Using Vapnik-Chervonenkis Dimension to Analyze the Testing Complexity of Program Segments*

Kathleen Romanik†
DIMACS Center for Discrete Mathematics and Theoretical Computer Science
Rutgers University
Piscataway, NJ 08855-1179

Jeffrey Scott Vitter‡
Department of Computer Science
Duke University
Durham, NC 27708-0129

Abstract: We examine the complexity of testing different program constructs. We do this by defining a measure of testing complexity known as VCP-dimension, which is similar to the Vapnik-Chervonenkis dimension, and applying it to classes of programs, where all programs in a class share the same syntactic structure. VCP-dimension gives bounds on the number of test points needed to determine that a program is approximately correct, so by studying it for a class of programs we gain insight into the difficulty of testing the program construct represented by the class. We investigate the VCP-dimension of straight line code, if-then-else statements, and for loops. We also compare the VCP-dimension of nested and sequential if-then-else statements as well as that of two types of for loops with embedded if-then-else statements. Finally, we perform an empirical study to estimate the expected complexity of straight line code.

---

*This paper is a significantly revised version of a preliminary report that appeared under the title "Using Computational Learning Theory to Analyze the Testing Complexity of Program Segments" in *The IEEE Computer Society's Seventeenth Annual International Computer Software and Applications Conference*.

†Part of this research was done while the first author was a graduate student at the University of Maryland at College Park and part was done while she was a postdoctoral fellow at McGill University in Montreal. The research was supported in part by NSF Grant CCR-9112976, ONR Grant N00014-92-J-1254, NSERC, and IRIS National Network of Centres of Excellence.

‡Supported in part by NSF Grant CCR-9007851.

1 Introduction

Program testing is an important subfield of the field of software engineering. Much work has been done in finding methods for selecting test data [GG75, MH81, DMMP87] and in evaluating different testing methodologies [Bud81, DN84, BS87, Ham89]. A related area of software engineering is the study of software complexity. Considerable research has been done in this area as well to devise software complexity measures [Hal77, McC76, WHH79] and to compare various measures for their effectiveness [Wey88, Tia92, TZ92].

Our work combines these two areas by looking at the "testing complexity" of different classes of programs. The reason that we study classes of programs is that by examining the complexity of a class of programs, where each program in the class has the same syntactic structure, we gain insight into the testing complexity of the syntactic structure that these programs share. Therefore, given a class of programs, each with the same syntactic structure, we investigate how difficult it is to distinguish one program in the class from the others using only input/output test pairs. Usually this is impossible to do with 100% accuracy. In other words, for most classes of programs it is impossible to distinguish one program in the class from all other programs that compute a different function when only a finite number of input/output test pairs is used to test the program.

For this reason we introduce another measure of testing complexity. In the field of computational learning theory [VC71, BEHW89] Vapnik-Chervonenkis dimension (or VC-dimension) characterizes the complexity of a class of objects to be learned. We define a similar notion of dimension for program classes, called VCP-dimension, that gives an indication of the testing complexity of these classes. Given a class P of programs computing functions from the reals to the reals, the *VCP-dimension* of P is the VC-dimension of the class obtained by identifying each program in P with the subset of the real line for which it evaluates to a positive number.

Using VCP-dimension we examine the testing complexity of different classes of programs to gain insight into how difficult it is to test various program constructs. In particular, we derive upper and lower bounds on the VCP-dimension of straight line code, if-then-else constructs, and for loops.

We also compare the VCP-dimension of different classes of programs to determine how the complexity increases when program constructs are combined in different ways. For example, we compare nested and sequential if-then-else statements, and we compare for loops with and without upper bounds on the number of iterations. This insight into testing complexity is important because it tells the programmer which types of program structures lead to more easily testable programs, and it shows the tester where more concentrated testing efforts should be applied.

VCP-dimension is similar to pseudo dimension [Pol84, Hau92], which was used by Haussler [Hau92] in proving results about the learnability of real valued functions. As in Haussler's work, we define the *error* of a tested program with respect to a specification to be the expected difference in value between its output and the specification for a randomly drawn input, where there is a probability measure defined over the set of inputs to the program. A program is said to be *approximately correct* with respect to a specification for error bound ϵ if this error is bounded by ϵ. We then define a class of programs to be *randomly approximately testable* if given a specification in the class, a confidence parameter, and an error bound, a finite number of random test points can be selected, and with high probability these test points will ensure that any program that tests correctly on these points will be approximately correct. These notions are defined formally in Section 3.

Haussler [Hau92] used pseudo dimension to get a bound on the number of random examples needed to PAC[1] learn a function from a class of functions. We use his results to find a bound on the number of random test points needed to approximately test a program from a class of programs. This can be done because a random sample of input/output pairs for a function that gives sufficient information to infer the rest of the function is also sufficient to determine that a program correct on the sample has small overall error.

Since the VCP-dimension of a class of functions is always less than the pseudo dimension of the class, our lower bounds on the VCP-dimension of various classes of program segments are also lower bounds on the pseudo dimension of these classes. Although in general pseudo dimension can be arbitrarily greater than VCP-dimension, we use an observation by Macintyre and Sontag [MS93] to show that our upper bounds on the VCP-dimension of different classes of program segments also hold for pseudo dimension.

Thus by combining our bounds on the VCP-dimension of various classes of programs with Haussler's bounds on the number of random examples needed to learn (or test) a function, we can derive results about which types of program classes are randomly approximately testable. For example, our results show that straight line programs with n lines of code are randomly approximately testable using $O(n^2 \log n)$ test points for a given error bound and confidence parameter. We also can show that for loops containing one embedded if-then-else statement and no other embedded constructs can be randomly approximately tested with a number of test points that is polynomial in both the number of lines of code inside the loop and the number of times that the loop is iterated. This means that if there is no upper bound on the number of times that the loop is executed (for example, when the input x is used as the upper bound on the loop index variable), then the construct is *not* randomly approximately testable.

Our work also bears some similarity to the work by Anthony, Bartlett, Ishai and Shawe-Taylor [ABIST94, ABIST95] on valid generalization from approximate interpolation. They

[1] PAC stands for probably almost correct.

say that a set of functions $\mathcal{H}$ (a hypothesis space) *validly generalizes* a set of functions $\mathcal{C}$ (a concept space) *from approximate interpolation* if for any proximity parameter, confidence parameter, and error bound, a finite number of random sample points of a target concept $t \in \mathcal{C}$ can be selected, and with high probability any hypothesis $h \in \mathcal{H}$ that approximates t to within the proximity parameter on the sample points will also approximate t to within the proximity parameter on most inputs in the domain, where there is a probability measure defined over the domain. A hypothesis h approximates a target concept t to within the proximity parameter η on point x_i if $|h(x_i) - t(x_i)| < \eta$. They prove that a set $\mathcal{H}$ of functions from a set X into $[0, 1]$ validly generalizes the set of all functions from X into $\mathbb{R}$ if and only if the pseudo dimension of $\mathcal{H}$ is finite.

Random approximate testability is similar to valid generalization from approximate interpolation where the proximity parameter used is zero. The difference is that in the work of Anthony, Bartlett, Ishai and Shawe-Taylor [ABIST94, ABIST95], the error of a hypothesis is the probability that a randomly drawn input is not within the proximity parameter, rather than being the expected difference in value between the hypothesis and the target for a randomly drawn input.

2 Measuring Testing Complexity

When we examine the testing complexity of classes of programs, we consider only a subset of programs that compute total recursive functions from the rationals ($\mathbb{Q}$) to the rationals. However, we feel that this subset is sufficient to provide insight into the relative testing complexity of different program constructs.

Definition. A *program* p computes a function $f_p : \mathbb{Q} \to \mathbb{Q}$, where a probability measure M is defined over the set of inputs. A *program class* is a set P of programs.

We will use p to denote both the program and the function f_p that it computes when it is clear from the context which meaning is being used. Programs are defined to compute functions with domain $\mathbb{Q}$ because the finite representation of numbers in the computer only allows rationals. However, sometimes it is necessary to consider an extension of such a function to the reals, and in this case we will use the natural extension.

The probability measure M is usually taken to be either the operational distribution on the inputs to the program, or a uniform distribution. In the former case, the error of a program (that is, the probability that an input chosen at random according to M will produce an incorrect output) is a measure of its unreliability; in the latter case, the error measures the fraction of the input domain for which the program computes an incorrect answer. The probability measure M can either be a discrete probability measure on $\mathbb{Q}$ or a continuous probability measure on the reals $\mathbb{R}$.

When defining a measure of testing complexity for a class of programs, we would ideally like a measure that can tell us how many test points (that is, input/output pairs) are needed to distinguish one program from all other programs in the class. We can define this notion formally as follows:

Definition. Given a program class P and a program $p \in P$, a *test set* for p with respect to P is a set of inputs $T \subset \mathbb{Q}$ such that for all other programs $q \in P$, if $p(x) = q(x)$ for all $x \in T$, then $p(x) = q(x)$ for all $x \in \mathbb{Q}$ (that is, p and q compute the same function). The *testing complexity* of the class P is the smallest integer k such that any program $p \in P$ has a test set of cardinality k.

The following results about testing complexity can be proven easily.

Proposition 2.1 *The testing complexity of a program class containing n programs is less than or equal to $n-1$.*

Proof. Let P be a class of n programs, and let $p \in P$ be given. For each $q \in P$ that computes a different function than p, choose an input x for the test set of p such that $p(x) \neq q(x)$. There will be at most $n-1$ such test points. □

Proposition 2.2 *The class of all programs computing polynomials of degree no greater than n has testing complexity $n+1$.*

Proof. Since $n+1$ points completely determine an n degree polynomial, any program in this class has a test set of size $n+1$. □

Although this definition of testing complexity yields some results, it does not allow us to compare different program constructs. This is because when more complicated program constructs are used, even very simple programs become impossible to "test", as the following example illustrates.

Example 2.1 Let the program class P be defined by the following schema:

$$\begin{aligned} P := \{p \mid p(x) := \ & \text{if } a_1 \circ_1 b_1 \text{ then} \\ & \quad \text{output}(a_2 \circ_2 b_2) \\ & \text{else} \\ & \quad \text{output}(a_3 \circ_3 b_3)\}, \end{aligned}$$

where $\circ_1 \in \{=, \neq, <, >, \leq, \geq\}$ and $\circ_2, \circ_3 \in \{+, -, \times, \div\}$ and $a_i, b_i \in \{x\} \cup \mathbb{Q}$

Now, consider the problem of selecting a test set for the program $q \in P$ defined as follows:

$$\begin{aligned} q(x) := \ & \text{if } 0 < 3 \text{ then} \\ & \quad \text{output}(x+2) \\ & \text{else} \\ & \quad \text{output}(x-3). \end{aligned}$$

Since the boolean expression in the if-then-else statement is always true, program q simply computes the linear function $x + 2$. Although this may seem like a contrived program since the boolean expression is obviously always true and one branch is never executed, real programs with more complicated branching structures can contain unexecutable paths that are not easily detected. Barzdin, Bicevskis and Kalninsh [BBK77] proved that the problem of determining which branches of a program are realizable (a branch is realizable if some input causes the branch to be taken during execution of the program) is undecidable.

Proposition 2.3 *The program q has no finite test set with respect to the class P.*

Proof. Suppose a finite test set T for q existed. Let $m := \max\{x \mid x \in T\}$. Define

$$\begin{aligned} q'(x) := \ & \text{if } x \leq m \text{ then} \\ & \quad \text{output}(x+2) \\ & \text{else} \\ & \quad \text{output}(x-2). \end{aligned}$$

Program q' is in P since it has the syntax specified by the class. It computes the same output as q for all inputs in T. However, it computes an incorrect value for all $x > m$, so T is not a test set for q. □

The program q has no test set with respect to P because it does not make use of both branches of the if-then-else statement. Thus, the unused branch can be used by another program in P to "trick" any supposed test set. Such a program looks like q on all inputs in the test set, but it diverges from q on inputs greater than those in the test set.

The example above demonstrates that it is impossible to test most programs to within 100% accuracy, even when they are tested with respect to simple program classes. Since absolute testing is an impossible task, we propose in the next section a less absolute but more meaningful measure of testing complexity. This measure indicates when it is possible to use a small number of randomly selected test points to determine whether any program in a given class is approximately correct with high probability. A program is *approximately correct* if on the average it computes a value "close to" the value of a program that is absolutely correct. These ideas will be made more formal in the next section.

We emphasize that we are not proposing random testing as an effective testing methodology, but rather we are using it as a basis for comparing the testing complexity of different program constructs and combinations of constructs. However, other work has been done recently [BK89, BLR90, Lip91, GLR+91] to make random testing a viable approach to testing software. The problem that is addressed in this other work is how to convert a program that has been shown through random testing to be correct for most inputs into a program that is correct with high probability on *all* inputs. The following simple example illustrates this idea.

Example 2.2 Suppose we are given a deterministic black box program p that computes the function $f(x) = x$. Suppose that we have performed a sufficient number of random tests on p to ensure that it is correct for a large fraction of the possible inputs. The following probabilistic program q calls p as a subroutine and uses a random number generator to compute the function f. If p is correct for a large fraction of the possible inputs, then the probabilistic program q computes f correctly with high probability on any input. It is assumed that q has access to fault-free addition and subtraction operators.

```
q(x) := begin
            y =random;
            z = p(x + y) − p(y);
            output(z)
        end
```

3 Random Approximate Testing

In order to define a meaningful measure of testing complexity, we define a model for random approximate testing of programs and relate a measure to this model. First we define the notion of error for a program.

Definition. Given a program class P and a program $p \in P$, the *error* of a program $q \in P$ with respect to p and the probability measure M is defined by

$$E_M(q,p) = \int_{\mathbb{R}} |q(x) - p(x)| dM(x).$$

In this definition p represents a specification, and q represents a program to be tested against the specification. The error of a program is the expected difference in value between its output and the specification for a randomly drawn input. If this error is bounded by ϵ, then q is *approximately correct* with respect to p for error bound ϵ. Note that the error function E_M is a pseudo-metric[2] on P. When the probability measure M is clear from context, we just use E for the error function.

Now we define what it means for a class of programs to be randomly approximately testable. Let I denote the open interval of rationals $(0,1)$, and let $m: I \times I \to Z^+$ be a positive integer valued function defined on $I \times I$. Let P be a class of programs computing functions from $\mathbb{Q}$ to $\mathbb{Q}$, and let M be a probability measure on $\mathbb{Q}$.

Definition. *P is randomly approximately testable (w.r.t. M) with test set size $m(\epsilon, \delta)$* if for all $\epsilon, \delta \in I$ and for all $p \in P$, if a set T of $m(\epsilon, \delta)$ inputs is selected at random from $\mathbb{Q}$ according to M, then with probability at least $1 - \delta$ for all $q \in P$, if $p(x) = q(x)$ for all $x \in T$ (that is, if q is *consistent* with p on T), then $E_M(q,p) \leq \epsilon$.

If the class P is randomly approximately testable, then given a specification $p \in P$, a confidence parameter $\delta \in I$, and an error bound $\epsilon \in I$, a finite number of random test points can be selected, and with high probability these test points will ensure that any program that tests correctly on these points will be approximately correct. If the function m is a polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$, then the class P is *polynomially randomly approximately testable* or just *polynomially testable.*

3.1 A Complexity Measure for Random Approximate Testing

Now that we have defined a model for random approximate testing of programs, we define a complexity measure that relates to this model. In computational learning theory [VC71, BEHW89] Vapnik-Chervonenkis dimension (or VC-dimension) characterizes the complexity of a class of objects to be PAC learned. If this dimension is finite, then any object in the class can be learned with high probability. This means that if a small number of random examples are selected and labeled according to a chosen object, then with high probability any other object in the class that is consistent with the chosen object on the examples will be approximately equal to that object.

Since our model for random approximate testing is similar to the model of [BEHW89] for PAC learning, we introduce a notion of dimension similar to VC-dimension for program classes that gives an indication of the testing complexity of these classes. This dimension allows us to determine when a small number of test points can be used to demonstrate that

[2] A *pseudo-metric* on a set P is a function $E : P \times P \to \mathbb{R}^+$ such that for all $x, y, z \in P$ the following three properties hold: (1) $x = y \Rightarrow E(x,y) = 0$, (2) $E(x,y) = E(y,x)$, (3) $E(x,y) + E(y,z) \geq E(x,z)$.

a program is approximately correct. In order to define this dimension we first identify a program with the set of intervals for which it evaluates to a positive number. These notions are formalized below.

Definition. Given a program class P, a finite set of inputs $T \subset \mathbb{Q}$ is *shattered* by P if for all $S \subseteq T$, there exists $p \in P$ such that $p(x) > 0$ for all $x \in S$ and $p(x) \leq 0$ for all $x \in T - S$. The *Vapnik-Chervonenkis program dimension* of P (or simply *VCP-dimension(P)*) is the largest integer k such that there exists a subset T of $\mathbb{Q}$ of cardinality k that is shattered by P. If no such k exists, the VCP-dimension of P is infinite.

If we identify each program in a class P with the subset of $\mathbb{Q}$ for which it evaluates to a positive number, then VCP-dimension(P) is simply the Vapnik-Chervonenkis dimension of this class of subsets of $\mathbb{Q}$.

3.2 VCP versus Pseudo Dimension

A similar measure to VCP-dimension, called *pseudo dimension*, has been defined by Haussler [Hau92]. He has shown that if this dimension is "small" for a set of functions, then the set of functions can be PAC learned. We define this measure now and compare it to VCP-dimension.

Definition. ([Hau92]) For a family of functions F from a set S into $\mathbb{R}$ the *pseudo dimension* of F (or $\dim(F)$) is the largest k such that there exist two sequences of length k, $\bar{x} = (x_1, \ldots, x_k) \in S^k$ and $\bar{t} = (t_1, \ldots, t_k) \in \mathbb{R}^k$, and for any subsequence $\bar{y} = (x_{i_1}, \ldots, x_{i_j})$ of $\bar{x}$, there exists $f \in F$ such that $f(x_i) + t_i > 0$ for all $x_i \in \bar{y}$ and $f(x_i) + t_i \leq 0$ for all $x_i \in \bar{x} - \bar{y}$ (that is, elements of the sequence $\bar{x}$ that are not in the subsequence $\bar{y}$). If no such k exists, then $\dim(F)$ is infinite.

The pseudo dimension of a class of functions is the size of the largest set of inputs that can be shattered by the class using a vector ($\bar{t}$) of translation values. Pseudo dimension differs from VCP-dimension because it allows for the possibility of translating a set of functions by a vector of constants before shattering a set of points.

It is easy to see that for any class of functions F, VCP-dimension(F) $\leq \dim(F)$ since a set of inputs that is shattered by F is still shattered if the vector of outputs produced by these inputs is translated by the vector 0^k. If arbitrary classes of functions are considered, then an inverse bound does not exist. That is, there are classes of functions with finite VCP-dimension and infinite pseudo dimension. For example, the class of all positive functions has VCP-dimension 0, but it has infinite pseudo dimension.

Although pseudo dimension can be arbitrarily greater than VCP-dimension, Macintyre and Sontag [MS93] have observed that given a class of functions $F: S \to \mathbb{R}$, a new class of functions $F': S \times \mathbb{R} \to \mathbb{R}$ can be created with the property that $\dim(F) \leq$ VCP-dimension(F')

(in fact, equality holds). This new class of functions F' is created from F by replacing each $f \in F$ with the new function $f(x) - y$, which takes two inputs $x \in S$ and $y \in \mathbb{R}$.

Using the above observation of Macintyre and Sontag, the upper bounds that we obtain in this paper for the VCP-dimension of various classes of programs are the same as those that can be obtained for the pseudo dimension of these classes. This is because the upper bounds on the VCP-dimension still hold if we convert each program class P into a program class P', where each $p' \in P'$ takes two inputs x and y and computes $p(x) - y$ for some $p \in P$. Therefore, the results of Haussler using pseudo dimension that we employ next to bound the number of test points needed to randomly approximately test a program are valid for the classes we consider.

Haussler [Hau92] has used the notion of pseudo dimension to get a bound on the number of random examples needed to PAC learn a function from a class of functions. These results can also be used to find a bound on the number of random test points needed to approximately test a program from a class of programs. This is because a random sample of input/output pairs for a function that gives sufficient information to infer the rest of the function is also sufficient to determine that a program correct on the sample has small overall error.

We now illustrate this relationship between PAC learning and random approximate testing in more detail. First we define several notions used in Haussler's result.

Definition. ([Hau92]) Let F be a family of functions from $S = \mathbb{R} \times \mathbb{R}$ into $[0, K]$. Let M be a probability measure on S. For any $f \in F$ the *expected value* $E(f)$ of f on an example chosen at random from S according to M is $\int_S f(x) dM(x)$. The *empirical estimate* of this expected value on a random sample $\bar{x} \in S^m$ is $\hat{E}_{\bar{x}}(f) := \frac{1}{m} \sum_{i=1}^{m} f(x_i)$. The "closeness" of these two values is given by the metric d_ν, where $\nu > 0$, defined by $d_\nu(x, y) := \frac{|x-y|}{\nu + x + y}$ for any non-negative reals x and y.

Haussler gives the following result relating finite pseudo dimension to PAC learning of functions. This theorem states that for a family of functions F with codomain $[0, K]$ and with finite pseudo dimension, to ensure that with high probability the empirical estimate of the expected value of any function in F on a random sample is "close to" its actual expected value, it is only necessary to choose a random sample whose size is a polynomial in the pseudo dimension of F, the bound K on the codomain of F, and several approximation parameters. The notion of a *permissible* family of functions used in the theorem is a measurability assumption that must be made when the family F is uncountable. Since the programs we examine only use rational constants, they compute countable classes of functions, so we do not need to be concerned with this notion.

Theorem 3.1 [Hau92] *Let F be a permissible family of functions from $S = \mathbb{R} \times \mathbb{R}$ into $[0, K]$ with $dim(F) = d < \infty$. Let M be a probability measure on S. If a random sample $\bar{x}$ of length*

m is drawn from S according to M, and if

$$m \geq \frac{8K}{\alpha^2 \nu}\left(2d \ln \frac{8eK}{\alpha\nu} + \ln \frac{8}{\delta}\right)$$

for $0 < \alpha, \delta < 1$ and for $0 < \nu \leq 8K$, then the probability that there exists $f \in F$ such that $d_\nu(\hat{E}_{\bar{x}}(f), E(f)) > \alpha$ is at most δ.

This theorem applies to PAC learning when the class F is defined to be the class of loss functions associated with a class $\mathcal{F}$ of functions from $\mathbb{R}$ to $\mathbb{R}$ used to estimate or "learn" an unknown distribution of input/output examples. A loss function L_f for a function f measures, for each input to f, the error or "loss" of f on that input. In this case the loss function is defined by $L_f(x, y) = |f(x) - y|$; that is, it measures how much the value given by f differs from a given y for a given input x. If the class of loss functions associated with $\mathcal{F}$ has finite pseudo dimension, then it is possible to use a small random sample to choose a function in $\mathcal{F}$ that has a small loss (that is, its error is close to the infimum of errors over all functions in $\mathcal{F}$) with respect to the unknown distribution of examples. This function will give a good representation of the unknown distribution of examples.

We can also apply Haussler's result to the problem of testing. When testing programs we would like to use a small set of test points to detect programs that vary greatly from a given specification. In particular, we would like to be able to say that any program that is correct on a small set of test points computes a function that is approximately correct with respect to the specification.

In order to apply Haussler's result to testing programs, we must use a class of functions with codomain $[0, K]$. We do this by choosing a maximum loss value K and defining a class of loss functions corresponding to the class of programs P. Since we test with respect to a specification function $p \in P$, we can define the loss function $L^K_{p,q}$ for a tested program q (w.r.t. p) by

$$L^K_{p,q}(x) := \begin{cases} |p(x) - q(x)| & \text{if } |p(x) - q(x)| < K \\ K & \text{otherwise} \end{cases}$$

When we define the loss function in this way, the error of a tested program corresponds to the expected value of its loss function, assuming that the error of a program on a particular input is bounded above by K.

Haussler's work differs from our approach since he uses probability measures defined on $\mathbb{R} \times \mathbb{R}$, rather than just on the domain $\mathbb{R}$ of the class of functions $\mathcal{F}$. However, our work can be put into his framework by defining the following probability measure M_p on $\mathbb{R} \times \mathbb{R}$ corresponding to the probability measure M defined on $\mathbb{Q}$ and the specification program p:

$$M_p(x, y) := \begin{cases} M(x) & \text{if } p(x) = y \\ 0 & \text{otherwise} \end{cases}$$

M_p is a probability measure that is 0 everywhere on $\mathbb{R} \times \mathbb{R}$ except on the graph of p, and its marginal on $\mathbb{R}$ is M.

By making these two adjustments, we can prove a result that is similar to Haussler's and apply it to the testing problem. First we relate the pseudo dimension of the class of loss functions associated with a given program $p \in P$ to that of the class P.

Lemma 3.1 *Let P be a class of programs computing functions from $\mathbb{Q}$ to $\mathbb{Q}$. For any $p \in P$ and $K \in \mathbb{Q}^+$, the class of loss functions $\mathbf{L}_p^K := \{L_{p,q}^K \mid q \in P\}$ associated with p is a subset of the sum of two classes of functions $\mathbf{L}_p^K \subseteq \mathbf{L}_{p,+}^K + \mathbf{L}_{p,-}^K$, such that dim$(\mathbf{L}_{p,+}^K) \leq$ dim(P) and dim$(\mathbf{L}_{p,-}^K) \leq$ dim(P).*

Proof. We show this in three steps. First, by a result of Wenocur and Dudley [WD81], for any $p \in P$, if we define a new class of functions $\mathbf{L}_p := \{q - p \mid q \in P\}$, then $\dim(\mathbf{L}_p) = \dim(P)$. This is easy to see. If $\mathbf{L}_p$ shatters $\bar{x}$ using the translation vector $\bar{t}$, then P shatters $\bar{x}$ using the translation vector $\bar{t}'$, where $t_i' = t_i - p(x_i)$.

Second, for any program $q \in \mathbf{L}_p$, the programs q_+ and q_-, computing the positive and negative part of q respectively, can be defined as follows:

$$q_+(x) := \begin{cases} q(x) & \text{if } q(x) > 0 \\ 0 & \text{otherwise} \end{cases} \qquad q_-(x) := \begin{cases} q(x) & \text{if } q(x) < 0 \\ 0 & \text{otherwise} \end{cases}$$

The classes $\mathbf{L}_{p,+} := \{q_+ \mid q \in \mathbf{L}_p\}$ and $\mathbf{L}_{p,-} := \{q_- \mid q \in \mathbf{L}_p\}$ have the property that $\dim(\mathbf{L}_{p,+}) \leq \dim(\mathbf{L}_p)$ and $\dim(\mathbf{L}_{p,-}) \leq \dim(\mathbf{L}_p)$. Suppose the subset $\mathbf{L}_{p,-}'$ of $\mathbf{L}_{p,-}$ shatters $\bar{x}$ using the translation vector $\bar{t}$. Then $\bar{x}$ is also shattered by $\mathbf{L}_p$ using the same $\bar{t}$. If there exists $x_i \in \bar{x}$ and $q_- \in \mathbf{L}_{p,-}'$ such that $q_-(x_i) = 0$ and $q_-(x_i) + t_i > 0$, then $q(x_i) + t_i > 0$. On the other hand, if there exists $x_i \in \bar{x}$ and $q_- \in \mathbf{L}_{p,-}'$ such that $q_-(x_i) = 0$ and $q_-(x_i) + t_i \leq 0$, then $\bar{x}$ is not shattered by $\mathbf{L}_{p,-}'$ since every $q_- \in \mathbf{L}_{p,-}'$ has $q_-(x_i) + t_i \leq 0$. A similar argument can be made for $\mathbf{L}_{p,+}$.

Third, for the classes $\mathbf{L}_{p,+}$ and $\mathbf{L}_{p,-}$ and for $K \in \mathbb{Q}^+$ we can define the classes $\mathbf{L}_{p,+}^K := \{q^K \mid q \in \mathbf{L}_{p,+}\}$ and $\mathbf{L}_{p,-}^K := \{-q^K \mid q \in \mathbf{L}_{p,-}\}$, where $q^K(x) = q(x)$ if $-K < q(x) < K$, $q^K(x) = K$ if $q(x) \geq K$, and $q^K(x) = -K$ if $q(x) \leq -K$. For both of these classes the pseudo dimension is no more than that of $\mathbf{L}_p$ using the same argument as above.

It is easy to see that $\mathbf{L}_p^K \subseteq \mathbf{L}_{p,+}^K + \mathbf{L}_{p,-}^K$. $\square$

To prove our result on testing programs we use the following notions and theorems from the literature.

Definition. ([Pol84, Hau92]) Let F be a family of functions from a set S into $\mathbb{R}$ and let M be a probability measure on S. For $\epsilon > 0$, the *covering number* $\mathcal{N}(\epsilon, F)$ of F is defined as the smallest m for which there exist functions $g_1, \ldots g_m$ (not necessarily in F) such that for all $f \in F$ there is a g_i with $E_M(f, g_i) \leq \epsilon$. The *$\epsilon$-separation number* $\mathcal{M}(\epsilon, F)$ of F is defined as the largest m for which there exists a set $H \subseteq F$ of functions of cardinality m such that for all distinct $h_i, h_j \in H$, $E_M(h_i, h_j) > \epsilon$. The family F has a nonnegative *envelope* $\mathbf{f}$ if $\mathbf{f}(x) \geq |f(x)|$ for all $f \in F$.

Theorem 3.2 [Hau92] *Let F be a family of functions with envelope $\mathbf{f}$. Then for any $\epsilon > 0$,*

$$\mathcal{M}(2\epsilon, F) \leq \mathcal{N}(\epsilon, F) \leq \mathcal{M}(\epsilon, F).$$

Theorem 3.3 [Pol84, Hau92] *Let F be a family of functions from a set S into $[0, K]$, where $dim(F) = d < \infty$. Let M be a probability measure on S. Then for all $0 < \epsilon \leq K$,*

$$\mathcal{M}(\epsilon, F) < 2\left(\frac{2eK}{\epsilon} \ln \frac{2eK}{\epsilon}\right)^d.$$

Theorem 3.4 [Pol86] *Let F be a permissible family of functions from a set S into $[0, K]$, and let M be a probability measure on S. Assume $\nu > 0, 0 < \alpha < 1$, and $m \geq 1$. Suppose that $\bar{x} \in S^m$ is generated by m independent random draws from S according to M. Then the probability that there exists $f \in F$ such that $d_\nu(\hat{E}_{\bar{x}}(f), E(f)) > \alpha$ is at most*

$$4\mathbf{E}(\mathcal{N}(\alpha\nu/8, F_{|\bar{x}}))e^{-\alpha^2 \nu m/16K}.$$

where $\mathbf{E}$ is expected value and $F_{|\bar{x}}$ is the restriction of F to $\bar{x}$; that is, $F_{|\bar{x}} = \{(f(x_1), \ldots, f(x_m)) \mid f \in F\}$.

Theorem 3.5 [NP87] *If F and G are families of functions with envelopes $\mathbf{f}$ and $\mathbf{g}$, then the class*

$$F + G := \{f + g \mid f \in F, g \in G\},$$

with envelope $\mathbf{f} + \mathbf{g}$, satisfies

$$\mathcal{N}(2\epsilon_1 + 2\epsilon_2, F + G) \leq \mathcal{N}(\epsilon_1, F)\mathcal{N}(\epsilon_2, G).$$

We use the above theorems to prove a result for testing programs that is similar to Haussler's result for learning functions.

Theorem 3.6 *Let P be a class of programs computing functions from $\mathbb{Q}$ to $\mathbb{Q}$ with $dim(P) = d < \infty$, and let M be a probability measure defined over the set of inputs $\mathbb{Q}$. Given $p \in P$ and*

$K \in \mathbb{Q}^+$, define the loss function for any $q \in P$ to be $L^K_{p,q}$ and define $\mathbf{L}^K_p := \{L^K_{p,q} \mid q \in P\}$. Also, define the probability measure M_p on $\mathbb{R} \times \mathbb{R}$ as above. If a random sequence of test points $\bar{x}$ of length m is drawn from $\mathbb{Q}$ according to M, and if

$$m \geq \frac{16K}{\alpha^2\nu}\left(4d \ln \frac{32eK}{\alpha\nu} + \ln \frac{16}{\delta}\right)$$

for $0 < \alpha, \delta < 1$ and for $0 < \nu \leq 8K$, then the probability that there exists $f \in \mathbf{L}^K_p$ such that $d_\nu(\hat{E}_{\bar{x}}(f), E(f)) > \alpha$ is at most δ.

Proof. We follow the form of Haussler's proof. By Theorem 3.4, if a random sequence $\bar{x}$ of m test points is selected, then the probability that there exists $f \in \mathbf{L}^K_p$ such that $d_\nu(\hat{E}_{\bar{x}}(f), E(f)) > \alpha$ is at most

$$4\mathbf{E}(\mathcal{N}(\alpha\nu/8, \mathbf{L}^K_{p,|\bar{x}}))e^{-\alpha^2\nu m/16K}.$$

Since $\mathbf{L}^K_p \subseteq \mathbf{L}^K_{p,+} + \mathbf{L}^K_{p,-}$ by Lemma 3.1, and using Theorem 3.5, this is at most

$$4\mathbf{E}(\mathcal{N}(\alpha\nu/32, \mathbf{L}^K_{p,+,|\bar{x}})\mathcal{N}(\alpha\nu/32, \mathbf{L}^K_{p,-,|\bar{x}}))e^{-\alpha^2\nu m/16K}.$$

By Theorems 3.2 and 3.3 and Lemma 3.1 this is at most

$$16\left(\frac{64eK}{\alpha\nu} \ln \frac{64eK}{\alpha\nu}\right)^{2d} e^{-\alpha^2\nu m/16K}.$$

Setting the above bound equal to δ and solving for m gives

$$m \geq \frac{16K}{\alpha^2\nu}\left(2d \ln\left(\frac{64eK}{\alpha\nu} \ln \frac{64eK}{\alpha\nu}\right) + \ln \frac{16}{\delta}\right).$$

Simplifying this expression using the fact that $\ln(a \ln a) < 2\ln(a/2)$ when $a \geq 5$ gives the final expression. □

In the previous theorem, the program $p \in P$ that is chosen to determine the loss functions and probability measure represents a specification against which other programs must be tested. As stated earlier, the loss function for a tested program q computes its error with respect to the specification p. The empirical estimate $\hat{E}_{\bar{x}}(L^K_{p,q})$ for the loss function of q represents the observed error of q on the test sequence. The expected value $E(L^K_{p,q})$ represents the actual error of q with respect to p.

The theorem states that if P has finite pseudo dimension, then with high probability these two measures of error will be close for all programs in P. With respect to testing this means that any program q that is consistent with p on the test set will have small error for appropriately chosen α, δ and ν. We now state this more formally as a corollary.

Corollary 3.1 *Let P be a class of programs computing functions from $\mathbb{Q}$ to $\mathbb{Q}$ with $dim(P) = d < \infty$, and let M be a probability measure defined over the set of inputs $\mathbb{Q}$. Given a specification $p \in P$, a maximum loss value K, an error bound $0 < \epsilon < 1$, and a confidence parameter $0 < \delta < 1$, if a random sequence of test points $\bar{x}$ of length m is drawn from $\mathbb{Q}$ according to M, and if*

$$m \geq \frac{64K}{\epsilon}\left(4d \ln \frac{64eK}{\epsilon} + \ln \frac{16}{\delta}\right)$$

then with probability at least $1 - \delta$ any tested program q that is consistent with p on $\bar{x}$ (i.e. $q(x) = p(x)$ for all $x \in \bar{x}$) has error no more than ϵ.

Proof. We use the values $\alpha = \frac{1}{2}$ and $\nu = \epsilon$ in the formula for m from Theorem 3.6. The theorem states that with probability at least $1 - \delta$ no program in P will have an error that differs by more than $\frac{1}{2}$ from its observed error on the random test sequence, when d_ϵ is used to measure this difference. For a program q that is consistent with p on the test sequence $\bar{x}$, $\hat{E}_{\bar{x}}(L^K_{p,q}) = 0$. Therefore, in order for

$$d_\epsilon(\hat{E}_{\bar{x}}(L^K_{p,q}), E(L^K_{p,q})) = \frac{|\hat{E}_{\bar{x}}(L^K_{p,q}) - E(L^K_{p,q})|}{\epsilon + \hat{E}_{\bar{x}}(L^K_{p,q}) + E(L^K_{p,q})} = \frac{E(L^K_{p,q})}{\epsilon + E(L^K_{p,q})}$$

to be less than $\frac{1}{2}$, the actual error of q, $E(L^K_{p,q})$, must be less than ϵ. □

In this section we have discussed the relationship between VCP-dimension and pseudo dimension and have shown how pseudo dimension can be used to determine the number of random test points needed to approximately test a program. These dimensions also give an intuitive measure of the testing complexity of a program class, so they can be used to compare the complexity of different program classes. In the following sections we investigate the VCP-dimension of different classes of program segments.

4 Testing Straight Line Programs

First we consider the case of testing straight line programs. We define P_n, the class of straight line programs with n lines of computing code and one output line, as follows:

Definition. Program class P_n is defined by the following schema:

$$\begin{aligned} P_n := \{p \mid p(x) := \; & y_1 = a_1 \circ_1 b_1; \\ & \quad \vdots \\ & y_n = a_n \circ_n b_n; \\ & \text{output}(y_n)\}, \end{aligned}$$

where $\circ_i \in \{+, -, *\}$ and $a_i, b_i \in \mathbb{Q} \cup \{x\} \cup \{y_j \mid j < i\}$.

Each line of code in a program from P_n either adds, subtracts or multiplies a constant, the input x, or a previous expression to either a constant, the input x, or a previous expression. Thus each line of code uses two operands that are polynomials. Since the set of polynomials with rational coefficients is a ring, and rings are closed under addition, subtraction, and multiplication, each $p \in P_n$ computes a polynomial over the rationals.

Definition. The class F_n of functions computed by P_n can be defined inductively as follows:

$$\begin{aligned} F_0 &:= \{x\} \cup \mathbb{Q} \\ F_n &:= \{f(h) \circ g(h) \mid f \in F_i, g \in F_j, h \in F_k, \circ \in \{+, -, *\}, i + j + k \leq n - 1\}, \end{aligned}$$

where the meaning of $f(h)$ for $f \in F_i, h \in F_k$ is that function f performs i elementary operations (from the set $\{+, -, *\}$) using the operands x, h, and any $c \in \mathbb{Q}$.

A function in F_n is built up from smaller functions $f(h)$ and $g(h)$, which have h as their largest common subexpression. The two functions $f(h)$ and $g(h)$ are joined together in the last computing step of a program in P_n. Obviously, $F_n \subseteq F_{n+k}$ for all $k \geq 0$ since any function in F_n can be realized by a program in P_{n+k} that has k lines of code that are not used in computing the final expression.

4.1 Lower Bounds on VCP-Dimension of Straight Line Code

Before investigating the VCP-dimension of the class P_n, we make a few observations. For a class P of polynomials to have VCP-dimension k, it must contain a program p that changes sign at least $k-1$ times. Therefore, p must compute a polynomial of degree at least $k-1$ that has at least $k-1$ distinct real zeros. Using Horner's method [Baa88] it is known that any degree k polynomial can be computed with $2k$ elementary operations (addition, subtraction, multiplication). Therefore VCP-dimension(P_n) is at least $\lfloor \frac{n}{2} \rfloor + 1$. Other work has been done [RS72] to get the number of operations for evaluating a degree k polynomial down to $\frac{3k}{2}$, which means that VCP-dimension(P_n) is even higher.

On the other hand, Borodin and Cook [BC76] have demonstrated a polynomial with $3^{\lfloor n/3 \rfloor}$ real, distinct zeros that can be computed with n operations, so just examining the largest number of real, distinct zeros that can occur in a polynomial computed by a program in P_n is not sufficient to obtain a good bound on the VCP-dimension. Since Borodin and Cook [BC76] have also proven that "most" polynomials of degree greater than or equal to $(n+2)^2$ cannot be computed with n $\pm$ operations, even when an unbounded number of multiplication operations are allowed, it appears that VCP-dimension(P_n) is no more than $O(n^2)$. In fact, in the next section we will prove an upper bound that is close to this one.

We now examine VCP-dimension(P_n) for some small values of n to get a feel for how the dimension grows with the size of the programs. As a base case, programs in P_0 have no computing lines and can only output a constant or the input x, so VCP-dimension(P_0) = 1. The class P_1 contains programs with one computing line and can be enumerated by $P_1 = \{k, kx, x+k, x-k, k-x, x^2\}$, where $k \in \mathbb{Q}$. It is easy to see that VCP-dimension(P_1) = 2.

Proposition 4.1 *VCP-dimension(P_2) = 3.*

Proof. First we show VCP-dimension(P_2) ≥ 3 by showing that the set $T = \{-2, 0, 2\}$ of 3 points is shattered by P_2. We represent a subset of T by an ordered list of $+$ and $-$ signs where a $+$ sign in the i^{th} position means that the i^{th} smallest element of T is in the subset. So, for example, $(+,-,+)$ represents the subset $\{-2, 2\}$ of T. The following list shows each subset of T along with a polynomial computed by a program in P_2 that obtains that subset. Each subset in the left column begins with a $-$ sign, and the corresponding subset in the right column is the negation of this subset, so the polynomial that obtains it is the negation of the polynomial in the left column. Subsets in the left column are ordered by increasing number of $+$ signs.

$(-,-,-)$	-3	$(+,+,+)$	3
$(-,-,+)$	$x-1$	$(+,+,-)$	$1-x$
$(-,+,-)$	$2-x^2$	$(+,-,+)$	x^2-2
$(-,+,+)$	$x+1$	$(+,-,-)$	$-1-x$

Now we show VCP-dimension(P_2) ≤ 3. In order for VCP-dimension(P_2) to be greater than 3, it must be possible to shatter a set of points of size at least 4 with P_2. Therefore, P_2 must contain a program that computes a function that obtains the subset $(+,-,+,-)$ for some set of 4 points. However, this can only be done with a polynomial that has at least 3 distinct real zeros, and the only polynomials of degree greater than 2 in P_2 are x^3 and x^4, which only have one real zero. So P_2 cannot shatter a set with more than 3 points.

Since VCP-dimension(P_2) ≥ 3 and VCP-dimension(P_2) ≤ 3, VCP-dimension(P_2) = 3. □

Proposition 4.2 *VCP-dimension(P_3) = 4.*

Proof. First we show VCP-dimension(P_3) ≥ 4 by showing that the set $T = \{-3, -1, 1, 3\}$ of 4 points is shattered by P_3. The following list shows each subset of T along with a polynomial computed by a program in P_3 that obtains that subset. As before, subsets beginning with a $-$ sign are in the left column.

$(-,-,-,-)$	-3	$(+,+,+,+)$	3
$(-,-,-,+)$	$x-2$	$(+,+,+,-)$	$2-x$
$(-,-,+,-)$	$1-(x-1)^2$	$(+,+,-,+)$	$(x-1)^2-1$
$(-,+,-,-)$	$1-(x+1)^2$	$(+,-,+,+)$	$(x+1)^2-1$
$(-,-,+,+)$	x	$(+,+,-,-)$	$-x$
$(-,+,-,+)$	x^3-4x	$(+,-,+,-)$	$4x-x^3$
$(-,+,+,-)$	$2-x^2$	$(+,-,-,+)$	x^2-2
$(-,+,+,+)$	$x+2$	$(+,-,-,-)$	$-2-x$

Now we show VCP-dimension(P_3) ≤ 4. In order for VCP-dimension(P_3) to be greater than 4, P_3 must contain a program that computes a polynomial that has at least 4 distinct real zeros. However, the only polynomials of degree greater than 3 in P_3 are $kx^4, x^4+k, (x(x+k))^2, (x+k)^4, (x^2+k)^2, x^2(x^2+k), x^4+x^3, x^4+x, x^5, x^6, x^8$ and none of these have 4 distinct real zeros, so P_3 cannot shatter a set with more than 4 points. $\square$

It can similarly be proven that VCP-dimension(P_4) $= 5$. From the above examples we could conjecture that VCP-dimension(P_n) $= n+1$. This is, in fact, a lower bound for VCP-dimension(P_n), but as n increases it is no longer an upper bound, as the following two theorems demonstrate.

Theorem 4.1 *VCP-dimension*(P_n) $\geq n+1$.

Proof. First we demonstrate how to build an n degree polynomial with n distinct real zeros using n operations, and then we show how a set of $n+1$ points can be shattered. If $c_1, c_2, \ldots c_{\lfloor \frac{n}{2} \rfloor}$ are distinct positive rationals, then the following polynomial has n distinct real zeros (if n is odd, a factor of x is inserted):

$$f(x) = (x^2 - c_1^2)(x^2 - c_2^2)\ldots(x^2 - c_{\lfloor \frac{n}{2} \rfloor}^2)$$

This polynomial can be computed in n operations with the following program:

$$\begin{aligned} p(x) := \quad & y_1 = x * x; \\ & y_2 = y_1 - c_1^2; \\ & y_3 = y_1 - c_2^2; \\ & y_4 = y_2 * y_3; \\ & \vdots \\ & y_n = x * y_{n-1}; \quad \text{(if } n \text{ is odd)} \\ & y_n = y_{n-2} * y_{n-1}; \quad \text{(if } n \text{ is even)} \\ & \text{output}(y_n) \end{aligned}$$

The program p builds $\lfloor \frac{n}{2} \rfloor$ degree 2 polynomials, $x^2 - c_i^2$, each with two distinct real zeros, c_i and $-c_i$, and multiplies these together. If n is odd, p multiplies this polynomial by x to make 0 the n^{th} zero.

A set T of $n+1$ points can be chosen by choosing a point between every two zeros of f, as well as a point at either end of the zeros of f. The polynomial f obtains one subset of T with n sign changes, and its negation, which can be obtained in n operations by negating one factor (that is, by using $(c_i^2 - x^2)$), obtains the other subset with n sign changes. Given any subset S of T with less than n sign changes, the following steps will give a polynomial computable in n operations which obtains that subset.

1. Build a polynomial similar to f, except skip the factor $(x^2 - c_i^2)$ if the points in T on either side of c_i or $-c_i$ are assigned the same sign by S. This will "save" two operations. If n is odd and the two points next to 0 are assigned the same sign by S, then remove the factor x.

2. For each c_i for which the factor $(x^2 - c_i^2)$ was skipped in step 1, if c_i $(-c_i)$ has the points on either side of it assigned different signs by S, then place a line through it. That is, insert the additional factor $c_i - x$ $(-c_i - x)$ into the polynomial. This additional factor will change the sign of all points greater than c_i (greater than $-c_i$). It requires two operations, the same number as were "saved" by not inserting the factor $(x^2 - c_i^2)$.

3. If no factors were inserted into the polynomial by either step 1 or step 2, then S contains no sign changes, so it is given by a constant polynomial. If the polynomial obtained by steps 1 and 2 assigns a different sign to the smallest point in T than S does, then it must be negated. This can be done by negating one of its factors.

The polynomial created by the above steps has a zero between every two points of S where a sign change occurs and has no other zeros. Since each of these zeros actually passes through the x axis, and no zero is shared by two or more factors, the resulting polynomial actually changes sign at each zero. Since step 3 ensures that the smallest point in T is assigned the correct sign by the polynomial, all other points will also be assigned the correct sign. □

Theorem 4.2 *For $n > 42$, VCP-dimension(P_n) $> n + 1$.*

Proof. We demonstrate how, using several "tricks", we can shatter a set of $n+2$ points, where $n > 42$. For n even, define $T := \{-(n+1), -(n-1), \ldots, -3, -1, 1, 3, \ldots, n-1, n+1\}$ and for n odd define $T := \{-n, -(n-2), \ldots, -3, -1, 0, 1, 3, \ldots, n-2, n\}$. Observe that any subset of T can be obtained by constructing a polynomial with exactly one zero (a zero where the polynomial actually "crosses" the x axis) between each pair of consecutive points in T where a sign change occurs. Since the points chosen for T are symmetric around 0, any subset of T that has at least one pair of sign changes missing (that is, any subset where there exists a $c \geq 3$ such that $-c$ and $-(c-2)$ are assigned the same sign and c and $c-2$ are assigned the same sign) can be obtained using the techniques in the proof of Theorem 4.1. Also, any subset with three consecutive sign change pairs (that is, any subset where there exists a $c > 6$ such that $-c, -(c-2), -(c-4), -(c-6)$ and $(c-6), (c-4), (c-2), c$ are assigned alternating signs) can be obtained, as the following claim shows.

Claim 4.2a. Any subset of T containing three consecutive sign change pairs can be obtained with n lines of computing code.

Proof. Let S be a subset of T with three consecutive sign change pairs, and let c be the constant such that S assigns alternating signs to $-c, -(c-2), -(c-4), -(c-6)$ and to $(c-6), (c-4), (c-2), c$. Assume S assigns a $+$ sign to $-c$. The following five lines of code will build the polynomial $(x^2-(c-3)^2)((x^2-(c-3)^2)^2-k)$, which evaluates to a positive number for $-c, -(c-4), c-4, c$ and a negative number for $-(c-2), -(c-6), c-6, c-2$ for an appropriately chosen k.

$$\begin{aligned} y_1 &= x * x; \\ y_2 &= y_1 - (c-3)^2; \\ y_3 &= y_2^2; \\ y_4 &= y_3 - k; \\ y_5 &= y_2 * y_4; \end{aligned}$$

An appropriate choice of k is one for which the polynomial $(x^2-(c-3)^2)^2-k$ realizes the sign sequence $+, -, -, +$ for $x = (c-6), (c-4), (c-2), c$, respectively. Such a k exists if

$$\min\{c^2-(c-3)^2, (c-3)^2-(c-6)^2\} \geq \max\{(c-2)^2-(c-3)^2, (c-3)^2-(c-4)^2\}.$$

Reducing these formulas gives

$$\min\{6c-9, 6c-27\} \geq \max\{2c-5, 2c-7\},$$

or $4c \geq 22$, which holds for all $c \geq 6$ Since the restriction on c is $c > 6$, a k with the correct properties always exists.

The above five lines of code produce a polynomial with six zeros (three symmetric pairs of zeros). In addition to the sign changes produced by these zeros, S contains a total of at most $\lceil \frac{n-5}{2} \rceil$ pairs of sign changes and single sign changes (a single sign change occurs when there exists a positive c such that c and $c-2$ are assigned the same sign but $-c$ and $-(c-2)$ are assigned different signs, or visa versa). Since each pair of sign changes can be obtained with two additional lines of code (by including the factor $x^2 - c_i$), and each single sign change can be obtained with two additional lines of code (by including the factor $x - c_i$), at most $2\lceil \frac{n-5}{2} \rceil$ additional lines of code are needed to obtain the rest of the sign changes in S. For n odd $2\lceil \frac{n-5}{2} \rceil = \frac{2(n-5)}{2} = n-5$, so the total number of lines of code needed to obtain S is n.

For n even $2\lceil\frac{n-5}{2}\rceil = \frac{2(n-4)}{2} = n-4$. However, one of the possible sign changes in S is at 0, and this sign change can be obtained in one line by adding the factor x, so a total of n lines of code are needed to obtain S in this case too. □

Now we must demonstrate how to obtain subsets of T that do not have any pairs of sign changes missing and do not contain three consecutive sign change pairs. We claim that such subsets must have one of the following two properties:

1. There exist three single sign changes $k_1 < k_2 < k_3$ (a sign change k_i is defined to be $c-1$ where S assigns different signs to c and $c-2$) such that $k_2 - k_1 = k_3 - k_2$.
2. There exist four single sign changes $k_1 < k_2 < k_3 < k_4$ such that $k_2 - k_1 = k_4 - k_3$.

A subset of T can be represented by a string of length $n+1$ from the alphabet $\Sigma = \{0,1,2\}$. A 2 in the i^{th} position of such a string represents a sign change between the i^{th} and the $(i+1)^{\text{st}}$ points of T that is one of a symmetric pair of sign changes. A 1 in the i^{th} position represents a sign change between the i^{th} and the $(i+1)^{\text{st}}$ points of T that is a single sign change. A 0 in the i^{th} position represents that no sign change occurs in T between the i^{th} and the $(i+1)^{\text{st}}$ points. By searching through longer and longer even length strings (for n even, the possible sign change at 0 is omitted) it is found that all even length strings of length at least 44 that contain at least one sign change of any possible pair of sign changes and do not contain three consecutive sign change pairs satisfy at least one of the above two properties. (Note: A search program was written and executed to do this.) A string of length 42 that does not satisfy either of the two properties is the following: 221122021202202212122220202212212021220022.

If a subset S of T has property 1, then the following four lines of code will build a polynomial that changes sign at k_1, k_2 and k_3:

$$\begin{aligned}
y_1 &= x - k_2;\\
y_2 &= y_1^2;\\
y_3 &= y_2 - (k_3 - k_2)^2;\\
y_4 &= y_3 * y_1;
\end{aligned}$$

S will contain, in addition to these sign changes, a total of at most $\lceil\frac{n-5}{2}\rceil$ pairs of sign changes and single sign changes. A polynomial can be built using $n-5$ operations to obtain these sign changes, as explained in the proof of Claim 4.2a. Using one more operation to multiply the two polynomials together, the subset S can be obtained with n operations.

If a subset S of T has property 2, then the following five lines of code, where $k_5 =$

$k_2 + \frac{k_3 - k_2}{2}$, will build a polynomial that changes sign at k_1, k_2, k_3 and k_4:

$$\begin{aligned} y_1 &= x - k_5; \\ y_2 &= y_1^2; \\ y_3 &= y_2 - (k_3 - k_5)^2; \\ y_4 &= y_2 - (k_4 - k_5)^2; \\ y_5 &= y_3 * y_4; \end{aligned}$$

In addition to these sign changes, S will contain a total of at most $\lceil \frac{n-7}{2} \rceil$ pairs of sign changes and single sign changes, and these can be obtained with $n - 7$ operations. Therefore S can be obtained with $n - 1$ operations.

We have demonstrated how any subset of T can be obtained using no more than n operations. Therefore, for $n > 42$, VCP-dimension(P_n) is at least $n + 2$. $\square$

4.2 Upper Bounds on VCP-Dimension of Straight Line Code

One way to obtain an upper bound on the VCP-dimension of P_n is to determine the largest number of real, distinct zeros occurring in any polynomial computed by a program in P_n. However, since we mentioned at the beginning of the previous subsection that there are programs in P_n that compute polynomials with a number of real, distinct zeros that is exponential in n, this technique does not yield a good upper bound on the VCP-dimension of the class.

To obtain better upper bounds we use recent results by Goldberg and Jerrum [GJ93] in computational learning theory on bounding the VCP-dimension of classes parameterized by real numbers. We use the following theorem from their work.

Theorem 4.3 [GJ93] *Let C be a concept class where concepts and instances are represented by k and n real values, respectively. Suppose that the test for membership of an instance x in a concept c consists of an algorithm $A_{k,n}$ taking $k + n$ real inputs representing c and x, whose runtime is $t = t(k, n)$, and which returns the truth value $x \in c$. The algorithm $A_{k,n}$ is allowed to perform conditional jumps (conditioned on equality and inequality of real values) and execute the standard arithmetic operations on real numbers $(+, -, *, /)$ in constant time. Then VC-Dimension$(C) = O(kt)$.*

In our terminology a concept is a program, and an instance is an input to the program. If we say that an instance x is in a program p ($x \in p$) if and only if $p(x) > 0$, then the VCP-dimension of a program class is the same as the VC-dimension of a class defined as in the above theorem.

Since we only examine programs taking one rational input, an instance can be represented by one real value. A program $p \in P_n$ can be represented by $5n$ real values by using five real numbers to encode the syntax of each line of the program. That is, line i of the program can be represented by parameters $(p_{i,1}, p_{i,2}, p_{i,3}, p_{i,4}, p_{i,5})$, where $p_{i,1}$ encodes the operator, $p_{i,2}$ and $p_{i,3}$ encode the first operand, and $p_{i,4}$ and $p_{i,5}$ encode the second operand. The encoding for $p_{i,1}$ uses "0" for "+", "1" for "−" and "2" for "$*$". The encoding for $p_{i,2}$ and $p_{i,4}$ uses "−1" for a constant, "0" for the input x and "j", where $1 \leq j \leq n-1$, for "y_j". If the operand is a constant, then $p_{i,3}$ or $p_{i,5}$ is used to store the constant, otherwise this parameter is 0 and is not used. As an example, the following program in P_3 can be represented by $p = (2, 0, 0, 0, 0, 1, 1, 0, -1, 6, 0, 2, 0, 0, 0)$:

$$
\begin{aligned}
p(x) := \quad & y_1 = x * x; \\
& y_2 = y_1 - 6; \\
& y_3 = y_2 + x; \\
& \text{output}(y_3)
\end{aligned}
$$

In order to apply Theorem 4.3, we must show how to write an algorithm that takes $5n+1$ real inputs representing a program $p \in P_n$ and an input x and determines if $p(x) > 0$. The algorithm takes $O(\log n)$ time to examine the parameters $p_{i,j}, 1 \leq j \leq 5$ and execute the code for line i of a program. Since a program has n lines of code, the total runtime of the algorithm is $O(n \log n)$. First the algorithm examines $p_{i,1}$ to determine the operation to be performed. Next it performs a one-sided binary search over possible values of $p_{i,2}$ to determine the first operand. It does the same thing to determine $p_{i,4}$. Finally it performs the operation for the i^{th} line of code. The following is part of the code for such an algorithm.

$$
\begin{array}{ll}
l_i: & \text{if } p_{i,1} = 0 \text{ then goto } l_{i,+} \\
 & \text{if } p_{i,1} = 1 \text{ then goto } l_{i,-} \\
 & \text{if } p_{i,2} = -1 \text{ then goto } l_{i,*,c} \\
 & \text{if } p_{i,2} = 0 \text{ then goto } l_{i,*,x} \\
 & \text{if } p_{i,2} > 1 \text{ then goto } l_{i,*,p2.1} \\
 & \text{goto } l_{i,*,y1} \\
l_{i,*,p2.1}: & \text{if } p_{i,2} > 2 \text{ then goto } l_{i,*,p2.2} \\
 & \text{goto } l_{i,*,y2} \\
l_{i,*,p2.2}: & \text{if } p_{i,2} > 4 \text{ then goto } l_{i,*,p2.3} \\
 & \text{if } p_{i,2} = 3 \text{ then goto } l_{i,*,y3} \\
 & \text{goto } l_{i,*,y4} \\
l_{i,*,p2.3}: & \text{if } p_{i,2} > 8 \text{ then goto } l_{i,*,p2.5} \\
 & \text{if } p_{i,2} > 6 \text{ then goto } l_{i,*,p2.4} \\
 & \text{if } p_{i,2} = 5 \text{ then goto } l_{i,*,y5} \\
 & \text{goto } l_{i,*,y6} \\
l_{i,*,p2.4}: & \text{if } p_{i,2} = 7 \text{ then goto } l_{i,*,y7} \\
 & \text{goto } l_{i,*,y8} \\
l_{i,*,p2.5}: & \text{if } p_{i,2} > 16 \text{ then goto } l_{i,*,p2.9}
\end{array}
$$

$$
\begin{array}{ll}
 & \vdots \\
l_{i,*,y3}: & \text{if } p_{i,4} = -1 \text{ then goto } l_{i,*,y3,c} \\
 & \text{if } p_{i,4} = 0 \text{ then goto } l_{i,*,y3,x} \\
 & \text{if } p_{i,4} > 1 \text{ then goto } l_{i,*,y3,p4.1} \\
 & y_i = y_3 * y_1 \\
 & \text{goto } l_{i+1} \\
 & \vdots \\
l_{i,*,y3,c}: & y_i = y_3 * p_{i,5}
\end{array}
$$

Using the algorithm described above, we can apply Theorem 4.3 to obtain the following upper bound on the VCP-dimension of the class P_n of straight line programs with n lines of computing code.

Theorem 4.4 *VCP-dimension*$(P_n) = O(n^2 \log n)$.

As mentioned in Section 3.2, the same upper bound holds for the pseudo dimension of the class P_n.

Corollary 4.1 *dim*$(P_n) = O(n^2 \log n)$.

Proof. If we convert each program $p \in P_n$ to a program p' that takes two inputs x and y and computes $p'(x, y) = p(x) - y$, then we obtain a new class of programs P'_n. An instance for a program in P'_n consists of two real values, and the program itself can be represented by $5n$ real values. We can use the same algorithm as in the proof of Theorem 4.4 to evaluate a program in P'_n by simply adding to the end of the algorithm the line "$y_{n+1} = y_n - y$". Therefore, VCP-dimension$(P'_n) = O(n^2 \log n)$. Since Macintyre and Sontag [MS93] showed that $\dim(P_n) \leq$ VCP-dimension(P'_n), the result follows. □

Since the pseudo dimension of P_n is polynomial, this implies that P_n is polynomially testable by Theorem 3.6.

The upper bound given in the previous result depends on the operations allowed in a straight line program. If we have a different set of basic operations, then we get different bounds on the VCP-dimension. For example, if we allow not only addition, subtraction and multiplication, but also exponentiation (denoted by $\uparrow$) and floor functions (denoted by $\lfloor\rfloor$), then we find that the VCP-dimension of straight line programs is infinite.

Definition. Program class P_n^* is defined by the following schema:

$$P_n^* := \{p \mid p(x) := \begin{array}{l} y_1 = e_1; \\ \quad\vdots \\ y_n = e_n; \\ \text{output}(y_n)\}, \end{array}$$

where $e_i = \lfloor a_i \rfloor$ or $e_i = a_i \circ_i b_i$ and $\circ_i \in \{+, -, *, \uparrow\}$ and $a_i, b_i \in \mathbb{Q} \cup \{x\} \cup \{y_j \mid j < i\}$.

Each line of code in a program from P_n^* either takes the floor of an operand, performs an exponentiation, or adds, subtracts or multiplies two operands. The operands permitted are constants, the input x, or a previous expression.

Theorem 4.5 *For $n \geq 7$, VCP-dimension(P_n^*) is infinite.*

Proof. We use techniques from [GJ93] for proving lower bounds to demonstrate a subclass of P_7^* that can shatter the set $\{0, 1, 2, \ldots, d-1\}$ for any $d > 0$. Since dummy lines can always be added to a program, this shows that VCP-dimension(P_n^*) is infinite for any $n \geq 7$.

The set $S = \{0, 1, 2, \ldots, d-1\}$ can be shattered by the 2^d programs of the following form, where $a_j = j * 2^{-d}$ for $0 \leq j < 2^d$. The bit representation of each a_j represents a different subset of S. When an integer $x \geq 0$ is input to program p_j, p_j extracts and outputs the $(x+1)^{\text{st}}$ bit to the right of the decimal point in a_j. Thus the program p_j obtains the subset $\{x_1, \ldots, x_k\}$ where the $(x_i+1)^{\text{st}}$ bit to the right of the decimal point in a_j is 1 for $1 \leq i \leq k$ and all other bits are 0. To make the presentation clearer, we show the binary representations of most of the intermediate variables.

$$\begin{array}{lll} p_j(x) := & y_1 = a_j; & (y_1 = 0.b_0 b_1 \cdots b_{d-1}) \\ & y_2 = 2 \uparrow x; & \\ & y_3 = y_1 * y_2; & (y_3 = b_0 \cdots b_{x-1}.b_x \cdots b_{d-1}) \\ & y_4 = \lfloor y_3 \rfloor; & (y_4 = b_0 \cdots b_{x-1}) \\ & y_5 = y_3 - y_4; & (y_5 = 0.b_x \cdots b_{d-1}) \\ & y_6 = 2 * y_5; & (y_6 = b_x.b_{x+1} \cdots b_{d-1}) \\ & y_7 = \lfloor y_6 \rfloor; & (y_7 = b_x) \\ & \text{output}(y_7) & \end{array}$$

Note that only the values of the constants a_j are dependent on d, so for any $d > 0$ a set of 2^d programs from P_7^* can be constructed to shatter $\{0, 1, 2, \ldots, d-1\}$. $\square$

4.3 Empirically Investigating Complexity

Since there is a gap between the upper and lower theoretical bounds on the VCP-dimension of the class P_n of straight line programs, we performed an empirical study to estimate the complexity of this class of programs. An empirical study is also important to give an indication of the "average" complexity of the class, as opposed to the worst case complexity indicated by the VCP-dimension. Since the VCP-dimension of a program class is determined by the *existence* of a set of inputs that can be shattered, the VCP-dimension may be high even though most sets of inputs of this size cannot be shattered. Also, even though a set of n inputs can be shattered by a program class, it may not be possible to shatter the set of inputs with most sets of 2^n programs.

The algorithm for empirically investigating complexity goes as follows:

Algorithm to Estimate Expected Complexity

1. Generate a random input sequence of length m.
2. Generate $m * 2^m$ random programs of length n.
3. For each program, determine the subset of the input sequence that it gives.
4. Count the number of different subsets obtained by the programs.
5. Repeat steps 1-4 several times to find the average number of distinct subsets obtainable by a large set of programs on an input sequence of length m.

The number of different subsets obtained gives a lower bound on the VCP-dimension of the class, using Sauer's Lemma [Ass83]. This result states that if the VCP-dimension of a class is d, then for an input sequence S of length m, where $m \geq d$, the class will obtain no more than $\sum_{i=0}^{d} \binom{m}{i} \leq m^d + 1$ different subsets of S. Therefore, if a randomly selected sample from the class P_n obtains k subsets of an input sequence of length m, then the dimension of P_n is at least $\log_m(k-1)$.

Using this result and the empirical data in Table 1, we obtain a lower bound estimate of three for VCP-dimension(P_n) for n ranging from 10 to 20. This empirical lower bound is less than the theoretical lower bound found in Section 4.1. This is because the sets of programs in P_n that can shatter input sequences of length $n + 1$ have a small probability.

Although the empirical data do not yield a useful bound on VCP-dimension(P_n), they do give an estimate of the expected complexity of the class. We now define this notion more formally.

Table 1: Empirical Data of Expected Complexity

$n\backslash m$	8	9	10	11	12	13	14	15	16
10	48	72	112	170	266	400	633	1031	1574
11	50	80	119	183	281	439	710	1106	1797
12	50	76	122	178	272	437	656	1054	1739
13	55	81	124	190	303	464	758	1207	1914
14	54	83	126	199	298	482	775	1220	2040
15	54	87	134	206	327	530	830	1338	2153
16	54	88	132	201	312	489	781	1260	2012
17	58	86	135	216	343	510	832	1333	2107
18	58	92	139	225	329	547	869	1422	2291
19	58	95	142	225	355	576	904	1454	2327
20	59	91	144	228	341	558	870	1396	2289

Definition. The *expected complexity* of a class of programs P for input size m is the expected number of subsets of a random input sequence of length m that can be obtained by a set of $m2^m$ randomly chosen programs.

We chose the bound $m2^m$ in the above definition for the following reason. If all subsets of an input sequence of length m were equally likely to be obtained by a random program, then by a well known probabilistic result, $m2^m$ random programs would be sufficient to determine the number of subsets obtainable. The probabilistic result states that if n equally likely balls are in an urn and $n \log n$ independent draws with replacement are made of the balls, then the expected number of different balls seen will be almost n. The "balls" in our case are subsets of an input sequence of length m, so n is at most 2^m, and thus at most $m2^m$ random programs would be needed to determine the number of subsets. For the program classes that we study, however, all subsets are not equally likely, but we are interested in knowing the expected number of subsets that will be obtained from this same sample size.

For the class P_n, the expected complexity is a function of both n and m. The empirical data in Table 1 indicate that the expected complexity of P_n does not grow very quickly. Thus it is feasible to use random testing of programs with a uniform probability distribution to ensure that with high probability a program in P_n computes a function that is approximately correct.

5 Testing If-Then-Else Statements

We now examine more complicated program classes to determine the difficulty of testing various program constructs. We begin by looking at programs containing if-then-else statements.

Definition. We define the class $P_{\text{if}}(P', P'')$ of programs containing an if-then-else statement as follows:

$$
\begin{aligned}
P_{\text{if}}(P', P'') := \{p \mid p(x) := \ & \text{if } a \circ b \text{ then} \\
& \quad p_1(x) \\
& \text{else} \\
& \quad p_2(x); \\
& \text{output}(y_k)\},
\end{aligned}
$$

where $\circ \in \{=, \neq, <, >, \leq, \geq\}$ and $a, b \in \{x\} \cup \mathbb{Q}$ and $p_1 \in P'$ and $p_2 \in P''$.

$P_{\text{if}}(P', P'')$ contains programs with an if-then-else statement where the then-clause contains a program segment from P' and the else-clause contains a program segment from P''. The boolean expression in the if-then-else statement of these programs cannot be more complicated than a comparison of the input with a constant. The output is a variable that is assigned a value in one of the two clauses.

Using this general definition we can then define the following class of programs:

$$P_{\text{if}}^k := P_{\text{if}}(P_k, P_k)$$

Each branch of a program in P_{if}^k contains k lines of straight line computing code. Let F_{if}^k denote the class of functions computed by P_{if}^k. It is obvious that $F_k \subset F_{\text{if}}^k$ since any function in F_k can be computed by an if-then-else program that has its boolean expression always evaluate to true and uses only the p_1 block of code for computing.

5.1 Complexity of If-Then-Else Statements

In this section we give upper and lower bounds on the VCP-dimension of the classes $P_{\text{if}}(P', P_k)$ and P_{if}^k.

Theorem 5.1 *VCP-dimension$P_{\text{if}}(P', P_k) \geq$ VCP-dimension(P')+ VCP-dimension(P_{k-1}).*

Proof. Let VCP-dimension$(P') = m$, VCP-dimension$(P_{k-1}) = n$, S be a set of m points shattered by P', and T be a set of n points shattered by P_{k-1}. Furthermore, let $t_{\min} := \min(T)$ and $s_{\max} := \max(S)$, and define the set $ST := S \cup \{s_{\max} + 1 + x - t_{\min} \mid x \in T\}$. The set ST has cardinality $m+n$ and contains the set S and a copy of T that is shifted to the right of S. It can be shattered by $P_{\text{if}}(P', P_k)$ as follows. Given any subset $ST' \subseteq ST$, let S' be the subset of ST' whose elements are less than or equal to $s_{\max}$, and let T' be the subset whose elements are greater than $s_{\max}$. The set S' is a subset of S, so there exists $p_1 \in P'$ that obtains this subset. Similarly, the set $\{x - s_{\max} - 1 + t_{\min} \mid x \in T'\}$ is a subset of T, so there exists $p_2 \in P_{k-1}$ that obtains this subset. Using this program we can obtain the k line program $p_2'(x) := y_1 = x - s_{\max} - 1 + t_{\min}; p_2(y_1)$, which obtains the subset T'. Therefore, the program

$$\begin{aligned} p(x) := \ & \text{if } x \le s_{\max} \text{ then} \\ & \quad p_1(x) \\ & \text{else} \\ & \quad p_2'(x); \\ & \text{output}(y_k) \end{aligned}$$

which is in $P_{\text{if}}(P', P_k)$, will obtain the subset ST' of ST. Since the subset ST' was arbitrarily chosen, any subset of ST can be obtained by $P_{\text{if}}(P', P_k)$, so $P_{\text{if}}(P', P_k)$ shatters set ST. □

Notice that the same result can be proven for $P_{\text{if}}(P_k, P')$.

Corollary 5.1 *VCP-dimension*$(P_{\text{if}}^k) \ge$ *VCP-dimension*$(P_k)+$ *VCP-dimension*(P_{k-1}).

Theorem 5.2 *VCP-dimension*$(P_{\text{if}}(P', P')) \le 2($*VCP-dimension*$(P')) + 1$.

Proof. Let VCP-dimension$(P') = n$, and suppose set T of cardinality $2n+2$ is shattered by $P_{\text{if}}(P', P')$. Consider the $n+1$ smallest points in T. Since no set of $n+1$ points can be shattered by P', there exists some subset S of these points that cannot be obtained by any program in P'. Now, consider all subsets of T that have S as the subset of the smallest $n+1$ points of T. There are 2^{n+1} such subsets, one for each possible subset of the largest $n+1$ points of T. For each of these subsets of T, the program in $P_{\text{if}}(P', P')$ that obtains this subset must use both blocks of code (that is, both clauses of the if-then-else) to obtain the subset S. Therefore, its boolean expression must divide the $n+1$ smallest points of T into two sets. However, since the only boolean expressions allowed are ones that compare x to a constant, any boolean expression that divides the $n+1$ smallest points of T into two sets must include all the $n+1$ largest points of T in one of these two sets. This means that each subset of T containing S obtains a subset of the $n+1$ largest points of T using only one block of code. Therefore, this set of $n+1$ points is shattered by P'. This contradicts the fact that VCP-dimension$(P') = n$. □

Note that the above theorem *cannot* be generalized to VCP-dimension($P_{\text{if}}(P', P'')$) $\leq$VCP-dimension(P')+ VCP-dimension(P'') + 1. Although VCP-dimension(P') $= m$ and VCP-dimension(P'') $= n$ implies that for any set of $m + n + 2$ points, the $m + 1$ smallest points will not be shattered by P' and the $n + 1$ largest points will not be shattered by P'', it may be possible to shatter a set of $m + n + 2$ points with programs in $P_{\text{if}}(P', P'')$. This is because a subset of the $m + 1$ smallest points that is unobtainable by P' may be obtained by either using the boolean condition of a program in $P_{\text{if}}(P', P'')$ to split this set in two, or by using a program from P'' to obtain it. For example, let P' on the set of inputs $T = \{t_1, t_2, t_3, t_4\}$ be $P' := \{(1,1,1,1), (0,1,1,1), (1,0,1,1), (1,1,0,1), (1,1,1,0)\}$ and let $P'' := \{(0,0,0,0), (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)\}$. Then VCP($P'$) $= 1$ and VCP(P'') $= 1$, but $P_{\text{if}}(P', P'')$ shatters T.

Corollary 5.2 *VCP-dimension(P^k_{if}) $\leq 2($VCP-dimension$(P_k)) + 1$.*

Any program in P^k_{if} can be represented by $10k + 5$ real values, where five real numbers are used to encode the syntax of each line inside one of the two branches and five additional numbers are used to encode a, b and $\circ$. An algorithm that evaluates a program encoded in this way can be written similarly to the one for programs in P_n. The runtime of the algorithm will be $O(k \log k)$. Therefore, it can be shown that the pseudo dimension of P^k_{if} is polynomial in k and thus P^k_{if} is polynomially testable.

5.2 Nested vs. Sequential If-Then-Else Statements

In most large programs, many if-then-else statements are used. Sometimes these statements follow each other in sequential order and sometimes they are nested. We now define and compare program classes containing nested if-then-else statements to classes containing sequential if-then-else statements to determine which are more difficult to test.

Definition. We define the class $P^{n,k}_{\text{nest-if}}$ of programs containing n nested if-then-else statements as follows:

$$
\begin{aligned}
P^{n,k}_{\text{nest-if}} := \{p \mid p(x) := {} & \text{if } a_1 \circ_1 b_1 \text{ then} \\
& \quad \text{if } a_2 \circ_2 b_2 \text{ then} \\
& \qquad \vdots \\
& \qquad\quad \text{if } a_n \circ_n b_n \text{ then} \\
& \qquad\qquad p_1(x) \\
& \qquad\quad \text{else} \\
& \qquad\qquad p_2(x) \\
& \qquad \vdots \\
& \quad \text{else} \\
& \qquad p_n(x) \\
& \text{else} \\
& \quad p_{n+1}(x); \\
& \text{output}(y_k)\},
\end{aligned}
$$

where $\circ_i \in \{=, \neq, <, >, \leq, \geq\}$ and $a_i, b_i \in \{x\} \cup \mathbb{Q}$ and $p_i \in P_k$.

Definition. We define the class $P^{n,k}_{\text{seq-if}}$ of programs containing n sequential if-then-else statements as follows:

$$
\begin{aligned}
P^{n,k}_{\text{seq-if}} := \{p \mid p(x) := {} & \text{if } a_1 \circ_1 b_1 \text{ then} \\
& \quad p_1(x) \\
& \text{else} \\
& \quad p_2(x); \\
& \quad \vdots \\
& \text{if } a_n \circ_n b_n \text{ then} \\
& \quad p_{2n-1}(x) \\
& \text{else} \\
& \quad p_{2n}(x); \\
& \text{output}(y_{nk})\},
\end{aligned}
$$

where $\circ_i \in \{=, \neq, <, >, \leq, \geq\}$ and $a_i, b_i \in \{x\} \cup \mathbb{Q}$ and $p_i \in P_k$.

Since $P^{n,k}_{\text{nest-if}} = P_{\text{if}}(P^{n-1,k}_{\text{nest-if}}, P_k)$, then by Theorem 5.1 and induction on n we can show that VCP-dimension$(P^{n,k}_{\text{nest-if}}) \geq n$(VCP-dimension (P_{k-1}))+ VCP-dimension(P_k). The following theorem finds an upper bound for VCP-dimension$(P^{n,k}_{\text{nest-if}})$ using a similar argument as in the case of one if-then-else statement.

Theorem 5.3 *VCP-dimension*$(P^{n,k}_{\text{nest-if}}) \leq (n+1)($*VCP-dimension*$(P_k)) + n$.

Proof. Let VCP-dimension$(P_k) = m$, and suppose set T of cardinality $(n+1)(m+1)$ is shattered by $P^{n,k}_{\text{nest-if}}$. Consider the partition of T into $n+1$ sets of $m+1$ points, where the $m+1$ smallest points of T are in the first set, the next $m+1$ points are in the next set and so on. Since no set of $m+1$ points can be shattered by P_k, there exists a subset S_i for each of these sets of points that cannot be obtained by any program in P_k. Now, consider the subset of T that is the union of all the S_i. To obtain this subset of T, a program in $P^{n,k}_{\text{nest-if}}$ must contain boolean expressions that divide each of the $n+1$ sets of points into different blocks of code. However, since the only boolean expressions allowed are ones that compare x to a constant, a boolean expression can divide at most one set of points. Therefore, $n+1$ boolean expressions are needed to do this. But programs in $P^{n,k}_{\text{nest-if}}$ only contain n boolean expressions, so no program in $P^{n,k}_{\text{nest-if}}$ will obtain the subset of T that is the union of the S_i. $\square$

In order to compare nested if-then-else constructs to sequential if-then-else constructs, we give the following theorem which gives a lower bound for VCP-dimension$(P^{n,k}_{\text{seq-if}})$.

Theorem 5.4 *VCP-dimension*$(P^{n,k}_{\text{seq-if}}) \geq$ *VCP-dimension*(P_{nk})+ *VCP-dimension*$(P_{nk-1}) \geq 2nk+1$.

Proof. We use a technique similar to that used in the proof of Theorem 5.1 to prove this theorem. Let VCP-dimension$(P_{nk}) = m$ and let T be a set of m points shattered by P_{nk}. Similarly, let VCP-dimension$(P_{nk-1}) = m'$ and let T' be a set of m' points shattered by P_{nk-1}. Let $d := \max(T)$ and let $d' := \min(T')$. If $d' > d$ define the set S to be $S := T \cup T'$; otherwise define S to be $S := T \cup \{x + (d - d') + 1 \mid x \in T'\}$. Set S, which has cardinality $m + m'$, contains T and a (perhaps shifted) copy of T'.

Any subset of S can be obtained by a program in $P^{n,k}_{\text{seq-if}}$ of the following form:

$$
\begin{array}{l}
p(x) := \text{if } x \leq d \text{ then} \\
\qquad p_1(x) \\
\qquad \text{else} \\
\qquad p_2(x); \\
\qquad \vdots \\
\qquad \text{if } x \leq d \text{ then} \\
\qquad p_{2n-1}(x) \\
\qquad \text{else} \\
\qquad p_{2n}(x); \\
\qquad \text{output}(y_{nk})\}
\end{array}
$$

where each $p_i \in P_k$. Since all the boolean expressions are the same, there are only two paths that the program can take. If the input is less than or equal to d, then the program executes $p_1; p_3; \ldots p_{2n-1}$; otherwise it executes $p_2; p_4; \ldots p_{2n}$. The code blocks $p_1; p_3; \ldots p_{2n-1}$ are chosen such that together they compute a function from P_{nk} that obtains the desired subset of the m smallest points of S (that is, a subset of T). The code blocks $p_2; p_4; \ldots p_{2n}$ are chosen such that the first line is $y_1 = x$ if $S = T \cup T'$ or is $y_1 = x - ((d - d') + 1)$ if $S = T \cup \{x + (d - d') + 1 \mid x \in T'\}$. The other $nk - 1$ lines combine to compute a function from P_{nk-1} that uses y_1 as its input parameter and obtains the desired subset of T'. That is, it obtains the subset of T' whose shifted copy is the desired subset of the m' largest points in S.

Using this technique, any subset of S can be obtained, so S is shattered by $P^{n,k}_{\text{seq-if}}$. $\square$

In the following corollary we combine the results of Theorems 5.3 and 5.4 to compare the nested branching construct with the sequential branching construct.

Corollary 5.3 *For large n and k, VCP-dimension($P^{n,k}_{\text{seq-if}}$) $>$ VCP-dimension($P^{n,k}_{\text{nest-if}}$).*

Proof. Since an exact bound on VCP-dimension(P_k) is not known, we consider two cases.

1. VCP-dimension(P_k) is linear. That is, VCP-dimension(P_k) $= ck + d$ where $c, d \in \mathbb{Q}$. Then by Theorem 5.3, VCP-dimension($P^{n,k}_{\text{nest-if}}$) $\leq (n+1)(ck+d)+n = nck+nd+ck+d+n$. By Theorem 5.4, VCP-dimension($P^{n,k}_{\text{seq-if}}$) $\geq cnk + d + c(nk-1) + d = 2cnk + 2d - c$. By combining these two inequalities we see that for $n > \frac{ck-d+c}{ck-d-1}$, $2cnk + 2d - c > nck+nd+ck+d+n$, and therefore VCP-dimension($P^{n,k}_{\text{seq-if}}$) $>$ VCP-dimension($P^{n,k}_{\text{nest-if}}$).

2. VCP-dimension(P_k) is not linear. In this case, for sufficiently large n, n(VCP-dimension (P_k)) $<$ VCP-dimension (P_{nk}). Also, since the highest degree polynomial computable in k steps is x^{2^k}, for sufficiently large k, 2(VCP-dimension (P_k)) $>$ VCP-dimension (P_{k+1}). By combining these two facts and using Theorem 5.4 we see that for large n and k, VCP-dimension ($P^{n,k}_{\text{seq-if}}$) $\geq$ VCP-dimension (P_{nk})+VCP-dimension (P_{nk-1}) $>$ VCP-dimension (P_{nk})$+\frac{1}{2}$(VCP-dimension(P_{nk})) $> n$(VCP-dimension (P_k))$+\frac{n}{2}$(VCP-dimension(P_k)) $= n$(VCP-dimension (P_k))+VCP-dimension(P_k)$+(\frac{n}{2}-1)$(VCP-dimension (P_k)). For $k > 1$ and n sufficiently large, $(\frac{n}{2}-1)$(VCP-dimension (P_k)) $> n$. By Theorem 5.3, VCP-dimension ($P^{n,k}_{\text{nest-if}}$) $\leq n$(VCP-dimension (P_k))+VCP-dimension (P_k) $+ n$. By combining this with the previous inequality we see that for large n and k, VCP-dimension ($P^{n,k}_{\text{seq-if}}$) $>$ VCP-dimension ($P^{n,k}_{\text{nest-if}}$).

□

This corollary shows that although nested branching constructs may be harder to understand from a programmer's point of view, they are actually less complicated from a testing point of view than are sequential branching statements.

6 Testing For Loops

Iteration is an important programming construct, so in this section we examine the complexity of iteration. In particular, we look at programs containing for loops.

Definition. We define the class $P^{n,k}_{\text{for}}$ of programs containing one for loop as follows:

$$\begin{aligned}
P^{n,k}_{\text{for}} := \{p \mid p(x) := \ & y_j = c; \\
& \text{for } i = l \text{ to } u \text{ do} \\
& \qquad y_1 = a_1 \circ_1 b_1; \\
& \qquad\quad \vdots \\
& \qquad y_k = a_k \circ_k b_k; \\
& \text{output}(y_k)\},
\end{aligned}$$

$$\text{where } 1 \leq j \leq k \text{ and } c \in \{x\} \cup \mathbb{Q} \text{ and } l, u \in Z \text{ and } u - l < n$$

$$\text{and } \circ_m \in \{+, -, *\} \text{ and } a_m, b_m \in \mathbb{Q} \cup \{x, i\} \cup \{y_h \mid 1 \leq h \leq k\}.$$

$P^{n,k}_{\text{for}}$ contains programs with one for loop where the loop index is bounded above and below by constants and the number of times through the loop is no more than n. There is one initialization line before the loop and k lines of straight line computing code inside the loop. Since a for loop must be able to access values computed in the previous iteration of the loop, the righthand side of each line inside the loop is allowed to use any y_m value, for m between 1 and k. In order to avoid the problem of uninitialized variables, we assume that all variables y_m are initialized to 0 before the program is executed.

6.1 Complexity of For Loops

Since a for loop with constant bounds on the index variable can be unrolled into a straight line program, every program in $P_{\text{for}}^{n,k}$ can be translated into an equivalent program in P_{nk+1}. Then by applying Theorem 4.4 we can obtain an upper bound of $O((nk)^2 \log(nk))$ on the VCP-dimension of $P_{\text{for}}^{n,k}$. However, the structure of programs in $P_{\text{for}}^{n,k}$ is more restrictive than that of programs in P_{nk+1}, so by making a closer examination of this structure we can obtain better upper bounds on the VCP-dimension.

Any program in $P_{\text{for}}^{n,k}$ can be represented by $5k + 5$ real values, where five real numbers are used to encode the syntax of each line inside the loop and additional numbers are used to encode y_j, c, l and u. Using this encoding an algorithm can be written that takes as input a program $p \in P_{\text{for}}^{n,k}$ and an input $x \in \mathbb{Q}$ and determines if $p(x) > 0$. The first part of the algorithm is similar to the algorithm for straight line code, except that it just determines the syntax of the body of the for loop without executing any lines of code. Since the binary search to determine the operands for each line of code only needs to search from -2 to k ("-2" is used to represent the loop index "i"), the runtime of the first part of the algorithm is $O(k \log k)$. Next the body of the for loop is executed by the following piece of the algorithm:

$$
\begin{array}{rl}
 & y_1 = 0 \\
 & \quad \vdots \\
 & y_k = 0 \\
 & y_j = c \\
 & i = l \\
\text{loop:} & y_1 = a_1 \circ_1 b_1 \\
 & \quad \vdots \\
 & y_k = a_k \circ_k b_k \\
 & i = i + 1 \\
 & \text{if } i \leq u \text{ then goto loop}
\end{array}
$$

This piece of the algorithm has a runtime of $O(nk)$ since it must execute the body of the for loop once for each value of the index variable. Thus the total runtime for the algorithm is $O(k \log k + nk)$. Since each program is represented by $O(k)$ real values, Theorem 4.3 can be applied to obtain the following upper bound on the VCP-dimension.

Theorem 6.1 *VCP-dimension*$(P_{\text{for}}^{n,k}) = O(k^2(\log k + n))$.

As in the proof of Corollary 4.1, the above algorithm for evaluating programs in $P_{\text{for}}^{n,k}$ can easily be changed to evaluate programs of the form $p'(x,y) = p(x) - y$, where $p \in P_{\text{for}}^{n,k}$, without changing the runtime of the algorithm. Thus we get the following corollary, which shows that $P_{\text{for}}^{n,k}$ is polynomially testable.

Corollary 6.1 *dim*$(P_{\text{for}}^{n,k}) = O(k^2(\log k + n))$.

6.2 Combining For Loops and If-Then-Else Statements

We now examine the complexity of two program constructs formed by combining iteration and branching. Each of these constructs consists of a for loop with an embedded if-then-else statement. In one construct the loop index is bounded above and below by constants, and in the other construct the upper bound on the loop index is the input x. This change produces a large difference in the testing complexity of the two constructs.

Definition. We define the class $P_{\text{for}(n,k),\text{if}(m)}$ of programs containing a for loop with an embedded if-then-else statement and a constant upper bound on the loop index as follows:

$$
\begin{aligned}
P_{\text{for}(n,k),\text{if}(m)} := \{p \mid p(x) := \; & y_j = a; \\
& \text{for } i = l \text{ to } u \text{ do} \\
& \qquad p_1(x); \\
& \qquad \text{if } b \circ c \text{ then} \\
& \qquad\quad q_1(x) \\
& \qquad \text{else} \\
& \qquad\quad q_2(x); \\
& \qquad p_2(x); \\
& \text{output}(y_{k+m})\},
\end{aligned}
$$

$$\text{where } 1 \le j \le k+m \text{ and } a \in \{x\} \cup \mathbb{Q} \text{ and } l, u \in Z \text{ and } u - l < n$$

$$\text{and } p_1 \in P'_{k_1}, p_2 \in P'_{k_2}, q_i \in P'_m \text{ and } k_1 + k_2 = k$$

$$\text{and } \circ \in \{=, \neq, <, >, \le, \ge\} \text{ and } b \in \mathbb{Q} \cup \{x, i\} \cup \{y_j \mid j \le k_1\} \text{ and } c \in \mathbb{Q}$$

$$
\begin{aligned}
\text{and } P'_n := \{p \mid p(x) := \; & y_1 = a_1 \circ_1 b_1; \\
& \vdots \\
& y_n = a_n \circ_n b_n; \},
\end{aligned}
$$

where $\circ_h \in \{+, -, *\}$ and $a_h, b_h \in \mathbb{Q} \cup \{x, i\} \cup \{y_j \mid 1 \leq j \leq k + m\}$.

Definition. We define the class $P_{\text{for}(x,k),\text{if}(m)}$ of programs containing a for loop with an embedded if-then-else statement and a variable upper bound on the loop index as follows:

$$
\begin{aligned}
P_{\text{for}(x,k),\text{if}(m)} := \{p \mid p(x) := \; & y_j = a; \\
& \text{for } i = l \text{ to } x \text{ do} \\
& \quad\quad p_1(x); \\
& \quad\quad \text{if } b \circ c \text{ then} \\
& \quad\quad\quad q_1(x) \\
& \quad\quad \text{else} \\
& \quad\quad\quad q_2(x); \\
& \quad\quad p_2(x); \\
& \text{output}(y_{k+m})\},
\end{aligned}
$$

$$\text{where } 1 \leq j \leq k + m \text{ and } a \in \{x\} \cup \mathbb{Q} \text{ and } l \in Z$$

$$\text{and } p_1 \in P'_{k_1}, p_2 \in P'_{k_2}, q_i \in P'_m \text{ and } k_1 + k_2 = k$$

$$\text{and } \circ \in \{=, \neq, <, >, \leq, \geq\} \text{ and } b \in \mathbb{Q} \cup \{x, i\} \cup \{y_j \mid j \leq k_1\} \text{ and } c \in \mathbb{Q}.$$

The classes $P_{\text{for}(n,k),\text{if}(m)}$ and $P_{\text{for}(x,k),\text{if}(m)}$ contain programs with one for loop with an embedded if-then-else statement. There is one initialization line before the loop, m lines of straight line computing code inside each branch of the if-then-else statement, and k lines of code inside the loop but outside the if-then-else statement. As in the class $P^{n,k}_{\text{for}}$ the righthand side of each line inside the loop is allowed to use any y_i value, for i between 1 and $k + m$. In order to avoid the problem of uninitialized variables, we assume that all variables y_i are initialized to 0 before the program is executed. In class $P_{\text{for}(n,k),\text{if}(m)}$ the loop index is bounded above and below by constants and the number of times through the loop is no more than n, but in class $P_{\text{for}(x,k),\text{if}(m)}$ the loop index is bounded above by the input x, so the number of times through the loop is unbounded. If the input x is not an integer, then the upper bound on the loop index for a program in $P_{\text{for}(x,k),\text{if}(m)}$ is $\lfloor x \rfloor$.

If we do not make use of the for loop in a program from $P_{\text{for}(n,k),\text{if}(m)}$ by simply recomputing the same function each time through the loop, then we can apply Corollary 5.1 and Theorem 4.1 and obtain a lower bound of $2m + 1$ on the VCP-dimension of the class. Such a lower bound would be obtained by a subclass of programs that use only the lines of code in the if-then-else statement to compute a function. Similarly, we can apply just Theorem 4.1 and obtain a lower bound of $k + m + 1$ on the VCP-dimension of the class by using a subclass of programs that have a boolean statement that is always true. These programs would use the m lines of code in the true clause of the if-then-else statement and the k lines of code outside the if-then-else statement to compute a function with straight-line code.

It is also possible to obtain a lower bound on VCP-dimension that is a function of n, the number of times that the for loop is executed. To do this we use a class of programs that are syntactically the same as those in $P_{\mathrm{for}(n,k),\mathrm{if}(m)}$ except that two initialization lines are permitted before the for loop. The following result gives a lower bound on the VCP-dimension of this class.

Theorem 6.2 *Let class $P_{\mathrm{for}(2,n,k),\mathrm{if}(m)}$ contain programs that are syntactically equivalent to those in $P_{\mathrm{for}(n,k),\mathrm{if}(m)}$ except that they contain two initialization lines of code before the for loop. Then for $k \geq 1, m \geq 4$, VCP-dimension($P_{\mathrm{for}(2,n,k),\mathrm{if}(m)}$) $\geq n$.*

Proof. The proof techniques we use are similar to those in the proof of Theorem 4.5 We demonstrate a subclass of $P_{\mathrm{for}(2,n,1),\mathrm{if}(4)}$ that can shatter the set $\{1, 2, \ldots, n\}$. Since dummy lines can always be added to a program, this shows that VCP-dimension($P_{\mathrm{for}(2,n,k),\mathrm{if}(m)}$) $\geq n$ for any $k \geq 1$ and $m \geq 4$.

$$
\begin{array}{l}
p_j(x) := y_2 = a_j; \\
\qquad\quad y_5 = 1; \\
\qquad\quad \text{for } i = 1 \text{ to } n \text{ do} \\
\qquad\qquad y_1 = 2 * y_2; \\
\qquad\qquad \text{if } y_1 \geq 1 \text{ then} \\
\qquad\qquad\quad y_2 = y_1 - 1; \\
\qquad\qquad\quad y_3 = x - i; \\
\qquad\qquad\quad y_4 = y_3 * y_3; \\
\qquad\qquad\quad y_5 = y_5 * y_4; \\
\qquad\qquad \text{else} \\
\qquad\qquad\quad y_2 = y_1; \\
\qquad\qquad\quad y_3 = 0; \\
\qquad\qquad\quad y_4 = 0; \\
\qquad\qquad\quad y_5 = y_5 * 1; \\
\qquad\quad \text{output}(y_5)
\end{array}
$$

The set $\{1, 2, \ldots, n\}$ can be shattered by the 2^n programs of the form shown above, where $a_j = j * 2^{-n}$ for $0 \leq j < 2^n$. The bit representation of each a_j represents a different subset of $\{1, 2, \ldots, n\}$. When an integer x between 1 and n is input to program p_j, p_j determines the x^{th} bit to the right of the decimal point in a_j and outputs a positive number if it is 0 or outputs 0 if it is 1. Thus the program p_j obtains the subset $\{x_1, \ldots, x_k\}$ of $\{1, 2, \ldots n\}$, where for $1 \leq i \leq k$ the x_i^{th} bit to the right of the decimal point in a_j is 0 and the rest of the first n bits to the right of the decimal point are 1.

The i^{th} iteration through the for loop extracts the i^{th} bit to the right of the decimal point in a_j. If this bit is one, then the factor $(x-i)^2$ is multiplied to y_5, otherwise y_5 is not changed. If the x^{th} bit to the right of the decimal point in a_j is 0, then on input x program p_j multiplies a number of positive factors together to get the final value of y_5, so $p_j(x) > 0$. If the x^{th} bit to the right of the decimal point in a_j is 1, then on input x the x^{th} iteration through the for loop multiplies the factor $(x-x)^2 = 0$ to y_5, so $p_j(x) = 0$. Therefore the subset of $\{1, 2, \ldots n\}$ represented by the zeros in the first n bit positions to the right of the decimal point in a_j is obtained by program p_j. $\square$

We get the following bounds on the VCP-dimension of the classes $P_{\text{for}(n,k),\text{if}(m)}$ and $P_{\text{for}(x,k),\text{if}(m)}$.

Theorem 6.3 *VCP-dimension*$(P_{\text{for}(n,k),\text{if}(m)}) = O(n(k+m)^2 \log(k+m))$.

Proof. We show that any program in $P_{\text{for}(n,k),\text{if}(m)}$ can be represented by $O(k+m)$ real values, and we describe an algorithm with $O(n(k+m)\log(k+m))$ runtime that can determine for any $p \in P_{\text{for}(n,k),\text{if}(m)}$ and $x \in \mathbb{Q}$ whether $p(x) > 0$. Then by applying Theorem 4.3 we obtain the desired bound.

Any program in $P_{\text{for}(n,k),\text{if}(m)}$ can be represented by $5(k+2m)+10$ real values. Parameters $p_{1,i}, \ldots, p_{k,i}$, where $1 \leq i \leq 5$, encode the syntax of the k lines of code inside the for loop but outside the if-then-else statement, parameters $p_{k+1,i}, \ldots, p_{k+m,i}$, where $1 \leq i \leq 5$, encode the syntax of the m lines of code inside the true clause of the if-then-else statement, and parameters $p_{k+m+1,i}, \ldots, p_{k+2m,i}$, where $1 \leq i \leq 5$, encode the syntax of the m lines of code inside the false clause of the if-then-else statement. Three additional parameters encode the initialization line $y_j = a$, four parameters encode the boolean clause $b \circ c$, and three parameters store k_1, l and u.

The algorithm to evaluate $p(x)$ for any program $p \in P_{\text{for}(n,k),\text{if}(m)}$ and any $x \in \mathbb{Q}$ first evaluates the initialization line and then enters a loop to evaluate the body of the for loop. After evaluating each line of code, it checks whether k_1 lines have been evaluated and, if so, jumps to a place that evaluates the if-then-else statement. Next it evaluates the remaining lines of code outside the if-then-else statement. Since the binary search to determine the operands for each line of code only needs to search from -2 to $k+m$, the algorithm takes $O(\log(k+m))$ time to evaluate each line of code. Each pass through the loop evaluates $k+m$ lines of code, and the loop is executed at most n times, so the total runtime is $O(n(k+m)\log(k+m))$. $\square$

The following code illustrates part of the algorithm from the proof of Theorem 6.3:

```
init:      code for y_j = a
           i = l
loop:      j = 1
           if j > k_1 then goto l_0
           code for line y_1
           j = j + 1
           if j > k_1 then goto l_1
               ⋮
l_{k_1}:   code for boolean clause b ∘ c
           if boolean clause true then goto l_{k_1,t}
l_{k_1,f}: code for line y_{k_1+1} using p_{k+m+1,i}, 1 ≤ i ≤ 5
               ⋮
           code for line y_{k_1+m} using p_{k+2m,i}, 1 ≤ i ≤ 5
           goto l_{k_1,k_2}
               ⋮
l_{k_1,k_2}: code for line y_{k_1+m+1} using p_{k_1+1,i}, 1 ≤ i ≤ 5
               ⋮
           code for line y_{k+m} using p_{k,i}, 1 ≤ i ≤ 5
           i = i + 1
           if i ≤ u then goto loop
```

Using techniques similar to those in Theorems 4.5 and 6.2 we can show that the VCP-dimension of the class $P_{\text{for}(x,k),\text{if}(m)}$ is infinite.

Theorem 6.4 *For $k \geq 1, m \geq 2$, VCP-dimension($P_{\text{for}(x,k),\text{if}(m)}$) is infinite.*

Proof. We demonstrate a subclass of $P_{\text{for}(x,1),\text{if}(2)}$ that can shatter the set $\{1, 2, \ldots, d\}$ for any $d > 0$. Since dummy lines can always be added to a program, this shows that VCP-dimension($P_{\text{for}(x,k),\text{if}(m)}$) is infinite for any $k \geq 1$ and $m \geq 2$.

The set $\{1, 2, \ldots, d\}$ can be shattered by the 2^d programs of the following form, where $a_j = j * 2^{-d}$ for $0 \leq j < 2^d$. The bit representation of each a_j represents a different subset of $\{1, 2, \ldots, d\}$. When a positive integer x is input to program p_j, p_j extracts and outputs the x^{th} bit to the right of the decimal point in a_j. Thus the program p_j obtains the subset $\{x_1, \ldots, x_k\}$ where the x_i^{th} bit to the right of the decimal point in a_j is 1 for $1 \leq i \leq k$ and all other bits are 0.

```
p_j(x) := y_2 = a_j;
          for i = l to x do
              y_1 = 2 * y_2;
              if y_1 ≥ 1 then
                  y_2 = y_1 − 1;
                  y_3 = 1
              else
                  y_2 = y_1;
                  y_3 = 0;
          output(y_3)
```

Note that only the values of the constants a_j are dependent on d, so for any $d > 0$ a set of 2^d programs from $P_{\text{for}(x,1),\text{if}(2)}$ can be constructed to shatter $\{1, 2, \ldots, d\}$. □

Theorem 6.4 shows that even simple program classes can be complex from a testing point of view. This indicates that program constructs that are simple from a syntactic point of view may not necessarily be simple to test. In particular, if there is not an upper bound on the number of times that a loop can be executed, it may lead to programs that are not randomly approximately testable.

7 Conclusion

Determining the difficulty of testing a program is an important part of assessing the complexity of the program. Since exact testing of a program is usually impossible, it is reasonable to use an approach that determines the difficulty of approximately testing the program. We have done this by defining a measure of testing complexity known as VCP-dimension and applying this measure to classes of programs, each with the same syntactic structure. We have investigated the VCP-dimension of straight line code, if-then-else constructs, and for loops, as well as several combinations of these constructs.

As a complexity measure, VCP-dimension has the drawback that it is not easy to compute exactly for most program classes. However, using techniques such as those in [GJ93] it is often possible to derive upper bounds fairly easily, as we have done in this paper. Also, unlike other complexity measures, we have proven results that relate this measure to the number of random test points required to determine that a piece of software is approximately correct. Although VCP-dimension is not a ready-to-use complexity measure to apply to real, large scale programs, a theoretical investigation into testing complexity is valuable for gaining a better understanding of the nature of testing complexity, and theoretical results may eventually lead to more meaningful complexity measures that are easy to compute.

One future direction of this research would be to determine the VCP-dimension of other program constructs and combinations of constructs. Another direction would be to develop simple methods to approximate the VCP-dimension of pieces of software and to develop tools to automate the process of making these approximate computations.

8 Acknowledgements

The authors would like to thank Eduardo Sontag of Rutgers University for referring them to the paper by Goldberg and Jerrum and for pointing out the connection between VC-dimension and pseudo dimension. They would also like to thank him for many helpful discussions on VC-dimension and pseudo dimension. In addition, the authors would like to thank Gabor Lugosi for help in proving Lemma 3.1 and Theorem 3.6 and an anonymous referee for many helpful suggestions, including improvements on Theorems 4.2, 4.5 and 6.2 and the results in Section 5.

References

[ABIST94] Martin Anthony, Peter Bartlett, Yuval Ishai, and John Shawe-Taylor. Valid Generalisation from Approximate Interpolation. Technical Report NC-TR-94-011, Royal Holloway University of London, December 1994.

[ABIST95] Martin Anthony, Peter Bartlett, Yuval Ishai, and John Shawe-Taylor. Valid Generalisation from Approximate Interpolation. *Combinatorics, Probability and Computing*, 1995. to appear.

[Ass83] P. Assouad. Densite et Dimension. *Annals Inst. Fourier, Grenoble*, 33(3):233–282, 1983.

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*, chapter 7. Addison-Wesley Publishing Company, Reading, MA, 2 edition, 1988.

[BBK77] J.M. Barzdin, J.J. Bicevskis, and A.A. Kalninsh. Automatic Construction of Complete Sample System for Program Testing. In B. Gilchrist, editor, *Proceedings of IFIP Congress 1977*, pages 57–62, Toronto, August 8-12 1977. International Federation for Information Processing, North-Holland Publishing Company.

[BC76] Allan Borodin and Stephen Cook. On the Number of Additions to Compute Specific Polynomials. *SIAM Journal of Computing*, 5(1):146–157, March 1976.

[BEHW89] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred Warmuth. Learnability and the Vapnik-Chervonenkis Dimension. *Journal of the Associaton for Computing Machinery*, 36(4):929–965, 1989.

[BK89] Manuel Blum and Sampath Kannan. Designing Programs That Check Their Work. In *1989 Symposium on Theory of Computation*, pages 86–97, Seattle, Washington, May 1989.

[BLR90] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. In *Proceedings of the* 22^{nd} *ACM Symposium on Theory of Computing*, pages 73–83, Baltimore, MD, May 14-16 1990.

[BS87] Victor R. Basili and Richard W. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, December 1987.

[Bud81] Timothy A. Budd. Mutation Analysis: Ideas, Examples, Problems and Prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland Publishing Company, 1981.

[DMMP87] Richard A. DeMillo, W. Michael McCracken, R.J. Martin, and John F. Passafiume. *Software Testing and Evaluation*, chapter 1,2,3.1,3.3.3. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987. This book is an edited version of the report of the Software Test and Evaluation Project to the Office of the Secretary of Defense (Research and Engineering).

[DN84] Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, July 1984.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[GJ93] Paul Goldberg and Mark Jerrum. Bounding the Vapnik-Chervonenkis Dimension of Concept Classes Parameterized by Real Numbers. In *Proceedings of the Sixth ACM Workshop on Computational Learning Theory*, Santa Cruz, CA, July 26-28 1993. ACM Press.

[GLR+91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-Testing/Correcting for Polynomials and for Approximate Functions. In *Proceedings of the* 23^{rd} *ACM Symposium on Theory of Computing*, May 1991.

[Hal77] M.H. Halstead. *Elements of Software Science*. Elsevier-North Holland, New York, 1977.

[Ham89] Richard Hamlet. Theoretical Comparison of Testing Methods. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37, Key West, Florida, December 13-15 1989.

[Hau92] David Haussler. Decision Theoretic Generalizations of the PAC Model for Neural Net and Other Learning Applications. *Information and Computation*, 100(1):78–150, September 1992.

[Lip91] Richard J. Lipton. New Directions in Testing. In *Distributed Computing and Cryptography*, volume 2, pages 191–202. DIMACS Series on Discrete Mathematics and Theoretical Computer Science, 1991.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[MH81] Edward Miller and William E. Howden, editors. *Tutorial: Software Testing and Validation Techniques*. IEEE Computer Society Press, 2 edition, 1981.

[MS93] Angus Macintyre and Eduardo D. Sontag. Finiteness Results for Sigmoidal "Neural" Networks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 325–334, San Diego, California, May 16-18 1993.

[NP87] Deborah Nolan and David Pollard. U-Processes: Rates of Convergence. *The Annals of Statistics*, 15(2):780–799, 1987.

[Pol84] David Pollard. *Convergence of Stochastic Processes*. Springer Series in Statistics. Springer-Verlag, New York, 1984.

[Pol86] David Pollard. Rates of Uniform Almost-Sure Convergence for Empirical Processes Indexed by Unbounded Classes of Functions. manuscript, 1986.

[RS72] Edward M. Reingold and A. Ian Stocks. Simple Proofs of Lower Bounds for Polynomial Evaluation. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 21–30, Yorktown Heights, NY, March 20-22 1972. Plenum Press.

[Tia92] Jianhui Tian. *Understanding and Using Program Complexity to Improve Software Development*. PhD thesis, University of Maryland at College Park, College Park, MD, June 1992. Technical Report CS-TR-2921/UMIACS-TR-92-72.

[TZ92] Jianhui Tian and Marvin V. Zelkowitz. A Formal Program Complexity Model and Its Application. *Journal of Systems and Software*, 17(3):253–266, March 1992.

[VC71] V.N. Vapnik and A.Ya. Chervonenkis. On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory Probability and its Applications*, 16(2):264–280, 1971.

[WD81] R.S. Wenocur and R.M. Dudley. Some Special Vapnik-Chervonenkis Classes. *Discrete Mathematics*, 33:313–318, 1981.

[Wey88] Elaine J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[WHH79] M.R. Woodward, M.A. Hennell, and D. Hedley. A Measure of Control Flow Complexity in Program Text. *IEEE Transactions on Software Engineering*, SE-5(1):45–50, January 1979.