to arbitrary digraphs. That theorem states that the minimum number of paths in a path cover for a digraph G is equal to the cardinality of the largest incomparable node set of G. They also give two methods for finding this number. This number provides a sound mathematical minimum bound. However, we argue that this is not the *real* bound on branch coverage. This problem is further discussed after introducing a more recent approach, that coincides with Ntafos and Hakimi with respect to branch coverage.

A unified theory has been introduced by Bache and Müllerburg (Bache and Müllerburg, 1990) to measure *testability*, the number of test cases needed not only

for branch testing, but for a wide family of control-flow based test strategies. They applied the results of Fenton-Whitty theory of program decomposition (Fenton and Whitty, 1986). The building blocks of the well-known Fenton-Whitty approach to control flow analysis are the *prime* flowgraphs, e.g. the IF-THEN-ELSE prime, the WHILE-DO prime, the 2-EXIT-LOOP prime, etc... Any flowgraph that is not a prime can always be created from the repeated sequencing and nesting of primes. Conversely, every flowgraph has a unique decomposition into a hierarchy of primes, which describes how the flowgraph is built by sequencing and nesting primes. Consequently, we can construct "hierarchical measures" by simply assigning a value $m(G)$ to each prime and then defining how to calculate $m(G)$ for the sequencing and nesting functions. Using the prime decomposition, the resulting function m then extends to a function over all flowgraphs.

By using prime decomposition, Bache and Müllerburg compute the minimum number of test cases required to satisfy various structural testing strategies (McCabe's structured testing and branch testing strategies, among others). Thus, they define the number of test cases needed to satisfy each test strategy considered on each of the primes in a basic set. Then, applying the decomposition property of flowgraphs, they define how to calculate testability for n flowgraphs in sequence and for flowgraphs nested onto each of the basic primes. Let us observe that the number they thus calculate to measure the testability for the branch testing strategy is exactly equal to the bound set by Ntafos and Hakimi in their generalization of Dilworth's theorem. This is so since their definition of testability for branch

testing strategy for each prime considered is consistent with Ntafos and Hakimi' notion of incomparability.

For example, let us consider again the program in Figure 2A, obtained by sequencing r IF-THEN-ELSE primes. For branch coverage, each IF-THEN-ELSE prime requires two test cases, and the sequencing function for r flowgraphs $F_1, ..., F_r$ holds the maximum value of testability among the flowgraphs in sequence. Hence, the number of test cases needed would be two, as one would intuitively think.

However, even though this measure is correct from a theoretical point of view, we believe that problems arise in practice if we want to branch-test programs with loops. To see why, it suffices to consider as an example their measure of testability for a flowgraph obtained by nesting a flowgraph of arbitrary complexity onto a WHILE-DO prime. The testability always gives one, i.e. only one test case is required for branch coverage. More generally, when a set of tests of minimum cardinality is used to measure program testability, the presence of loops will reduce the number of required test cases, without reflecting the number of cases needed in practice.

To summarize, we have examined existing branch coverage measures, and we have concluded that they do not fulfill their purposes, mainly because they represent a mathematical measure without an associated "testing" meaning, i.e., they have not been developed taking into account the use of this number in practice. The real problem is to determine the smallest number of test cases we must plan in order to measure testing effort.

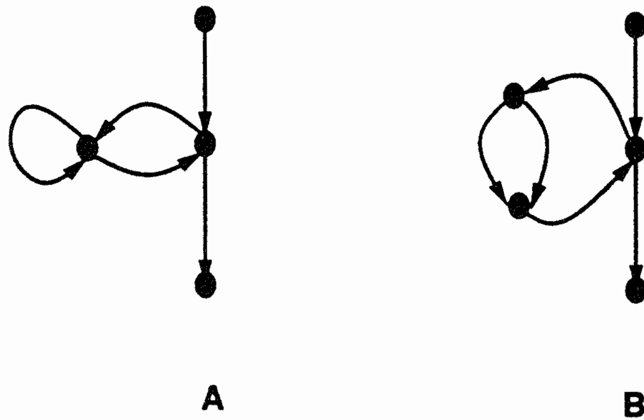


Figure 3: Number of test cases: Simple programs with loops

To introduce our approach, let us now consider the two simple cases of a WHILE-DO prime and an IF-THEN-ELSE prime both nested onto a WHILE-DO prime (Figures 3A and 3B). How many test cases should we plan for them? McCabe would say three for both cases, and Bache and Müllerburg (with Ntafos and Hakimi) would say one for both cases.

We say that planning one test case is sufficient for the flowgraph in Figure 3A. In fact, a test path which exercises the inner WHILE-DO loop would cover any other arc (note that this is the only unconstrained arc for this ddgraph). And, conversely, it must be necessarily exercised at least once to achieve total branch coverage. So, in this example, we agree with the theoretical minimum number. Instead, for the flowgraph in Figure 3B, we say that two test cases should be realistically planned: one covering the THEN part and the other covering the ELSE part of the nested

IF-THEN-ELSE statement. It is true that only one path would suffice in theory, but this in practice would require finding a test input that forces the control-flow execution to enter the WHILE-DO loop at least twice, passing once by the THEN part and once by the ELSE part. This path may be feasible, but possibly it may be not (i.e., no input data might exit to execute this path). More pragmatically, we would say that two sets of test input should be found, one to exercise the THEN part and another one to exercise the ELSE part. And, deriving the test inputs for these two simple paths, or determining if either of them is unfeasible, is certainly easier than for a complex path iterating the cycle.

In the rest of this paper, we shall present a measure of the testing effort based on meaningful properties that a set of test paths that achieves branch coverage should satisfy. Let us first describe our approach intuitively: given a ddgraph $G = (V, E)$, there exists a minimum set of arcs, the set UE of unconstrained arcs, which guarantees branch coverage. Then, of course, the number of test cases needed to achieve branch coverage is $\leq |UE|$. However, for an arbitrary ddgraph, an entry-exit path may, in general, cover more than one unconstrained arc: the strategy by which we combine together more unconstrained arcs into one path affects both the properties and the cardinality of the set of test paths thus obtained. Trying to combine into each path the more unconstrained arcs would minimize the number of test paths. However, this may not be meaningful from a practical point of view, as Figure 3B illustrates.

The intuitive notion of combining unconstrained arcs to form meaningful paths is captured within a rigorous, mathematical framework by introducing the notion of *weak incomparability*. The latter, enlarging the notion of incomparability between arcs (used by Ntafos and Hakimi), allows us to formally state the notion of meaningfulness of test paths introduced above: we shall consider meaningful those paths containing a low number of decisions. For example, a meaningful path does not combine together unconstrained arcs belonging to two cycles in sequence, nor unconstrained arcs belonging to a same cycle but that can be covered by a same path only by entering the cycle at least twice.

Obviously, for this bound to be meaningful, we should know how to calculate it. A method that exploits Ntafos and Hakimi results is introduced in the next section. Moreover, in Section 5, we also relate our bound to Bache and Müllerburg' theory and we show how we can apply their method to measure testability according to our "meaningful" branch coverage strategy.

4 The β_{branch} Bound

4.1 Definition of the β_{branch} Bound

In this section we formalize the notions introduced above. We are looking for the cardinality of a set of paths that is useful to guarantee branch coverage. It must be a meaningful set of minimal cardinality. For this purpose, we look for a set of paths that covers the set of unconstrained arcs; in fact, the fundamental

property of unconstrained arcs guarantees that this set of paths covers every branch in the ddgraph. More precisely, we search a set of paths $\wp$ that groups the set of unconstrained arcs in a meaningful way; i.e., we assume that a path in $\wp$ may cover more than one unconstrained arc, only if some precise conditions are met. In particular, if a path $P \in \wp$ covers an unconstrained arc e belonging to a cycle, the path P cannot enter the cycle more than once, and it cannot enter another cycle in the ddgraph, with the obvious exception of nested cycles.

Now we shall define when two arcs in G cannot be covered by the same path. In this case we shall call them two *weakly incomparable* arcs. Intuitively, two arcs are weakly incomparable if:

1. each of them can be covered by a simple path, but they both cannot be covered by the same simple path, or
2. one of them can be covered by a simple path and the other cannot be covered by a simple path, or
3. they belong to the same cycle in G and one reaches the other only by entering the cycle at least twice, or
4. they belong to different (not nested one within the other) cycles in G.

For example, in the ddgraph G_{ex} of Figure 1:

- arcs e_1 and e_4 are (weakly) incomparable (according to point 1);
- arcs e_1 and e_2 are weakly incomparable (according to point 2);

- arcs e_{11} and e_{12} are weakly incomparable (according to point 3), but are not incomparable: one reaches the other entering the cycle twice;
- arcs e_{11} and e_{13} are not weakly incomparable, since they belong to the same cycle but e_{11} reaches e_{13} entering the cycle just once;
- arcs e_2 and e_{11} are weakly incomparable (according to point 4);

We now define formally this relation with reference to a subset of E, since we shall use it later on the set of unconstrained arcs to define our β_{branch} bound.

Definition 6 *Weakly Incomparable*

Let $G = (V, E)$ be a ddgraph. Let $e, e' \in A \subseteq E$ and $e \neq e'$. Arcs e and e' will be called two weakly incomparable arcs in A if:

- *there does not exist a simple path from e_0 to e_k in G containing both e and e', and*
- *for any (not simple) path which contains both e and e', it is always possible to derive a subpath containing only one of them that is a path in G from e_0 to e_k.*

Note that if G is an acyclic ddgraph, $e, e' \in A \subseteq E$ and $e \neq e'$, then e and e' are weakly incomparable if and only if they are incomparable.

Now, let us analyze in general how many paths we need to cover all the arcs in a subset $A \subseteq E$. We are interested in the case $A = UE$. Let us suppose that we have identified a largest set of weakly incomparable arcs L in A, i.e., any two arcs

in L are weakly incomparable, and every arc in $A - L$ is not weakly incomparable with some arc in L. Then, we can construct a set $\wp$ of $|L|$ paths, such that each arc in L is covered by a different path in $\wp$. In particular, if $A = UE$, this set of paths is a meaningful path cover. A formal definition of a largest weakly incomparable arc set follows:

Definition 7 *Largest Weakly Incomparable Arc Set*

Let $G = (V, E)$ be a ddgraph, $A \subseteq E$. $LWI(A)$ is a largest weakly incomparable arc set for A in G if it is a subset of A satisfying:

- *for each $e, e' \in LWI(A)$, $e \neq e'$, then e and e' are weakly incomparable,*
- *$|LWI(A)| = max\{|E'|: E' \subseteq A$ and for each $e, e' \in E'$, if $e \neq e'$ then e and e' are weakly incomparable$\}$.*

Notice that given a ddgraph and a non empty subset of arcs A, there is at least one largest weakly incomparable set of arcs for A, and possibly it is not unique. For example, $LWI(E)_1 = \{e_1, e_2, e_4, e_6, e_7, e_{11}, e_{12}\}$ and $LWI(E)_2 = \{e_2, e_3, e_4, e_6, e_7, e_{11}, e_{12}\}$ are largest weakly incomparable arc sets for E in the ddgraph G_{ex} of Figure 1. $LWI(UE) = \{e_2, e_4, e_6, e_7, e_{11}, e_{12}\}$ is the only largest weakly incomparable set for UE in G_{ex}.

Now, bringing together the notions of unconstrained arcs and of weakly incomparable arcs, we introduce a formal definition of the β_{branch} bound, as the maximum number of unconstrained arcs that are mutually weakly incomparable. In

other words, this bound considers a different path in a path cover for each arc in $|LWI(UE)|$.

Definition 8 β_{branch}

Let $G = (V, E)$ be a ddgraph, UE the set of unconstrained arcs of G. Then β_{branch} is defined as the cardinality of a largest weakly incomparable arc set $|LWI(UE)|$ for UE in G.

For example, $\beta_{branch}(G_{ex}) = |LWI(UE)| = |\{e_2, e_4, e_6, e_7, e_{11}, e_{12}\}| = 6$.

4.2 Computation of the β_{branch} Bound

Now we outline a method to calculate the value of β_{branch} for a given ddgraph G. For this purpose, we observe the two following facts.

1. Suppose that there exists a simple path cover for G: $\wp = \{P_1, ..., P_n\}$. Then, the minimum number of paths needed to cover all the arcs in UE (and thus all arcs in G) can be calculated by solving an associated minimum flow problem (MIN-FLOW)[3]. In particular, suppose that we associate a minimum capacity constraint of value one to each arc in the set UE, and a minimum capacity constraint of value zero to the arcs in $E - UE$. Then, the value of a solution to MIN-FLOW with those constraints will be the number of arcs in a largest weakly incomparable arc set for UE.

[3]We can solve a MIN-FLOW problem in $O(|V||E|)$ time using a simple modification of the algorithms used to solve the maximum flow problem (Ford and Fulkerson, 1962)

2. Now, suppose that a simple path cover for G does not exist. Then, there is a subset of arcs that can only be covered by visiting a cycle in G (i.e., any path from e_0 to e_k containing an arc e in this subset is a not simple path). We can decompose the ddgraph G into a connected digraph G^a that contains only those arcs that can be covered by simple paths, and a set of maximal connected subgraphs $G_1, ..., G_r$ of G that do not contain arcs in G^a.

These two facts allow us to design a procedure to calculate the value of β_{branch}, by decomposing a ddgraph G into a "weakly simple" subgraph (i.e., a subgraph having a simple path cover) and a set of cyclic components. By visiting the ddgraph we can identify the arcs that cannot be covered by a simple path. Then, we consider the subgraph G^a of G containing only those arcs that can be covered by simple paths. We derive the set $UE \cap E^a$, i.e. the subset of the unconstrained arcs of G which are arcs in G^a. We can calculate the maximum number of weakly incomparable arcs in $UE \cap E^a$, by solving an associate MIN-FLOW problem. On the other hand, we derive the maximal cyclic components $G_1, ..., G_r$ of G that are not contained in G^a. Analogously, for each of them we derive the set $UE \cap E_i$, i.e., the subset of the unconstrained arcs of G which are arcs in G_i. We can calculate the maximum number of weakly incomparable arcs for the set $UE \cap E_i$ for each G_i, by recursively applying the same procedure. Finally, we just sum the values calculated for every component of G and obtain the β_{branch} corresponding to the ddgraph G. Formally:

$$|LWI(UE)| = |LWI(UE \cap E^a)| + |LWI(UE \cap E_1)| + ... + |LWI(UE \cap E_r)|.$$

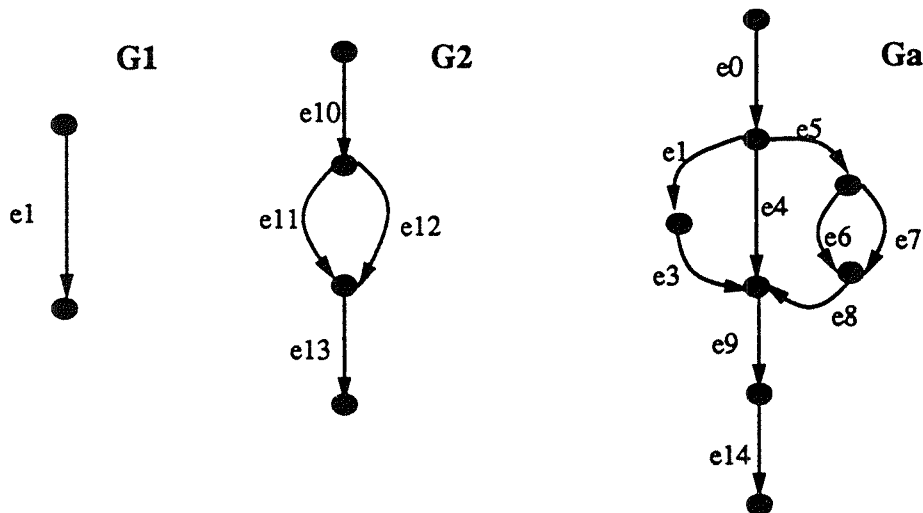


Figure 4: Decomposition of G_{ex}

As an example, let us consider the ddgraph G_{ex} and its decomposition presented in Figure 4, and $A = UE$. We have:

$UE \cap E_1 = \{e_2\}$, then $|LWI(UE \cap E_1)| = 1$;

$UE \cap E_2 = \{e_{11}, e_{12}\}$, then $|LWI(UE \cap E_2)| = 2$;

$UE \cap E^a = \{e_4, e_6, e_7\}$, then $|LWI(UE \cap E^a)| = 3$;

In fact, we already know that

$|LWI(UE)| = 6$.

5 Discussion

In this section we discuss the characteristics of the β_{branch} bound. We evaluate the new bound with respect to Weyuker's properties (Weyuker, 1988) and we analyze the relationship between the new bound and the testability numbers introduced by Bache and Müllerburg (Bache and Müllerburg, 1990).

First, we note that our bound can be considered, in some sense, as a measure of the complexity of a program's control-flow structure. Therefore, we evaluate the bound with respect to Weyuker's axioms (Weyuker, 1988), which are nine desirable properties that a general software complexity measure should satisfy. These axioms have been designed as a basis for the wide class of syntactic complexity measures. In our case, we consider one particular complexity measure: the complexity of the decision structure of the program. Therefore, some axioms do not apply to this particular case. Our bound, if regarded as a complexity metric, does not satisfy Axioms 2 and 9. According to citation (Weyuker, 1988):

Axiom 2: Let c be a nonnegative number. Then there are only finitely many programs of complexity c.

This property allows to distinguish between a program that performs very little computation and a program that performs massive amounts of computation, even though they have the same decision structure. However, in our case β_{branch} is related to branch coverage and then, it only depends on the decision structure of the program. For example, it is reasonable to associate $\beta_{branch} = 1$ with all

programs with sequential control flow.

Axiom 9 reads:

Axiom 9: There exist program segments X and Y such that the complexity of the code $X;Y$ is greater than the sum of the complexities of X and Y.

Again, we are analyzing the control flow complexity of the program. Then, it is reasonable to think that the control flow complexity of the program obtained by sequencing X and Y is not greater than the sum of the single control flow complexities of the segments X and Y.

However, β_{branch} satisfies the properties that a reasonable measure of the control flow complexity of a program should satisfy. Therefore, we conclude that our bound is a good metric of the complexity of a program's control flow structure.

Second, we observe that our approach can be considered as a new test coverage strategy, which we will refer to as "meaningful branch testing". Then, it should be possible to compute the testability for this strategy through Bache's and Müllerburg's method. To do this, we need to know how to compute β_{branch} for the primes considered as well as for sequencing and nesting. For each prime, we state that β_{branch} is given by the sum of two values: $c+l$. Informally, in the largest weakly incomparable set of unconstrained arcs whose cardinality is given by β_{branch}, c corresponds to the number of arcs in the set that are incomparable and l to the number of remaining arcs that are weakly incomparable, but not incomparable.

Thus, Table 1 defines c and l for the basis set of primes considered by Bache and Müllerburg. In the table, P_1 corresponds to the trivial flowgraph; D_0 to the

P_1	D_0	D_1	C_n	D_2	D_3	D_4	L_2
1,0	2,0	2,0	n,0	0,1	0,1	0,1	2,1

Table 1: c, l values for primes

	$F_1; ...; F_n$
c	$\max(c(F_1), ..., c(F_n))$
l	$l(F_1) + ... + l(F_n)$

Table 2: Sequencing function

IF-THEN prime; D_1 to the IF-THEN-ELSE prime; C_n to the CASE-OF-n prime; D_2 to the WHILE-DO prime; D_3 to the REPEAT-UNTIL prime; D_4 to the EXIT-FROM-MIDDLE prime and L_2 to the 2-EXIT-LOOP prime. Thus, for instance, for the IF-THEN-ELSE prime: $c = 2$ and $l = 0$, yielding $\beta_{branch} = 2 + 0 = 2$, and for the WHILE-DO prime: $c = 0$ and $l = 1$, yielding $\beta_{branch} = 0 + 1 = 1$.

Table 2 defines the sequencing function for flowgraphs F_1, .., F_n. The computation is performed separately for the values of c and l and then β_{branch} is derived by adding the two values together. Thus, for example, if an IF-THEN-ELSE prime and a WHILE-DO prime are combined in sequence, then $c = max(2,0) = 2$, $l = 0 + 1 = 1$, yielding $\beta_{branch} = 2 + 1 = 3$.

Table 3 defines the nesting function for each prime considered. For example, if we nest two flowgraphs F_1 and F_2 onto an IF-THEN-ELSE prime, the value of c is the sum of the values c_1 and c_2 for the two nested flowgraphs, and the value

	$D_0(F)$	$D_1(F_1, F_2)$	$C_n(F_1, ..., F_n)$
c	$c(F) + 1$	$c(F_1) + c(F_2)$	$c(F_1) + ... + c(F_n)$
l	$l(F)$	$l(F_1) + l(F_2)$	$l(F_1) + ... + l(F_n)$

	$D_2(F)$	$D_3(F)$	$D_4(F_1, F_2)$	$L_2(F_1, F_2)$
c	0	$c(F)$	$c(F_1)$	$c(F_1) + 1$
l	$l(F) + c(F)$	$l(F) + 1$	$l(F_1) + l(F_2) + c(F_2)$	$l(F_1) + l(F_2) + c(F_2)$

Table 3: Nesting function

of l is the sum of the values l_1 and l_2 for the two nested flowgraphs. If we nest a flowgraph F onto a WHILE-DO prime, the value of c is zero (which is equal to the value of c for the WHILE-DO prime) and the value of l is equal to the value of $c_1 + l_1$ for the nested flowgraph.

Thus, applying Bache's and Müllerburg's method, we can now recursively compute the β_{branch} bound from the decomposition tree of any flowgraph. [4].

[4]Note that the sequencing and nesting functions given in Table 2 and Table 3 respectively, do not apply to the trivial flowgraph. For it, Table 1 must be considered instead, i.e., the sequencing or nesting of P onto any flowgraph G does not change the value of c and l of the flowgraph G.

6 Conclusions and Further Work

In this paper we have analyzed the problem of establishing a lower bound on the number of test cases needed for branch testing. This is an important problem for various reasons. On one hand, measures of structural coverage are often used to evaluate testing thoroughness. On the other hand, branch coverage is commonly accepted as a minimum mandatory testing requirement. Then, a lower bound on branch testing can be useful to predict how much effort should be expected to be necessary for testing a given program.

We have shown that existing bounds on branch coverage are not adequate, since the use of this number in practice has been ignored during the development of the bounds. Then, by combining the notions of unconstrained arcs and of weakly incomparable arcs, we have introduced a new, meaningful lower bound on the number of test cases to be planned for branch coverage. The dominance and implication relations establish two partial orderings over the arcs of a ddgraph G, represented by the trees $DT(G)$ and $IT(G)$, respectively. These relations allow us to identify the unconstrained arcs, that is, the minimum subset of the set of ddgraph arcs such that a set of paths covering the arcs is a path cover for the ddgraph. Moreover, the notion of weak incomparability captures the intuitive idea of combining unconstrained arcs to create meaningful paths.

Recent work (Yates and Malevris, 1989) has given statistical evidence to the intuitive notion that the lower the number of predicates in a path, the more likely

it is for the path to be feasible. Our bound, by covering at most one (outermost) cycle with a path, takes into account this notion. Thus, the bound addresses the real problem in branch testing: to find an executable path cover. We have also provided a method to compute the new bound. We have evaluated the β_{branch} bound according to Weyuker's axioms (Weyuker, 1988), a set of desirable properties that a general software complexity measure should satisfy. We have concluded that our bound can also be considered a good metric for the complexity of a program control-flow structure. We have also analyzed the relation between our approach, considered as a new, different testing strategy, and the unified testability theory of Bache and Müllerburg. We have then shown how the β_{branch} bound can also be computed recursively from the decomposition tree of any flowgraph.

Our future research will be devoted to extending the measure of meaningful bounds to other structural testing strategies. Calculation of the bound on other testing strategies would require finding a minimum set of entities notion analogous to the unconstrained arcs for branch testing, which ensures the coverage criterion considered. On the other hand, the notion of weak incomparability should be generalized. This notion must represent the concept of meaningfulness in each particular case. In other words, for each testing strategy, an abstraction of the properties of the relevant entities must be identified. This abstraction will be used to select the subset of entities of interest. We shall try to extend these notions to include a complete family of lower-bounds on the number of test cases needed to achieve various kinds of coverages, for example data flow cover or predicate

condition cover. Finally, we intend to generalize these results to integration testing as well. We believe that it is also important to evaluate the performance of the new bounds through experimentation.

Acknowledgements

The authors wish to express their gratitude to Dr. Boris Beizer for his precious comments and suggestions.

References

R. Bache, R., and M. Müllerburg, Measures of Testability as a Basis for Quality Assurance, *Software Engineering Journal*, 86-92, March 1990.

Beizer, B., *Software Testing Techniques, Second Edition*, Van Nostrand Reinhold, New York, 1990.

Bertolino, A., Unconstrained Edges and Their Application to Branch Analysis and Testing of Programs, *Journal of Systems and Software*, 20 (2), 125-133 (1993).

Deutsch, M. S., and Willis, R. R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Dilworth, R. P., A Decomposition Theorem For Partially Ordered Sets, *Annals Math.*, 51 (1), 161-166 (1950).

Fenton, N. E., *Software Metrics: a Rigorous Approach*, Chapman & Hall, London, 1991.

Fenton N. E. and Whitty, R.W. , Axiomatic Approach to Software Metrication through Program Decomposition, *Computer Journal*, 29 (4), 329-339 (1986).

Ford, L. R. and Fulkerson, D. R., *Flows in Netwoks*, Princeton University Press, New Jersey, 1962.

Hecht, M. S., *Flow Analysis of Computer Programs*, Elsevier, New York, 1977.

Hedley, D.and Hennell, M. A., The Causes and Effects of Infeasible Paths in Computer Programs, *Proc. of the 8th. Int. Conf. on Software Engineering*, London, UK, 259-266, Aug. 1985.

McCabe, T. J., A Complexity Measure, *IEEE Trans. on Software Engineering*, SE 2 (4), 308-320 (1976}.

McCabe, T. J., *Structured Testing*, IEEE Comp. Soc. Press, Silver Spring, Maryland, 1982.

Miller, E. F., Software Testing Technology: An Overview, in *Handbook of Software Engineering*, (C. R. Vick and C. V. Ramamoorthy, eds.), Van Nostrand Reinhold Company, New York, 1984.

Ntafos, S. C., A Comparison Of Some Structural Testing Techniques, *IEEE Trans. on Software Engineering*, SE-14 (6), 868-874 (1988).

Ntafos, S. C. and Hakimi, S. L., On Path Cover Problems in Digraphs and Applications to Program Testing, *IEEE Trans. on Software Engineering*, SE-5 (5), 520-529 (1979).

Rapps, S. and Weyuker, E. J., Selecting Software Test Data Using Data Flow Information, *IEEE Trans. on Software Engineering*, SE-11 (4), 367-375 (1985).

Weyuker, E. J., Evaluating Software Complexity Measures, *IEEE Trans. on Software Engineering*, SE-14 (9), 1357-1365 (1988).

Yates, D. F. and Malevris, N., Reducing the Effects of Infeasible Paths in Branch Testing, *ACM SIGSOFT Software Engineering Notes*, 14 (8), 48-54 (1989).