

Learning edge momentum: A new account of outcomes in CS1

Anthony Robins
anthony@cs.otago.ac.nz
Computer Science
The University of Otago
Dunedin, New Zealand

Abstract

Compared to other subjects the typical introductory programming (CS1) course has higher than usual rates of both failing and high grades, creating a characteristic bimodal grade distribution. In this paper I explore two possible explanations. The conventional explanation has been that learners naturally fall into populations of programmers and non-programmers. A review of decades of research, however, finds little or no evidence to support this account. I propose an alternative explanation, the learning edge momentum (LEM) effect. This hypothesis is introduced by way of a simulated model of grade distributions, then grounded in the psychological and educational literature. LEM operates such that success in acquiring one concept makes learning other closely linked concepts easier (whereas failure makes it harder). This interaction between the way that people learn and the tightly integrated nature of the concepts comprising a programming language creates an inherent structural bias in CS1 which drives students towards extreme outcomes.

Keywords: Learning to program; programming; CS1; grade distribution; CS1 distribution; bimodal; momentum; edge effects; learning edge momentum; LEM.

1 Introduction

A central focus of computer science education (CSEd) research is the teaching and learning of a first programming language. Research into novice programmers spans more than thirty years and has examined a wide range of issues, such as comparisons of novice and expert characteristics, the acquisition and use of both knowledge and strategies, the role of mental models of programs and “notional machines”, explorations of the generation and comprehension of programs, and so on. For reviews of this literature see Robins, Rountree & Rountree (2003), Pears *et al.* (2007).

Much of the motivation for this research stems from the fact that learning to program appears to be, for many people, a very difficult task. In introductory programming courses (which I will by convention refer to as “CS1”) failure rates are often high. One recent statement reflects widely reported experience: “between 30% and 60% of every university computer science department’s intake fail the first programming course” (Dehnadi & Bornat, 2006). The high failure rate is only one side of the story however. Paradoxically, typical CS1 courses also have unusually high rates of high grades as well. Consequently CS1 grade distributions, having fewer mid range results, are often described as bimodal:

“One of the most disconcerting aspects of teaching under-graduate courses in statistics and computer science is the appearance of a bimodal distribution of grades.” (Hudak & Anderson, 1990).

“A strongly bimodal distribution of marks in the first programming course is frequently reported anecdotally, and corresponds to our experience in several different academic institutions over a considerable period of time.” (Bornat, Dehnadi & Simon, 2008).

“Faculty data typically have shown a bimodal distribution of results for students undertaking introductory programming subjects with a high proportion of students receiving a high mark and a high proportion of students receiving a low or failing mark.” (Corney, 2009).

While this pattern appears to be typical of large CS1 courses with a general intake, smaller or more selective courses can of course show considerable variation. Bennedsen & Caspersen (2007) surveyed 67 institutions internationally and reported a huge variation in fail rates, from 0% to over 60%, as shown in Figure 1. Some of the

variation is correlated with class size, small classes (less than 30 students) had an average fail rate of 18%, large classes 31%. Further variation arises from the makeup of the student population and other factors. However, as a general rule, for large open entry CS1 papers it appears that failure rates of 30% to 60% as claimed by Dehnadi and Bornat are not uncommon. At the other end of the scale Forbes & Garcia (2007) surveyed 32 “top-rated” CS departments in America. As might be expected failure rates were low and A grade rates were high, but there was significant variation in the specifics of grade distributions (Forbes & Garcia, 2007, online resources).

Discussion in this paper will focus on the general case of CS1 courses that are open entry, usually with large numbers of students, and that are taught in typical tertiary institutions. By “CS1 distribution” I mean the typical bimodal distribution described above with high proportions of both fail and high grades.

The CS1 course at my own institution (COMP160 at the University of Otago) illustrates many of these issues. COMP160 is offered in two different formats. As a regular semester course it is taught over 13 weeks with a class size of 200 to 300. The grade distribution is similar each year, the average distribution for 2005 to 2008 is shown in Figure 2. The distribution is strongly bimodal, with a failure rate (D and E grades) of 48%¹. In its second format as a summer school course COMP160 is taught over 6 weeks with a class size of 20 to 40. The student population is different, consisting mostly of students who are repeating the course or who have gained accelerated entry from high school. In this format the course has a failure rate of just 20%. I will use the results of the regular large scale version of the course, as shown in Figure 2, as an illustrative example of the CS1 distribution.

What causes the CS1 distribution? It doesn't seem difficult to explain the large fail group. It can be argued that the material is very challenging and for most students it requires better than average study skills and habits to master it. In this context it is more difficult to explain the large high grade group however – it would seem difficult to argue that the material is simultaneously both hard and easy. The generally

¹ This includes 10% of “ghost students” who are technically enrolled but never attend, a common occurrence in our general first year courses, but not summer school.

accepted explanation, which implicitly or explicitly underlies a lot of current practice and research, has been succinctly stated as follows:

“All teachers of programming find that their results display a ‘double hump’. It is as if there are two populations: those who can, and those who cannot, each with its own independent bell curve.” (Dehnadi, 2006).

This appears to explain the facts. The “can” population generates the high end of the CS1 distribution, the “cannot” population generates the low / fail end. The question which naturally arises, however, is what is the factor that determines or characterises these two different populations? Section 2 of this paper reviews the extensive literature relating to the range of possible factors and predictors. I will argue that this literature has not identified convincing predictors of success, and has not successfully characterised two populations, and that it is in fact doubtful that these two distinct populations actually exist.

The main contribution of this paper is to propose an alternative explanation for the CS1 distribution. This learning edge momentum (LEM) hypothesis is introduced by way of a simulated model of grade distributions in Section 3, then grounded in the psychological and educational literature in Section 4. I suggest that LEM arises as a consequence of the interaction of two factors: the widely accepted principle that we learn at the edges of what we know; and the new claim that the concepts involved in a programming language are unusually tightly integrated. In short, successfully acquiring one concept makes learning other closely linked concepts easier, while failing makes further learning harder. This interaction between the way that people learn and the nature of the CS1 subject material creates an inherent structural bias which drives CS1 students towards extreme outcomes. Pedagogical implications and directions for further research are explored in Section 5.

2 Explaining programmers and non-programmers?

The assumption that there are two kinds of people, programmers and non-programmers underlies (either explicitly or implicitly) a lot of computing related practice and CSEd research. In this section I briefly explore the literature relating to attempts to understand the factors that predict success in learning to program, or to explain what it is that defines or characterises these two hypothetical groups.

2.1 Aptitude tests

The attempt to understand and manage high failure rates in learning programming, and the often associated idea that programming ability must be “innate”, are issues with a long history in the computing profession and in CSEd research:

“Katz (1962) administered several tests from the Army Classification Battery to 190 trainees in the Army’s Automatic Data Processing Programming course. He used these test results in an attempt to reduce the wasted training time and costs associated with the prevailing high attrition rate.” (Bauer, Mehrens & Vinsonhaler, 1968).

“One can conclude that programming ability (as measured in this study) may be much more innate than 'business training course spiels' would have one believe. Anyone cannot be a programmer...” (Newstead, 1975).

The use of aptitude tests to predict programming ability was commonplace in the early decades of the profession, the most popular being the “IBM Programmer Aptitude Test” (PAT). As early as the mid 1960s such tests were used by 68% of computing organisations surveyed in the United States and 73% in Canada (Dickmann & Lockwood, 1966). Other popular tests included the Computer Programmer Aptitude Battery (CPAB) and the Wolfe Programming Aptitude Tests (WPAT), as described for example by Pea & Kurland (1984). In the academic community there was an ACM Special Interest Group in Computer Personnel Research (SIGCPR), which published two “major journals”, *Computer Personnel* and the yearly *Proceedings of the Nth Annual Computer Personnel Research Conference* (Simpson, 1973).

Despite their widespread use, it was never clear that early programmer aptitude tests were actually effective:

“Ever since the 1950s, when the [PAT] was developed by IBM to help select programmer trainees, consistently modest correlations (at their best from 0.5 to 0.7, hence accounting for only a quarter to a half of the variance), and in many cases much lower, have existed between an individual's score on such a measure and his or her assessed programming skill.” (Pea & Kurland, 1984).

The predictions of programmer aptitude tests with respect to actual job performance were also poor (Mayer & Stalnaker, 1968; Bell, 1976). Although Curtis (1986) states that such tests had “already fallen into disfavor” by 1968, and the PAT was last distributed in 1973, there are many subsequent explorations of their use. Mazlack(1980) reported low predictive values for the PAT, and suggested that tests of this type should not be administered to university students because the results are unreliable. Further critiques can be found in Evans & Simkin (1989), and Subramanian & Joshi (1996). Lorenzen & Chang (2006) note that historically popular aptitude tests have low predictive value, and that IBM no longer endorses the PAT.

As the limitations of early tests became clear, various alternatives for predicting programming success were explored, including the development of more sophisticated tests (some focusing specifically on university level education), and the exploration of a broader range of variables such as demographic factors and high school SAT scores. Of this second generation of tests the Berger Aptitude for Programming Test (B-APT) (see Berger as cited in Mayer & Stalnaker (1968), Simpson (1973)), emerged as the most popular, with claims that it “obtained some of the highest validities to date, although these studies have not been reported in the archival literature” (Curtis, 1986).

Wileman, Konvalina & Stephens (1981) explored predictors of success (final exam score) in a beginning computer science course. Of the three demographic and five test based factors studied, four test based factors (reading comprehension, sequence completion, logical reasoning, and algorithmic execution) were found to be

significant. A stepwise multiple regression including all eight factors explained approximately 25% of the variability in the final exam scores. Werth (1986) reviewed and explored a wide range of factors, including demographic details, academic background (particularly mathematics), work experience and commitments, personality tests, cognitive development tests and cognitive style. Werth reported “modest correlations” and “conflicting results”, and noted that: “The models which have been produced are not very powerful; many students who do not fit the proposed criteria still do well in their computer science classes”. Similarly Evans & Simkin (1989) review the use of aptitude tests, demographic background, past high school achievement, and “general cognitive processes” (such as problem-solving strategies). They note that linear regression models (based on 54 variables) had “only limited predictive power”, and conclude that:

“... no single set of variables – demographic, behavioral, cognitive, or problem solving – dominated the others as a "best" set of predictors of student performance. Rather, the results suggest that several factors from all four areas may be useful in forecasting computer aptitude.” (Evans & Simkin, 1989).

Subramanian & Joshi (1996) explored the Computer Aptitude, Literacy & Interest Profile test (CALIP), which was specifically designed for evaluating novices, concluding that while some dimensions of the test were better than others, overall it “is not a good predictor of programming performance”. More recently Whittington, Bills & Lawrence (2003) note that: “Performance in high school coursework, SAT scores, AP courses, and even pre-testing can be poor predictors of success for individuals”.

From today’s perspective, the early close links between the professional and the academic exploration of programmer aptitude have not been maintained. The Special Interest Group in Computer Personnel Research (SIGCPR) is no longer active, and the new Special Interest Group in Computer Science Education (SIGCSE) has emerged. There is far less emphasis on the exploration of aptitude tests in the academic literature. Despite their historically mixed performance, however, such tests are still in use in the computing profession. Various organisations offer programming aptitude tests as a professional service, for example Psychometrics I.T. Tests states on its web pages:

“The B-APT has become the DP trainee selection instrument of choice at many Fortune 500 corporations, governmental agencies, and foreign companies. Organizations use the B-APT primarily to identify high aptitude candidates for programmer training.” (Psychometrics I.T. Tests, 2009)².

Tests used by large companies such as Microsoft and Google have attracted considerable popular attention. There are also a multitude of “free” online “tests”, and books and other resources devoted to advice on how to prepare for and pass such tests. Soh *et al.* (2005) describe the use of “placement tests” to stream students in some university computing departments.

2.2 Possible explanations

In this section I briefly review the kinds of factors which have been explored as possible predictors of or explanations for programming ability in the aptitude testing and general CSEd literatures.

Cognitive capacity

The most obvious explanation, and the assumption that underlies most of the huge effort devoted to aptitude tests, is that programming depends on some particular cognitive capacity or capacities which we can identify and measure. In order to directly account for the CS1 bimodal distribution, we would expect to find a bimodal distribution of this capacity – two populations, those that can and those that can’t.

The possibilities explored in the aptitude testing literature were many and varied. For the tests noted above, the final version of the IBM PAT consisted of letter series, figure analogies, and number series and arithmetical reasoning subparts (Pea & Kurland, 1984; Subramanian and Joshi, 1996). The CPAB consisted of verbal meaning, reasoning, letter series, mathematical reasoning, and flow chart

² For similar organisations see also for example (retrieved July 3rd, 2009):
<http://www.psychometrics-uk.co.uk/>
<http://pages.netaxis.ca/winrow/>

diagramming subparts (Pea & Kurland, 1984). The WPAT was designed to test accuracy, deductive and logical ability, the ability to reason with symbols, and reading comprehension, by requiring participants to manipulate numbers according to a complicated set of procedural instructions (Wolfe, 1969; Tukiainen & Mönkkönen, 2002). The CALIP focused on the detection of patterns, with subparts relating to mathematical patterns, logical patterns, and visual patterns (Subramanian & Joshi, 1996). The B-APT, while not assuming any programming background, requires participants to use an invented language to write short “programs” which explore concepts such as looping, incrementing and branching. The test used by Wileman, Konvalina & Stephens (1981) had five components: reading comprehension, alphabetic and numeric sequences, logical reasoning, algorithmic execution, and alphanumeric translation.

Clearly this range of tasks covers many broad cognitive capacities and factors often associated with intelligence and the attempt to measure IQ. These include verbal skills, mathematical skills, spatial skills, analogical reasoning and working memory capacity. As noted above, however, no clear pattern has emerged from the aptitude testing literature. These measures of various cognitive capacities have at best moderate predictive power, and certainly nothing like a bimodal distribution of capacity has been reported.

Of the aptitude (and demographic) factors explored, probably the most widely studied and intuitively appealing possible explanation for programming success is mathematical ability. Most (though not all) studies that explore it conclude that mathematical ability or background is one of the better predictors. Pea and Kurland, however, sound an important note of caution:

“To our knowledge, there is no evidence that any relationship exists between general math ability and computer programming skill, once general ability has been factored out. For example, in some of our own work we found that better Logo programmers were also high math achievers. However, these children also had generally high scores in English, social studies, and their other academic subjects as well. Thus, attributing their high performance in computer programming to their math ability ignores the relationship between math ability and general intelligence.” (Pea & Kurland, 1984).

I return to the issue of general intelligence in Section 2.3 below.

Cognitive development

Rather than a specific cognitive capacity, several authors have explored more general measures relating to “cognitive development”, such as Piaget’s Stages of Cognitive Development (Piaget, 1971a; Piaget, 1971b; Feldman, 2004), or Bloom’s Taxonomy of Educational Objectives (Bloom *et al.*, 1956). This approach seems to lend itself more naturally to a “two populations” bimodal explanation, as it can be argued that individuals have either reached a certain level of development or they have not.

Kurtz (1980) used his own test to classify students at three levels of intellectual development, and found that these were “strong predictors of poor and outstanding performance, respectively”, but that they were better predictors of performance on tests (accounting for 80% of variance) than programming (39%). Barker & Unger (1983) used a shortened version of the test on a much larger population of students and found its predictive power to be much weaker.

In a brief review of the literature White & Sivitanides (2002) conclude that reaching Piaget’s formal operational stage “is a required cognitive characteristic of people for learning procedural programming”, and claim that “the majority of adults and many college students fail to develop to full formal operational thinking skills.” Relating these claims explicitly to grade distributions the authors go on to suggest that:

“Some programming classes may have a bimodal distribution of students’ grades. The low mode may indicate Piaget’s concrete operation stage. The high mode may indicate Piaget’s formal operation stage. This is supported by [Hudak & Anderson (1990)]. That study showed formal operation level students did better than concrete level students in a Statistics course and an Introduction Computer Science course.” (White & Sivitanides, 2002).

In contrast, Bennedsen & Caspersen (2006) found no correlation between stage of cognitive development (particularly “abstraction ability”) and programming ability (final grade in CS1).

While not focused on predicting performance, similar topics have been explored by Scott (2003) in a study of assessment items classified according to Bloom's taxonomy. Scott notes that:

“Many of the students are able to answer the lower level questions and a much smaller percentage can answer the higher level categories. This results in a double bump in the frequency versus score graph.” (Scott, 2003).

The use of Bloom's taxonomy in CS1 teaching and assessment is also investigated in Lister (2000) and Lister & Leaney (2003).

Claims relating to developmental stages are difficult to formalise, test and refute. With respect to Piaget's four stages (the most influential of the theories) there is considerable ongoing debate within the psychological literature on the validity and utility of the framework, and in particular the nature of the “miraculous” transition between stages, see Feldman (2004) for an excellent overview. While there is some variation, as often stated the transition to the formal operational stage takes place roughly between the ages of 12 and (in various accounts) 16 or 18 years old, or later. In other words, most of the students tested in CS1 courses should have reached the formal operational stage. To attribute the high failure rates in CS1 to a failure to reach this stage is to suggest that the underlying theory, and the evidence from other fields that supports it, is wrong in its specification of the typical age range. Furthermore, it is not clear at the highest formal stage of development what the relationship is between individual differences in development and individual differences in IQ, and what exactly is measured by the different tests for these attributes. For discussions of the relationship between Piaget's stages and IQ see for example Sternberg (1982) or Cianciolo & Sternberg (2004).

Cognitive style

A wide range of other factors have been considered in various literature, described by terms including cognitive style, learning style, personality type and similar. I will briefly consider these under the general heading of “cognitive style”.

In an early review Cross (1970) noted that “personality tests have not been used very successfully for predicting work adjustment in the programming occupation, nor have aptitude tests been entirely satisfactory”. Cross used tests (“scales”) exploring programmers’ styles, preferences and values, finding that in many cases results were inconsistent between an original study and a replication study. Bush and Schkade, in their overview of the diverse results emerging from aptitude surveys, concluded that no ideal cognitive style or personality type had emerged (Bush & Schkade, 1985).

Inconclusive or contradictory results such as these are common. For example the well known Myers-Briggs Type Indicator, a questionnaire designed to measure psychological types in the ways in which people perceive the world and make decisions (Myers, 1980/1995), was found to be a poor predictor by Pocius (1991), but found to be useful by Whitley (1996). Bishop-Clark (1995) reviewed literature relating cognitive styles and personality traits concluding that results have been “been both scattered and difficult to interpret”, and suggesting that it may be important to relate such traits to specific stages of the programming process (problem representation, planning and design, coding, debugging). Woszczyński *et al.* (2004) explored learning styles described by Krause (2000), concluding that “intuitive thinkers” tended to perform better than “sensor feelers”, but finding “no other differences in performance between other paired profiles”.

Other studies have reached firmer conclusions, although in general the predictive power remains weak. Hudak & Anderson (1990) administered Kolb's Learning Style Inventory (Kolb, 1985) to 94 students, finding that learning style – in particular the absence of a reliance on the “concrete experiences” learning style – predicted success in introductory statistics and computer science courses. In a study of 211 CS1 students Allert concluded:

“Several learning style variables were correlated with outcome. Reflective and verbal learning style students achieved top grades more frequently and lower grades less frequently than their scale opposites (active and visual learners respectively).” (Allert, 2004).

White & Sivitanides (2002) explored the topic of “cognitive hemispheric style”,

where the left hemisphere of the brain has been found to be associated with language, rational, and objective judgments, and the right hemisphere with intuitive, subjective judgment, music and spatial skills. These authors concluded that “the literature has shown left hemispheric thinking style of learners as another characteristic necessary for success with procedural programming”.

Attitude and motivation

A further range of factors relating to attitude and motivation have also been explored. It is not simply the case that those who work hard pass CS1:

“On the other hand -- still contrary to popular belief, time spent on the course and working with other students correlated negatively with the dependent variables. [...] These results seem to indicate that though poorer students may spend much time and ask many questions of their instructors and fellow students, it won't improve their grade.” (Newstead, 1975).

Our collective experience with our CS1 course is similar. While completing assessed laboratory work is a good predictor of success, the amount of effort which goes in to making good progress is widely variable. Every year we see students who put in enormous amounts of work and are completely unsuccessful.

In a multinational study de Raadt *et al.* (2005) administered the Biggs revised two-factor Study Process Questionnaire R-SPQ-2F (Biggs, 2001) to 177 CS1 students. The questionnaire assesses the participant’s attitudes to study, and generates two main scores, deep approach (DA) and surface approach (SA). DA, representing the attitude of seeking a true understanding of material, is a score aggregated from subscales measuring “deep motive” and “deep strategy”. SA, representing the attitude of seeking only a superficial understanding (repeating facts and passing assessment with minimum engagement), is similarly aggregated from subscales measuring “surface motive” and “surface strategy”. This study found that DA had a significant but modest positive correlation with outcome (CS1 grade), and SA had a significant but modest negative correlation.

A number of recent studies have found consistent results relating to students’ self

reports of their expectations, attitudes or responses to their CS1 course. Rountree, Rountree & Robins (2002) explored a range of pre-course expectations and demographic factors, finding that the strongest predictor of success was how well the student expected to do. In a study of non-major CS1 students Wiedenbeck (2005) explored measures of “self-efficacy” (self judgments of capability) taken at the start of the course (pre-self-efficacy) and during the course (post-self-efficacy), finding that the post measure was a significant predictor but the pre measure was not³. Wilson & Shrock (2001) examined twelve factors, finding three that were predictive, maths background and (post) comfort level with the course (which were positively correlated with success), and (post) attribution of success/failure to luck (negatively correlated). In a series of studies Bergin & Reilly (2005a, 2005b, 2006) found variously that a student’s (post) perception of their understanding of the material, “intrinsic motivation” (rather than extrinsic) and (post) “comfort level” were correlated with success. In a study covering multiple factors Ventura concluded:

“Cognitive and academic factors such as SAT scores and critical thinking ability offered little predictive value when compared to the other predictors of success. Student effort and comfort level were found to be the strongest predictors of success.” (Ventura, 2005).

Bennedsen and Caspersen explored the “emotional health and social well-being” of CS1 students in terms of five variables: perfectionism, self-esteem, coping tactics, affective states and optimism, concluding that:

“Surprisingly, we found no correlation between emotional health and social well-being on the one hand and success in computer science as indicated by course grades on the other. However, in most of the courses, the students who pass have a statistically significant higher self-esteem than those who do not.” (Bennedsen & Caspersen, 2008).

Other and multiple factors

One occasional speculation relating to the high CS1 fail rate is demographic in nature. The suggestion is that CS1 gets a much broader student intake than similarly formal subjects (mathematics and sciences), because high school experience enables students

³ I will use “(post)” to indicate a measure taken after a substantial part of the course has passed.

to self select in other subjects, but not in computing related fields. However, considering the full history of decades of computing aptitude testing in industry, and the range of different kinds of programming courses (some targeting mature students or professionals), this explanation does not seem to be general enough in scope.

Furthermore, the picture arising from demographic factors is hardly compelling. As noted above, most (but not all) studies which explore mathematical background or ability find that it is a significant but modest predictor. Other demographic and background factors have even more mixed results – in a brief review Woszczyński, Haddad & Zgambo (2005a) note studies which have explored age, gender, ethnicity, marital status, GPA, mathematics background, science background, ACT/SAT math scores, ACT composite score, SAT verbal scores, high school rank, and previous computer experience. The same authors also note that:

“Although demographic variables have also been gathered, including gender and age, the results have been mixed, with some studies showing a relationship between demographic variables and performance, and other studies showing no statistically significant relationships.” (Woszczyński, Haddad & Zgambo, 2005b).

Allert (2004) found that, except for involvement with computer gaming (which was negatively correlated), background familiarity with computers and software was not highly correlated with CS1 outcome.

While much of the discussion above has been organised around highlighting single features such as test scores or specific cognitive attributes, it has long been recognised that a single factor alone was unlikely to account for success or failure. Mayer & Stalnaker (1968) present an early review covering the range of factors which had been explored at that time, see also Pea & Kurland (1984), Curtis (1986), Evans & Simkin (1989), Wilson & Shrock (2001), Woszczyński, Haddad & Zgambo (2005) and Ventura (2005). Studies employing multiple factors and various forms of regression analysis include Bateman (1973), Newstead (1975), Wileman, Konvalina & Stephens (1981), Cronan, Embry & White (1989), Evans & Simkin (1989), Bennedsen & Caspersen (2005) and Bergin & Reilly (2006). Rountree *et al.* (2004) employed a decision tree classifier to analyse a range of factors, finding various predictive rules

such as: “background is not science AND not in first year AND under 25 years old AND no prior mathematics”, which could be used to identify clusters of students with a (relatively) high probability of failure.

A recent trend in the CSEd literature is the recognition that it may be necessary to move beyond small scale local studies. It may be necessary to explore not only multiple factors, but multiple factors over multiple contexts:

“Research in the comparatively young field of Computer Science Education (CSEd) consists almost exclusively of small-scale local studies. Many individual studies are of high quality and present significant and useful results. Overall, however, it is probably fair to say that the field of CSEd lacks a foundation of established theory and methods, is characterised by isolated findings that are difficult to assemble into a coherent whole, and thus have varying impact on practice.” (Fincher *et al.*, 2005a).

Fincher *et al.* argue that large scale “Multi Institutional Multi National” (MIMN) studies offer several possible advantages, and go on to review several such studies in the recent CSEd literature. The advantages described include: increased statistical power, richer structure (where variation across institutions and cultures constitutes a “natural laboratory”), a broader exploration of background factors, and the possibilities of both improved methodology and hypothesis generation. One such recent MIMN study explored two possible predictors which are more abstract than those usually employed, the ability to articulate a search strategy (Fincher *et al.*, 2005b, Simon *et al.* 2006a, 2006b), and map drawing style (Fincher *et al.*, 2005b, Simon *et al.* 2006a, Tolhurst *et al.*, 2006), finding modest positive correlations with success in both cases.

2.3 Discussion

In reviewing the literature relating to predicting success in learning a first programming language, no clear result emerges. As other recent summaries put it:

“However, even with almost 40 years of study into the factors that predict success, researchers and educators alike have failed to agree on exactly which

variables actually predict student success.” (Woszczyński, Haddad & Zgambo, 2005b).

“Despite a great deal of research into teaching methods and student responses, there have been to date no strong predictors of success in learning to program.” (Bornat, Dehnadi & Simon, 2008).

Despite our failure to convincingly identify the predicting factors, the belief that such factors must exist seems to be as strong as ever. Similarly, despite our failure to convincingly account for the bimodal CS1 distribution, the underlying assumption still appears to be that the explanation must lie in some preexisting factors which divide the world into populations who can and can't learn to program. In the recently popular research on threshold concepts, for example, “pre-liminal variation” (variation which exists before the learning process begins) is regarded as the key to understanding why students may be effective or ineffective in the learning process (Meyer & Land, 2003; Rountree & Rountree, 2009).

When considering the partial predictors which have been fairly consistently identified, such as mathematical ability, two important issues do not appear to have received much attention. One of these is that there is little or no attempt (that I am aware of) to compare the factors associated with success in CS1 with those in other subjects. It seems obvious that factors such as mathematical ability, various personality attributes, attitudes to learning, and feelings of self confidence, will be moderately associated with success in a wide range of subjects. Why should any of these factors be specific to CS1 and the task of learning a first programming language? If they are not specific (or nearly specific) to CS1, then how can they form the basis of an explanation of the CS1 bimodal distribution?

The other seldom discussed issue, as alluded to in the discussion of mathematical ability in Section 2.2 above, is the extent to which cognitive factors associated with success in CS1 have any predictive power which is independent of intelligence (IQ). The literature relating to IQ is huge and diverse, with debate as to whether IQ should be thought of as single general factor “g”, a few major factors (such as verbal and spatial intelligence), or multiple factors (such as the logical, linguistic, spatial, musical, kinesthetic, naturalist, intrapersonal and interpersonal intelligences proposed

by Gardener (1993)). One of the most pervasive and general results, however, is that performance on one standard psychometric test is highly predictive of performance on other such tests, a phenomenon known as “the positive manifold” (Jensen, 1998) and sometimes used as an argument for the existence of g.

It is understandable that (apart from some of the early professional aptitude testing) full IQ tests do not appear to have been used in the CSEd literature. They take a long time to administer, and students for various reasons may not consent to participating. This does however leave very open the question of the explanatory power of the various cognitive factors which have been observed. This issue has been confronted directly by Pea and Kurland, and the earlier authors that they cite:

“Although most studies in which programming aptitude test scores correlated significantly with programming "success" (generally indicated by grades in industrial programming training courses or college programming courses) observed that "general intelligence" (when test scores were available) also correlated very highly with programming success, this does not seem to have moved the researchers to go further and ask whether the "programming aptitude" supposedly linked to programming skill constituted a specific aptitude factor above and beyond "general intelligence." We suspect that it may not. In fact, one survey (Mayer & Stalnaker, 1968) revealed that many companies use intelligence tests as their predictors of programmer success. In a general review of the computer personnel research presented to ACM Special Interest Group in Computer Personnel Research, Stalnaker says: "I think that if we have to have a very concise summary of our current knowledge, it is that the more intelligent person you can find, the better programmer you can probably get.” (Pea & Kurland, 1984).

From the review above the situation today appears to be unchanged. The cognitive factors which are partial predictors of success in CS1 may have no explanatory power which is independent of IQ. Furthermore, intelligence is at least roughly normally distributed in the population (there is some debate, but it is certainly not the case that the distribution defines two populations). Hence, while IQ certainly has some impact on success, it is not at all obvious how variations in IQ can simply account for the bimodal CS1 distribution.

This state of affairs leaves us without an obvious explanation for the pattern of learning outcomes in CS1, and thus creates something of a puzzle. The bimodal grade

distribution is a “big effect”. It is robust across several countries, several decades, many programming languages, and large numbers of individual teachers and students. A big, robust effect should have a big robust cause, and we should have found such a cause over decades of research. The factors so far examined, however, are moderate predictors at best, are almost certainly not specific to the task of learning to program, may well (in most cases) have no explanatory power that is distinct from IQ, and do not define two obvious populations. In short, this approach appears to offer little hope of a satisfactory explanation for the bimodal grade distribution observed in CS1.

It is possible that we have been looking for the explanation in the wrong place. I suggest that the “two populations” explanation is incorrect, and that we will not find the magic factor which distinguishes the programmer from the non-programmer, because it does not exist.

In the remainder of this paper I will argue that we might consider a very different kind of explanation for the pattern of learning outcomes in CS1, namely that an inherent structural bias, arising from the interaction between the way people learn and the nature of the subject material, acts to drive populations of students towards extreme outcomes. This alternative account will be introduced in the following Section 3 by way of an abstract model which explores some of the factors which could influence grade distributions. A preliminary psychological and educational interpretation of predictions arising from the model is presented in Section 4, with implications and further research outlined in the Section 5.

3 A simple grade distribution model

The purpose of this model is to provide a framework for examining the factors which produce distributions in the simplest possible terms. I will describe and interpret the model in language which takes the distributions as representing the grades of a population of students. In this abstract world it is possible to identify one simple effect which turns normal distributions into “anti-normal” bimodal distributions.

3.1 The model and its interpretation

The basic version of the grade distribution model M0 is shown in Figure 3. The outer loop repeats for a specified number of “runs” (in this paper fixed at 10,000). Each repetition / run generates a single score and increments a counter for that score, thus over all runs producing a distribution of scores. The score is generated by the inner loop, which repeats for a certain number of “chunks” (in this paper 10). Each repetition may or may not increment the value of the score, hence the score produced by the inner loop is an integer in the range 0 to 10 (inclusive).

To interpret this as a distribution of grades, each run can be thought of as a “student” going through a process and emerging with a score. The process (the inner loop) represents some course or programme of study made up of ten chunks, where chunks represent topics or concepts, or weeks work, or units of some kind. In this simple model the student achieves or does not achieve each chunk and gets one score increment for every chunk achieved, hence a final score of 0 to 10. Student performance for each chunk is represented by a random number. If the value (in the range 0 to 1) exceeds a threshold (for example 0.5) the chunk is considered to be achieved / passed and the value of the score increments.

Clearly this is a trivially simple model when compared to real students, real courses and the real world. The purpose of the model is just to explore the factors which produce grade distributions in the most abstract possible terms. With the parameter settings described above (10 chunks each with threshold = 0.5) M0

produces a normal distribution as shown in Figure 4(a). With threshold = 0.3 (each chunk is easier to pass) it produces a normal distribution skewed towards higher grades as in Figure 4(b). The lowered threshold could be interpreted as representing various educational phenomena, for example a skilled teacher facilitating student learning so that each chunk is easier to pass, or an institutional decision to mark more leniently. Having illustrated this effect all remaining simulations use threshold = 0.5.

3.2 The assumption of independence

The model embodies assumptions both explicit and implicit. In this section I want to explore a variation on just one of the assumptions, namely that performance on each chunk is independent of the other chunks. This assumption is currently implicit in the fact that nothing (in the current version of the model M0) represents any kind of interaction between chunks.

If chunks are not independent, how might they interact? One obvious possibility is that success or failure one chunk could have an impact on success or failure on the next. The impact of such an effect can be explored by introducing a slight variation into the model. The model version M1 is configured such that that passing one chunk lowers the threshold for passing the next, and (symmetrically) that failing a chunk raises the threshold for the next one. M1 is constructed by replacing the kernel of M0 (the contents of the inner loop, currently a single “if” statement) with the new kernel shown below:

```
if (rand.nextDouble() > threshold) {
    score++;
    threshold -= offset;
} else {
    threshold += offset;
}
```

where the variable offset is declared and initialised with the set up variables above the loops. Performance of M1 with offset = 0.0 is equivalent to M0. Performance for offset = 0.03 is shown in Figure 4(c). Creating a degree of dependence between the chunks has – compared with the basic distribution in Figure 4(a) – slightly “squashed” the distribution, lowering the height of the central maximum and raising the height of

the extremes. With $\text{offset} = 0.06$ this effect is even more pronounced as shown in 4(d). With $\text{offset} = 0.18$ scores have been pushed almost completely to the extremes as shown in 4(e). I will call this an “anti-normal” distribution.

3.3 The momentum effect

The dependencies between chunks created by the offset parameter in M1 result in a kind of learning *momentum effect* (the cumulative effects of previous learning / chunks passed or not). Success on one chunk makes the next one easier, failure makes the next one harder. As the dependence between chunks which causes the momentum increases in strength (in M1 the value off the offset) it turns a normal distribution into a bimodal / anti-normal distribution.

The momentum effect has been presented as implemented in M1 because it is the simplest possible variation on M0, but in educational terms it is unrealistic. In particular, there are no limits to the impact of repeated offsets on the threshold, so that if the threshold falls below 0.0 then every subsequent chunk (for that run) will be passed, and likewise if it rises above 1.0 every subsequent chunk will be failed. In an educational context it hardly seems plausible that the threshold can fall infinitely low, and there are various modifications which could be implemented to prevent it. Again presenting the simplest possibility, as a minor variation within M1 when can alter the kernel shown above by, after the statement “`threshold -= offset;`”, adding a further statement “`if (threshold < lowerBound) threshold = lowerBound;`” (where `lowerBound` is declared and initialised with the other set up variables). In other words, by replacing the simple offset increment with an “offset function” we can set a lower bound for the threshold.

Figure 4(f) shows the distribution produced with $\text{offset} = 0.18$ and $\text{lowerBound} = 0.14$. Obviously an equivalent change could be made to produce a symmetrical distribution showing the same limit effect for the low end of the grade distribution. However, as discussed in Section 4 below, the distribution as shown in Figure 4(f) looks remarkably like the distribution of grades in a typical introductory programming

course (for example Figure 2). I will refer to it as a “simulated CS1 distribution”.

3.4 Richer models

The momentum effect, arising from varying one assumption in a simple grade distribution model, turns normal distributions into anti-normal distributions, and can easily be tuned to produce a simulated CS1 distribution. From my preliminary exploration it appears that any reasonable function which implements a form of momentum (such as a “Zeno” offset function which always halves the distance from the current threshold to a lower or upper bound) produces similar outcomes. In short, the general principle of the momentum effect is robust.

There are many ways in which the model can be varied and extended to make it more realistic and provide a framework for exploring other phenomena. As already described, different parameter settings will produce different distributions. For M1 different offset values for the success and fail conditions (of the kernel if..else) could be used so that the effects of achieving or not achieving a chunk are not necessarily symmetrical. The addition or subtraction of fixed offset values can be replaced with more complex offset functions, such as the lower bounded offset function described above. More realistic offset functions could be implemented, though as noted above the momentum effect does not appear to rely on any particular implementation.

Further extending the model allows a richer range of issues to be explored. For example, in the current M1 there is no way to model chunks which have different thresholds, because the threshold is being used to represent the cumulative effects of previous learning (chunks acquired / passed or not). Version M2 has been constructed so as to represent momentum as a separate named term. Momentum can be added to terms representing student performance (in the model so far the random number generator) or equivalently it can be subtracted from the threshold. The code shown below illustrates the former. Specifically, M2 can be constructed by replacing the kernel of M0 with the version shown below:

```
if (rand.nextDouble() + momentum > threshold) {
    score++;
    momentum += offset;
} else {
    momentum -= offset;
}
```

where the variable `offset` is declared and initialised with the set up variables above the loops, and `momentum` is declared and initialised to 0.0 inside the outer loop with `score`. Note that in M2 we are now adding offsets in the success condition to represent positive momentum (instead of subtracting them as in M1), and vice versa in the fail condition. Positive momentum in M2 is equivalent to the lowered threshold of M1.

For the same parameters M2 is equivalent to M1. With `offset = 0.0` it produces the distribution in Figure 4(a). With `offset 0.18` it produces Figure 4(e). With `offset 0.18` and an upper bound on momentum of 0.36 (equivalent to a lower bound on threshold of 0.14 in M1) M2 produces Figure 4(f). Java code for this version of M2 is shown in Figure 5. However, while M2 can be equivalent to M1, it also creates a more flexible framework. With further modification the threshold is now free to vary for each chunk, and further terms could be added to the threshold that represent other aspects of the learning context. Asymmetric or different offset values or bounds could be set for different chunks or for different students, and further terms could be also be added to represent other aspects of student performance. Similarly an overall momentum function which is more powerful than the current simple random walk (perhaps a non-stationary Markov chain) might allow for richer or more accurate predictions.

3.5 Summary

The model described above allows us to explore some of the factors that affect simulated grade distributions. It seems unlikely that the specific parameter settings or implementation details are of any particular significance, rather I am interested in the general effects, principles and predictions that it enables us to investigate.

When chunks are independent the model generates normal distributions. Chunks can be made dependent, so that success in acquiring one chunk tends to make acquiring the next chunk easier, and failure makes the next one harder. This results in a self-reinforcing momentum effect in learning. Weak momentum flattens the normal distribution. Strong momentum drives grades to the extremes, producing an “anti-normal” distribution. In short, varying one assumption in the model introduces a momentum effect, and a strong momentum effect can easily be tuned to produce a simulated bimodal CS1 distribution.

4 A psychological and educational account

Producing a simulated grade distribution is all well and good as an abstract exercise, but to make the case that this has anything useful to say about the real world the effects illustrated by the model must be able to be interpreted in terms of real learning mechanisms.

In this section I will relate the simulated momentum effect to known psychological phenomena, and attempt to thus provide a plausible educational account. The main issues to be addressed are: (a) what does dependency between concepts created by the offset term in the model correspond to in educational terms; (b) how would such dependency give rise to momentum (both positive and negative) in learning; and (c) why is the effect of learning momentum so pronounced in CS1? I will argue that all that is needed to answer these questions is to simply take seriously two factors that are already well known and well accepted: firstly that the concepts involved in programming languages are tightly integrated; and secondly that we learn at the edges of what we already know. These factors combine to create learning edge momentum as novices learn to program, the real world equivalent of the momentum effect in the model.

4.1 Programming concepts are tightly integrated

I believe that there would be widespread agreement with the claim that the concepts involved in a programming language are well defined and densely connected (possible methods of substantiating this claim are discussed in Section 5.2 below). They are well defined because the elements that make up a programming language (almost always) have a precise syntax and semantics, these are not ambiguous or a matter of interpretation. They are densely connected because the concepts form an (almost) closed and highly self-referential system. It is very difficult to describe or understand one concept / language element (such as a for loop) independent of describing or understanding many others (flow of control, statements, conditions, boolean expressions, values, operators), which themselves involve many other

concepts, and so on. In short, a programming language is an example of a domain of *tightly integrated concepts*.

We can see the consequences of this tight integration in the various features of the pedagogy of the field. There is no agreement among CS educators on the correct order in which to teach topics in a given language. Even the very basic choice of “objects early” or “objects late” is still a matter of active debate. It is seldom that two textbooks on a language have the same ordering of subject material. There is no general agreement on the “right place to start” teaching a language, because every topic seems to depend on something else. Even the simplest Java program, for example, necessarily involves so many concepts (“public static void main (String [] args)”) that it cannot be fully explained until well into a course of study. The influential ACM curriculum guidelines openly acknowledge this state of confusion:

“Throughout the history of computer science education, the structure of the introductory computer science course has been the subject of intense debate. Many strategies have been proposed over the years, most of which have strong proponents and equally strong detractors. Like the problem of selecting an implementation language, recommending a strategy for the introductory year of a computer science curriculum all too often takes on the character of a religious war that generates far more heat than light.

In the interest of promoting peace among the warring factions, the CC2001 Task Force has chosen not to recommend any single approach. The truth is that no ideal strategy has yet been found, and that every approach has strengths and weaknesses.” (ACM, 2001).

The reason that these debates persist is that there are no right answers to the questions. There is, I suggest, no right place to start, and no correct ordering of topics, because a programming language is a domain of tightly integrated concepts, where almost every concept depends on many others.

4.2 We learn at the edges of what we know

The second major factor to be considered in this account is that we learn at the edges of what we already know. Almost by definition, understanding (on a short time scale)

and learning (over a longer period) depends on fitting new material into the context of existing knowledge. This basic feature of human learning and its consequences have been explored in the literature of several different fields.

Cognitive psychologists, for example, have shown that new information is stored, retrained and retrieved most effectively when it is integrated into existing knowledge. The richer and more elaborate the links between new and old knowledge the more effective learning appears to be. Thus effective learning depends on giving new information a “meaningful interpretation”, processing it in a “deep and meaningful way”, or elaborating / embellishing it with further meaningful details (Anderson, 2005). Note that elaborative processing is typically most effective when the elaborations are generated by the learner (rather than provided), and when they are related to or in some way constrain the new material (Reder, 1982; Anderson, 2005).

The active role of existing knowledge in assisting new learning is highlighted by literatures relating to analogy in cognition and transfer in learning and skill acquisition. Analogical reasoning is generally taken to involve the transfer of structural information from a known domain (the “source” or “base” domain) to the new domain to be explained or understood (the “target” domain) (Vosniadou & Ortony, 1989). Clearly the source domain must already exist in memory, with a known structure, in order to be able to evaluate and select source analogues, and to apply them to the target. “Analogy, in its most general sense, is [the] ability to think about relational patterns” (Holyoak, Gentner & Kokinov, 2001), it lies at “the core of cognition” (Hofstadter, 2001).

The literature on transfer in learning, particularly skill transfer, explores related concepts. Much of this work focuses on practical and vocational issues relating to training, performance and the factors involved in achieving good transfer from a training task to a real world situation (Robins, 1996). However transfer in general refers to any use of past learning when learning something new, and is “the very foundation of learning, thinking and problem solving” (Haskell, 2001). As further discussed by Haskell: “When learning is viewed as transfer, the primary factor influencing learning is the knowledge the individual brings into and uses in the

particular learning situation” (Voss, 1977).

The dependence of new learning and new capabilities on old is also stressed, albeit at a very different time scale, in the developmental literature. The whole concept of a sequence of stages of development (as briefly discussed above) is built on the assumption that cognition at some given level depends on, and cannot develop without, the capabilities of prior levels. Several influential ideas arose from the work of early psychologist Lev Vygotsky (for an overview see Rieber & Robinson (2004)). Vygotsky noted that there was a distance between children’s developmental level as determined by their independent performance, and their potential level as determined by performance when assisted. Vygotsky called this space, within which children are capable of learning and making progress, the “zone of proximal development” (ZPD). The ZPD is generally assumed to be the most effective level to target instruction and assistance, and Vygotsky’s prediction that teaching input would mostly occur at the edge of the ZPD was supported by McNaughton & Leyland (1990).

The educational literature combines cognitive, developmental, practical and social perspectives. Once again the theme of existing knowledge forming the context for new understanding and learning appears in many forms. Bereiter and Scardamalia, for example (influenced by Vygotsky and the ZPD) explore concepts such as “progressive problem solving” and “working at the edge of one’s competence” (Bereiter & Scardamalia, 1993). Bereiter (1985) raises the issue of “how progress toward higher levels of complexity and organization is possible without there already being some ladder or rope to climb on”, and explores possible mechanisms of “bootstrapping” which might address this. While bootstrapping is a general concept, some accounts specifically set out mechanisms for integrating new learning:

“We can conceptualize such a bootstrapping process as a series of *local repairs* of a knowledge structure. Local repairs require simple mechanisms such as adding links, deleting links, reattaching links, and so forth. The critical condition for local repairs is that the student recognizes that the repairs are needed, by reflecting on the differences between his or her existing knowledge and new knowledge.” (Chi & Ohlsson, 2005).

Other authors have focused on the ladder eschewed by Bereiter. The concept of

“scaffolding” (also heavily influenced by Vygotsky) emphasises the role of the teacher in learning, and the dynamic process of providing support for learning where it is needed and withdrawing it as learning is secured. Effective scaffolding provides “support at the edge of a child’s competence” (Gaskins *et al.*, 1997). Finally, “Constructivism” the broad and influential educational theory of knowledge arising from the work of Piaget, stresses that as humans we construct our own understanding of the world, generating knowledge and meaning from our experiences, and that we learn by integrating new knowledge into knowledge that we already have.

4.3 Learning edge momentum

As just explored, a wide range of converging evidence stresses the fundamental importance of the *learning edge*, the boundaries of existing knowledge which form the context into which newly learned concepts (information, understanding and skills) must be integrated. This fact is widely understood and accepted, and the basis of many sound pedagogical practices. Building on this foundation, I suggest that it is also useful to explore two emergent issues which appear to have received less attention, namely the ways in which the processes of the learning edge might unfold over time, and the ways in which they might interact with the structure of the target domain of new concepts to be learned.

In particular, I propose a new mechanism, learning edge momentum, which is that over time the effects of either successful or unsuccessful learning become self-reinforcing. When combined with the observation that a programming language is a domain of tightly integrated concepts, a new account of CS1 learning outcomes emerges.

The central claims of the learning edge momentum (LEM) hypothesis can be generally stated as follows. Given some target domain of new concepts to be learned, any successful learning makes it somewhat easier to acquire further related concepts from the domain, and unsuccessful learning makes it somewhat harder. Thus when learning a new domain the successful acquisition of concepts becomes self-

reinforcing, creating momentum towards a successful outcome for the domain as a whole, and similarly the failure to acquire concepts becomes self-reinforcing, creating momentum towards an unsuccessful outcome. This LEM effect varies in strength depending on the properties of the target domain. In particular, momentum strength varies in proportion to the extent to which the concepts in the domain are either independent or integrated. When the target domain consists of tightly integrated concepts the momentum effect (positive or negative) will be strong.

Why should learning one concept from a domain make learning further concepts easier, and why should the strength of this effect vary with the degree of integration in the domain? The answers to these questions will be found in the various fields which study the learning process, as reviewed above. The learning of a programming language, however, provides an interesting case study at one extreme end of the spectrum, and I propose the following initial suggestions. The multiple strong relationships between concepts in a programming language mean that a given subset of learned concepts has rich and well defined edges, creating strong constraints on where new concepts can be added. There is a “right place” for each new concept to fit⁴. Adding a new concept in its correct place creates richer edges and further constraints which (while leaving considerable flexibility in the order of the sequence) make adding the next concept easier, and so on. Furthermore, the multiple relationships between concepts afford multiple opportunities for the learner to process a newly learned concept deeply and elaborate it with meaningful connections that constrain and anchor it. Finally, the well defined structures and conceptual relationships within a programming language afford multiple possibilities for identifying patterns and base analogues that allow analogical reasoning to assist learning, and that facilitate transfer of learning between different parts of the domain⁵.

The equivalent negative side of this story is easy to tell. If we fail to acquire some concepts from a given domain we lack the structure within which to set further concepts, and learning becomes harder.

⁴ For example the concept of modulus division might be new to a learner, but there is only one “place” it can fit, it must behave like other arithmetic operators.

⁵ For example the concept of boolean expressions may be new, but they behave in a lot of ways like arithmetic expressions, with operators, operands, precedence and values.

By analogy, we can think of learning as adding pieces to an existing structure of some kind. For the target domain of a new programming language, this process is like adding pieces to a growing jigsaw puzzle. The edges of the puzzle pieces are sharply defined, there is only one correct place that each new piece can fit. Higher order structure in the puzzle's picture also provides constraints which assist the process. As the puzzle grows the emerging edges and structures provide so many constraints that adding each new piece becomes easier. If we have not successfully begun to build the puzzle, however, new pieces have no where to fit, and if they are lost or placed at random the task quickly becomes impossible. Where the target domain is less tightly integrated than a programming language the task is more akin to building a tower out of blocks. There is no single correct place for any given block, and there is probably considerable flexibility in the range of exact forms that an acceptable finished tower might take. Placing blocks does not necessarily get any easier, so that placing the last block may not be any easier or any harder than placing the first.

If the LEM hypothesis is correct then the explanation for the pattern of learning outcomes in introductory programming becomes clear. The CS1 distribution arises because of an inevitable interaction between the way that we learn and the nature of the material to be learned. The game is rigged, creating an inherent bias towards extreme outcomes.

4.4 Simulated and real effects

The principle of the LEM mechanism was introduced by way of a model in Section 3. That abstract account can now be anchored in the psychological and educational context described above. Using the language of the M2 version of the model, the chunks to be learned represent the concepts of the target domain of a given programming language. The dependency between chunks encoded by the offset term in the model represents the magnitude of the contribution that a given concept makes to learning other related concepts (or in other words represents the extent to which the

concepts in the target domain tend to be either independent or integrated). Passing some chunk results in a positive offset being added to the momentum term in the model, this represents the fact that successfully learning a given concept assists in learning related concepts. The momentum term itself in the model represents an overall measure of the success (or failure) of learning so far. An overall positive momentum term in the model increases the learner's performance (or equivalently lowers the threshold) for the next concept, representing a positive LEM effect in the cognitive processes of learning.

The larger the offset term (the more integrated the concepts in the target domain) the stronger the momentum effect, which flattens and broadens the normal distribution. An unusually high offset term (such as I have suggested applies in the tightly integrated domain of programming) produces an anti-normal distribution. Adding one further realistic constraint (that positive momentum cannot become indefinitely large) results in a simulated bimodal distribution which closely resembles the actual grade distributions observed in a typical CS1 course.

In all probability the model is only useful as a way of exploring very general principles. It is not likely that the specific implementation details and parameter values have any particular cognitive significance. The model as so far presented also contains many simplifying assumptions, such as for example that the offsets for success and failure are symmetrical (in real learning the momentum costs for failing to acquire a concept are probably higher than the momentum benefits of success). In short the model may crudely capture some of the effects that occur in human learning, but it tells us nothing about the subtle and complex processes that underlie those effects.

4.5 Discussion

If the arguments presented above are correct then the long underlying assumption that there are two populations, those who can program and those who cannot, is incorrect (there is no programmer gene!). CS1 students are much like any others, and they

succeed or fail for reasons which are idiosyncratic and complex, although in programming (as in other subjects) factors such as IQ, attitude and certain aspects of prior experience can affect the probability of success. The CS1 distribution arises not because our students are different, but because our subject is different (probably falling at one end of a natural continuum, as further discussed below).

In short, the pattern of CS1 learning outcomes arises because of an interaction between the learner and the learned. There is a big cause which gives rise to the big robust effect of the bimodal CS1 distribution. It turns out to be the only factor that is unique CS1 students – that they are all engaged in the task of learning a programming language. At one level this is of course stating the obvious, but it is also drawing attention to the impact of the nature of the material to be learned, rather than the nature of the learners. While as noted in Section 1 it would seem difficult to argue that a programming language is simultaneously both hard and easy to learn, the account presented here suggests that something close to this is in fact the truth. During the process of a typical CS1 course, a programming language does effectively *become* both harder and easier to learn for two different emerging groups of learners.

5 Discussion

5.1 Further work

Conceptual integration

The argument presented in this paper depends in general terms on there being a factor which is particular to programming languages, and which has an impact on learning such that a momentum effect emerges. I have argued that the relevant factor is the degree of integration of concepts in the subject material – the extent to which concepts are related to each other, are defined in terms of each other, or can only be understood properly or used properly if other related concepts are also understood. (In some sense the concepts of a programming language are highly parallel knowledge which gets forced through the serial pipe of a CS1 curriculum.) How might we go about investigating the claim that programming languages are tightly integrated knowledge⁶?

In Section 4.1 I argued that circumstantial evidence could be found in the lack of agreement among CS educators on the structure of the CS1 curriculum (compared to other disciplines with widely agreed paths through their curricula, or where the ordering of topics is not crucial). One approach to providing more concrete evidence is to explore measures generated by experts. Consider a measure collected from educators which attempts to characterise the material in the courses that they teach at first year tertiary level. One such useful characterisation would be a “mind map” (a loose form of semantic network popularised by Buzan & Buzan (1996), in simple terms a connected graph made up of concepts / nodes and links representing connections between concepts). Are programming mind maps more densely connected than those of other subjects? Do they contain more and tighter self-referential loops? Are there more crucial concept bottlenecks?

⁶ Clearly these suggestions relate to the concept of intrinsic cognitive load (Sweller, 1994), and exploring this connection may yield further insights and directions for future work.

A similar kind of measure, but more objective, might be generated using the statistical tools of corpus linguistics (see for example Manning & Schütze (1999)) to analyse text relating to different subjects, such as course text books or related general literature. Do proximity relationships between key concept terms vary across subjects? If so, are measures derived from programming related material indicative of more than usual conceptual density, complexity or dependency?

Similar attempts could be made to measure students' perceptions. Certainly there is plenty of anecdotal evidence relating to perceptions of complexity and negative momentum. Students in our CS1 course frequently comment that the material quickly "builds on itself" or "snowballs", and that falling behind means disaster as it is difficult to catch up. More speculatively, research reviewed above showing that growing confidence and levels of comfort with the subject material are correlated with success in CS1 may reflect students' reactions to the subjective experience of a positive momentum effect.

One of the further possibilities which could be explored with such methods is the extent to which the domain of programming concepts is, in addition to being tightly integrated internally, also isolated from other general knowledge. If programming is a comparatively self contained island (with few connections to the mainland) then this would also act to increase the significance of internal conceptual connections during learning.

Interdisciplinary comparison

In terms of supporting or refuting the proposed mechanism of LEM an experimental exploration could be based on comparing groups of subjects learning artificially constructed domains (with known and provably different degrees of conceptual integration). Whether the mechanism has any practical utility, however, will probably best be established by exploring the properties of grade distributions across different subject areas.

If it makes any sense to claim that a domain such as programming is characterised

by tightly integrated concepts, then of course other subject areas will have other varying degrees of integration. It follows that different subject areas must define a natural continuum from independent to tightly integrated subject material. In short, if CS1 is different, it is different not in kind, but only in degree. We would expect to find other subjects with bimodal or somewhat bimodal distributions, as the same interactions between subject material and basic mechanisms of learning would give rise to LEM effects of varying strength. Correlations between measures of conceptual integration and the strength or robustness of bimodal outcomes would support the LEM hypothesis.

Consider, for example, that as quoted in Section 1, Hudak & Anderson (1990) note that bimodal distributions are also observed in statistics courses. In my own institution a preliminary investigation of recent data suggests that other first year subjects (such as languages, music, and some science subjects) also have, at least on some occasions, bimodal grade distributions. How robust are these tendencies within a given subject area and across multiple institutions? What kind of continuum do bimodal subjects lie along? Is it, as suggested here, the continuum of the degree of conceptual integration, or is it something else?

Prior knowledge

Another topic which deserves further exploration is the role of prior programming language experience in CS1 outcomes. The LEM hypothesis would seem to suggest that any prior programming experience would be extremely useful, providing a richer base of existing knowledge within which to integrate and elaborate the new concepts encountered in CS1 – a flying head start of positive momentum. But the studies reviewed above are ambiguous. Some show a modest positive impact of prior experience, and some show none. Does this disprove the LEM hypothesis?

More detailed exploration of this issue is required. In most of the studies reviewed methods and results are reported only briefly, we do not know what kinds of questions were asked of participants, and consequently a lot of crucial information is missing. In order to explore this issue properly we need to know, for example, how recent the

prior experience was. Given that newly learned information is forgotten if it is not actively used and maintained, and especially that detail is lost quickly (Anderson, 2005), it may be the case that the time frame within which prior experience is useful to CS1 is short. (Anecdotally, my colleagues teaching CS2 sometimes remark that our students seem to have forgotten everything that they learned in CS1!). It would also be useful to know whether students' prior experience of programming was successful or unsuccessful. There is no reason to expect that previous unsuccessful attempts will be beneficial in CS1.

Details such as the paradigm of the prior language may also be important. Carey & Shepherd (1988) found “both positive and negative transfer effects” for programmers learning their first language in a new paradigm, and the damaging impact of incorrect prior assumptions in general have also been highlighted (see for example Bonar & Soloway (1989)). Reflecting the complexity of this issue, some authors have argued that it is easier to move from an object oriented language to a procedural one (Kölling, 1999), while some have argued the opposite (Hu, 2004). In a recent review Lister *et al.* (2006) found that there was “not a consensus in the research literature as to whether the objects first approach or the imperative approach is harder to learn”.

Our own previous study including prior experience as a factor (Rountree, Rountree & Robins, 2002) explored none of these issues, and I suspect that this is typical of broad studies of predictors in general. There has however been some research which has focused on prior experience specifically and in more depth (see for example Scholtz & Wiedenbeck (1993), Morrison & Newman (2001)). The LEM hypothesis suggests that it would be useful to build on such foundations and explore the impact of previous programming language experience in a richer and more systematic way.

Individual differences

In proposing the mechanism of LEM as a general account of the pattern of learning outcomes in CS1 I do not mean of course to suggest that it tells the whole story. While momentum compounds the effects of early success or failure and drives results

towards extremes, it doesn't tell us anything about why the crucial early success or failure occurs in the first place. The reasons for any given individual's success or failure remain as opaque as ever, and it remains the case that even seemingly major factors such as IQ have only a moderate predictive value. What starts one student down the path to a positive outcome, and the other down a negative path? The answer is not predetermined, or we would have found our two populations by now. The answer must lie in factors which are variable, idiosyncratic and situational. If a useful answer can be found amidst the complexity of human motivation and behaviour, it may require the large scale methodologies outlined by Fincher *et al.* (2005a) to find it.

Similar points have no doubt been made by many authors over many years. The following comments from Leither & Lewis (1978), for example, foreshadow some of the suggestions made in this paper, and outline some of the individual variation that is of particular interest:

“It is evident from teaching computer science to this large and diverse group of students that the varieties in their experience are numerous. The most experienced or quantitatively facile students learn quickly and easily; the most passive or negligent students discover too late how much they have to make up. In between are two groups which are more interesting from the standpoint of both pedagogy and cognitive psychology: Those who are mystified for weeks, and eventually make a quantum leap in understanding; and those who, despite extraordinary diligence and time-consuming effort, never succeed in making this transition. We do not support the somewhat common attitude that many individuals simply do not have the "aptitude" for learning computer science.” (Leither & Lewis, 1978).

Other topics

In general terms there are several open questions which may be worth further investigation. Can grade distribution modeling be used to explore other factors that shape the learning outcomes for various populations? What are the factors which influence the significance of the LEM effect across different individuals? Are there other significant effects arising from the interaction between the mechanisms of learning and the specific subject material of the target domain?

Finally, if the LEM hypothesis and the proposed importance of the learning mechanisms which underlie it are correct, can we teach students to use the tools of

positive momentum? By focusing pedagogy on highlighting the connections between concepts, ways of elaborating concepts, and various opportunities for analogy and transfer, perhaps by explicitly describing these mechanisms and their importance, can we achieve a substantially enhanced positive momentum effect and an improvement in CS1 learning outcomes? What are the further specifics of the processes that occur at the learning edge to integrate new and existing knowledge, and can understanding them provide any practical guidelines to educators or to learners? How do these cognitive effects interact with other factors such as personality and motivation? For example, are there complementary self-reinforcing reward and motivation effects that arise from the subjective experience of LEM?

5.2 Pedagogical implications

If the LEM hypothesis is correct, it implies that the very early stages of learning a programming language are critical to the outcome of the process. Once negative momentum is established it is very hard to overcome. Ideally positive momentum should be established right from the start.

In the context of CS1 this suggests a focus on the start of the course. From early analysis of our unpublished study in progress, we (S. Fincher, J. Tennenberg and I) believe that the first one or two weeks are crucial, and that certainly by the third week momentum effects are well under way. Laboratory attendance records as early as Week 1, and the quality of students' answers to diagnostic questions administered in weeks 3 and 5 ("What happens when you compile a program?" and "What is an object in a Java Program?") are highly predictive of final grade in the course.

What practical steps can be taken? Everything possible should be done to ensure that the start of the course runs smoothly, and to facilitate learning during this critical time. Extra support such as increased access to tutors / demonstrators should be provided. Students showing signs of disengagement (missing labs or tutorials, failing to submit work) should be followed up vigorously and immediately, as early as Week 1. Students could be told the reasons that early weeks are critical, as this "meta-knowledge" may increase engagement and motivation. Students should absolutely be encouraged to seek immediate help at the first sign of difficulty with the material –

keeping up with the flow of newly introduced concepts is vital.

Particular attention should be paid to the presentation of concepts and the connections between them in the early stages of CS1. Good pedagogical practice already links newly introduced concepts to the foundation of existing knowledge, but there may be further opportunities to understand and exploit the mechanisms of elaboration, analogy and transfer. As a community we may be able to arrive at some agreed best practice methods in this respect.

It is likely, however, that an even more significant intervention would be to break this overarching constraint of a single fixed flow (rate and path of progression through the curriculum). One size does not, and never will, fit all. There is absolutely no point in expecting a student to acquire a new layer of complex concepts when the foundation simply does not exist. This can only be addressed by introducing some flexibility into the delivery of the curriculum, so that students are more able to work and learn in ways that allow them to make sustainable progress.

While options for achieving flexibility are resource intensive, it is worth outlining at least some of the possibilities. CS1 could be offered in multiple streams which progress at different rates (some streams would of course cover more material than others). Students could self select streams, or move between them, possibly as advised by an early (Week 3?) diagnostic test of some kind. “Recovery streams” could be offered at certain points in the course for students who want to backtrack and revise. For maximum flexibility the whole concept of a stream (though which all students progress at the same rate) could be abandoned, with all learning being self paced. For example, some universities have offered CS courses consisting of just resource materials and a sequence of exercises. Passing a certain number of exercises (which students could attempt at any time) would result in a passing grade (at which point a student could choose to stop), and passing subsequent exercises would raise the grade.

I suspect that some disciplines learned these lessons in practical terms long ago, and the mastery model of learning (see for example Block (1971)) is the result. The

mastery model is a practical approach to teaching and learning based on the premise that learning is a function of time, not ability, and that given time (and appropriate learning conditions) all learners should be able to achieve. Learners do not progress to the next level until they have demonstrated mastery of the current one. The approach has its origins in the tradition of apprenticeships (and possibly musical training). It grew into a method which has been widely used in schools at varying times since the 1920s, and has also been used by some tertiary institutions. Although it is difficult to see how the mastery model can be adapted to large numbers in CS1, it may be worth further exploring this approach and its emphasis on self paced student learning. Perhaps the process of joining the programming community of practice is best regarded as an apprenticeship.

5.3 Summary

What causes the characteristic bimodal CS1 grade distribution, and ultimately what if anything can we do to smooth the path of learning and raise the pass rates for introductory programming? This paper has explored two possible explanations for the bimodal phenomenon.

The first explanation is embodied in the long standing assumption that there must be two different populations of people, those who can learn to program and those who cannot. Despite decades of extensive research, however, it has not proved possible to identify the factor or factors which characterise these two hypothetical groups. Aptitude tests are not reliable predictors of success, and have not identified any cognitive capacity which is especially significant for learning to program. Evidence relating to predictors such as educational or demographic background, cognitive development or style, or a range of other factors, is also ambiguous or weak. On balance some factors (such as mathematical background / ability, or self ratings of confidence / comfort as the course progresses) emerge as moderately correlated with success. But such attributes and correlations are far from the robust explanation that was sought. They are almost certainly not specific to the task of learning to program, they are hard to distinguish from the effect of IQ, and they do not define two obvious

populations of learners. In short, this approach has not so far offered any realistic hope of providing an explanation for the CS1 bimodal grade distribution. I suggest that the hypothesis that learners are inherently divided into two populations can now be regarded as highly doubtful, and that if it is to be supported then significantly new directions will need to be explored.

The main contribution of this paper has been to propose an alternative account of the pattern of CS1 outcomes, the learning edge momentum (LEM) effect. LEM is grounded in the principle, well accepted in the psychological and educational literature, that we learn at the edges of what we know. Building on this foundation, I suggest that it is also useful to explore two naturally emergent issues, namely the ways in which the processes of the learning edge might unfold over time, and the ways in which they might interact with the structure of the material to be learned.

The central claim of the LEM hypothesis is that, given some target domain of new concepts to be learned, any successful learning makes it somewhat easier to acquire further related concepts from the domain, and unsuccessful learning makes it somewhat harder. Depending on the strength of this effect, early success or failure to acquire concepts can become self-reinforcing, creating momentum towards extreme outcomes. The LEM effect varies in strength depending on the properties of the target domain, in particular it is dependant on the extent to which the concepts in the domain are either independent or integrated. I have further argued that the concepts involved in a programming language are unusually tightly integrated, thus resulting in a stronger than average LEM effect. In short, the interaction between the way people learn and the nature of the CS1 subject material creates an inherent structural bias which acts to drive populations of students towards extreme outcomes.

The failure to identify populations of programmers and non-programmers suggests that CS1 students are much like any others. Factors such as IQ and attitude have an impact, but ultimately students succeed or fail for reasons which are idiosyncratic, complex and situational. The LEM hypothesis suggests that the CS1 distribution arises not because our students are different, but because our subject material is different. In pedagogical terms this account highlights the crucial significance of the

early stages of learning, where progress in acquiring initial concepts begins to establish momentum towards successful or unsuccessful final outcomes. By understanding and adapting to this effect, by focusing resources on the early stages of CS1, by identifying and exploiting the mechanisms of successful learning, or ideally by adapting the way in which the curriculum is offered, it may be possible to significantly raise the rate of successful learning outcomes in CS1.

Acknowledgements

Thanks to colleagues Sally Fincher, Josh Tenenber, Patricia Haden, Michael Winikoff, Brendan McCane and Richard O'Keefe for very helpful comments and suggestions. Thanks to the University of Otago for study leave, and to the Computer Science Department at the University of Kent at Canterbury which hosted the visit during which the early stages of this research took place. Thanks also to the seminar audiences at The University of Kent and The University of Otago for general feedback and support.

REFERENCES

ACM (2001). *Computing curricula 2001 computer science final report (December 15, 2001)*. Retrieved July 3rd, 2009, from http://www.acm.org/education/education/education/curric_vols/cc2001.pdf

Allert, J. (2004). Learning Style and Factors Contributing to Success in an Introductory Computer Science Course. *Fourth IEEE International Conference on Advanced Learning Technologies (ICALT'04)*, 385–389.

Anderson, J.A. (2005). *Cognitive Psychology and its implications (6th ed.)*. NY: Worth Publishers.

Bauer, R., Mehrens, W.A. & Vinsonhaler, J.F. (1968). Predicting performance in a computer programming course. *Educational and Psychological Measurement*, 28, 1159–1164.

Barker, R.J. & Unger, E.A. (1983). A predictor for success in an introductory programming class based upon abstract reasoning development. *SIGCSE Bulletin*, 15 (1), 154–158.

Bateman C.R. (1973). Predicting performance in a basic computer course. *Proceedings of the Fifth Annual Meeting of the American Institute for Decision Sciences*. Atlanta, GO: AIDS Press, 130–133.

Bell, D. (1976). Programmer selection and programmer errors. *The Computer Journal*, 19, 202–206.

Bennedsen, J. & Caspersen, M.E. (2005). An investigation of potential success factors for an introductory model-driven programming course. *Proceedings of the First International Workshop on Computing Education Research (ICER '05)*. NY: ACM, 155-163.

- Bennedsen, J. & Caspersen, M.E. (2006) Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bulletin*, 38(2), 39-43.
- Bennedsen, J. & Caspersen, M.E. (2007). Failure rates in introductory programming. *SIGCSE Bulletin*, 39 (2), 32–36.
- Bennedsen, J. & Caspersen, M.E. (2008). Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science. *Computer Science Education*, 18 (1), 1–16.
- Bereiter, C. (1985). Toward a solution of the learning paradox. *Review of Educational Research*, 55, 201–226.
- Bereiter, C. & Scardamalia, M. (1993). *Surpassing ourselves: An inquiry into the nature and implications of expertise*. Chicago: Open Court.
- Bergin, S. & Reilly, R.G. (2005a). Programming: factors that influence success. *SIGCSE Bulletin*, 37 (1), 411–415.
- Bergin, S. & Reilly, R.G. (2005b). The influence of motivation and comfort level on learning to program. *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group (PPIG 05)*, 293–304.
- Bergin, S. & Reilly, R. (2006). Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, 16 (4), 303–323.
- Biggs, J.B. (2001). The revised two-factor Study Process Questionnaire: R-SPQ-2F, *British Journal of Educational Psychology*, 71, 133–149.
- Bishop-Clark, C. (1995). Cognitive style, personality, and computer programming. *Computers in Human Behavior*, 11 (2), 241–260.

Block, J. (1971). *Mastery learning: Theory and practice*. NY: Holt, Rinehart, & Winston.

Bloom, B., Englehart, M.D., Furst, E.J., Hill, W.H. & Krathwohl, D. (1956). *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. NY: Longmans.

Bonar, J. & Soloway, E. (1989). Preprogramming knowledge: a major source of misconceptions in novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 324–353). Hillsdale NJ: Lawrence Erlbaum.

Bornat, R., Dehnadi, S. & Simon (2008). Mental models, consistency and programming aptitude. *Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)*, 53–62.

Bush, C. M. & Schkade, L.L. (1985). In search of the perfect programmer. *Datamation*, 31 (6), 128 - 132

Buzan, T. & Buzan, B. (1996). *The Mind Map Book*. NY: Plume/Penguin Books.

Carey, T.T. & Shepherd, M.M. (1988). Towards empirical studies of programming in new paradigms. *Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science (CSC '88)*, 72–78.

Chi, M.T.H. & Ohlsson, S. (2005). Complex Declarative Learning. In K.J. Holyoak & R.G. Morrison (Eds.), *Cambridge Handbook of Thinking and Reasoning* (pp. 371–399). NY: Cambridge University Press.

Cianciolo, A.T. & Sternberg R.J. (2004). *Intelligence: A Brief History*. Oxford, UK: Blackwell Publishing.

Corney, M. (2009). Designing for engagement: Building IT systems. *ALTC First Year Experience Curriculum Design Symposium 2009*. Queensland, Australia: QUT Department of Teaching and Learning Support Services, 19–21.

Cronan, T.P., Embry, P.R. & White, S.D. (1989). Identifying factors that influence performance of non-computing majors in the business computer information systems course. *Journal of Research on Computing in Education*, 21 (4), 431–441.

Cross, E.M. (1970). The behavioral styles of computer programmers. *Proceedings of the Eighth Annual SIGCPR Conference*, 69–91.

Curtis, B. (1984). Fifteen years of psychology in software engineering: Individual differences and cognitive science. *Proceedings of the 7th international Conference on Software Engineering*, 97–106.

de Raadt, M., Hamilton, M., Lister, R., Tutty, J., Baker, B., Box, I., Cutts, Q., Fincher, S., Hamer, J., Haden, P., Petre, M., Robins, A., Simon, Sutton, K., Tolhurst, D. (2005). Approaches to learning in computer programming students and their effect on success. *Higher Education in a changing world: Research and Development in Higher Education*, 28, 407–414

Dehnadi, S. & Bornat, R. (2006). The camel has two humps. *Little PPIG 2006*, Retrieved 4th June, 2009, from <http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf>

Dehnadi, S. (2006). Abstract for Dehnadi & Bornat (2006). Retrieved 4th June, 2009, from <http://www.cs.mdx.ac.uk/research/PhDArea/saeed/>

Dickmann, R.A. & Lockwood, J. (1966). *Computer personnel research group, 1966 survey of test use in computer personnel selection*. Technical Memo, Johns Hopkins University Silver Spring MD Applied Physics Lab.

Evans, G.E. & Simkin, M.G. (1989). What best predicts computer proficiency?. *Communications of the ACM*, 32 (11), 1322–1327.

Feldman, D.H. (2004). Piaget's stages: the unfinished symphony of

cognitive development. *New Ideas in Psychology*, 22, 175–231.

Fincher, S., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Robins, A., Simon, Sutton, K., Tolhurst, D., Tutty, J. (2005b) *Programmed to succeed?: A multi-national, multiinstitutional study of introductory programming courses*. Computing Laboratory Technical Report 1-05, University of Kent, Canterbury, UK.

Fincher, S., Lister, R., Clear, T., Robins, A., Tenenberg, J. & Petre, M. (2005a) . Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs. *Proceedings of the First international Workshop on Computing Education Research (ICER '05)*, 111–121.

Forbes, J. & Garcia, D.D. (2007). “...But what do the top-rated schools do?”: a survey of introductory computer science curricula. *SIGCSE Bulletin*, 39 (1), 245 - 246.

See also resources retrieved 4th July, 2009, from <http://www.cs.duke.edu/csed/openwiki/doku.php?id=teaching:start>

Gardner, H. (1993). *Frames of mind: The theory of multiple intelligences*. NY: Basic Books.

Gaskins, I.W., Rauch, S., Gensemer, E., Cunicelli, E., O’Hara, C., Six, L., & Scott, T. (1997). Scaffolding the development of intelligence among children who are delayed in learning to read. In K. Hogan & M. Pressley (Eds.), *Scaffolding student learning: Instructional approaches and issues* (pp. 43-73). Cambridge, MA: Brookline.

Haskell, R.E (2001). *Transfer of learning: Cognition instruction and reasoning*. CA: Academic Press.

Hofstadter, D.R. (2001). Epilogue: Analogy as the core of cognition. In D. Gentner, K.J. Holyoak & B.N. Kokinov (Eds), *The analogical mind: Perspectives from cognitive science* (pp. 499–538). Cambridge MA: MIT Press.

Holyoak, K.J., Gentner, D. & Kokinov, B.N. (2001). Introduction: The place of analogy in cognition. In D. Gentner, K.J. Holyoak & B.N. Kokinov (Eds), *The analogical mind: Perspectives from cognitive science* (pp. 1–19). Cambridge MA: MIT Press.

Hu, C. (2004). Rethinking of Teaching Objects-First. *Education and Information Technologies*, 9 (3), 209–218.

Hudak, M.A. & Anderson D.E. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology*, 17(4), 231–234.

Jensen, A.R. (1998). *The g factor*. Westport, CT: Praeger.

Katz, A. (1962). Prediction of success in automatic data processing course. *U.S. Army Personnel Research Office Technical Note, No. 126*.

Kölling, M. (1999). The Problem of Teaching Object-Oriented Programming, Part 1: Languages. *Journal of Object Oriented Programming*, 11(8), 8–15.

Kolb, D.A. (1985). *Learning style inventory (revised edition)*. Boston, MA: McBer and Company.

Krause, L. (2000). *How we learn and why we don't*. Cincinnati, OH: Thomson Learning.

Kurtz, B.L. (1980). Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *SIGCSE Bulletin*, 12 (1), 110–117.

Leither, H.E. & Lewis, H.R. (1978). Why Johnny can't program: A progress report. *SIGCSE Bulletin*, 10 (1), 266–276.

Lister, R. (2000). On blooming first year programming, and its blooming assessment. *Proceedings of the Australasian Conference on Computing Education (Melbourne, Australia (ACSE '00))*, 158–162.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C. & Whalley, J. L. (2006). Research perspectives on the objects-early debate. *SIGCSE Bulletin* 38 (4), 146–165.

Lister, R. & Leaney, J. (2003). First year programming: Let all the flowers bloom. *Fifth Australasian Computing Education Conference (ACE 2003)*, 221–230.

Lorenzen, T. & Chang, H.L. (2006). MasterMind©: A Predictor of Computer Programming Aptitude. *INROADS - The SIGCSE Bulletin*, 38 (2), 69–70.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education (ITiCSE-WGR '01)*, 125–180.

McNaughton, S. & Leyland, J. (1990). The shifting focus of maternal tutoring across different difficulty levels on a problem solving task. *British Journal of Developmental Psychology*, 8, 147–155.

Manning, C. & Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Mayer, D.B. & Stalnaker, A.W. (1968). Selection and evaluation of computer personnel – the research history of SIG/CPR. *The Proceedings of the 1968 ACM National Conference (23rd ACM National Conference)*, 657–670.

Mazlack, L.J. (1980). Identifying potential to acquire programming skill. *Communications of the ACM*, 23 (1), 14–17.

- Meyer, J.H.F. & Land, R. (2003). Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising. In C. Rust (Ed.), *Improving Student Learning Theory and Practice – Ten Years On* (pp. 412–424). Oxford: OCSLD.
- Morrison, M. & Newman, T.S. (2001). A study of the impact of student background and preparedness on outcomes in CS I. *SIGCSE Bulletin*, 33 (1), 179–183.
- Myers, I.B. (1980/1995). *Gifts Differing: Understanding Personality Type*. Mountain View, CA: Davies-Black Publishing. (Original edition 1980; Reprint edition 1995).
- Newstead, P.R. (1975). Grade and ability predictions in an introductory programming course. *SIGCSE Bulletin*, 7, 87–91.
- Pea, R.D. & Kurland, D.M. (1984). *On the Cognitive Prerequisites of Learning Computer Programming*. Technical Report No.18, Bank Street College of Education, New York, NY.
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *SIGCSE Bulletin* 39 (4), 204–223.
- Piaget, J. (1971a). Developmental stages and developmental processes. In D.R. Green, M.P. Ford, & G.B. Flamer (Eds.), *Measurement and Piaget* (pp. 172–188). NY: McGraw-Hill.
- Piaget, J. (1971b). The theory of stages in cognitive development. In D.R. Green, M.P. Ford, & G.B. Flamer (Eds.), *Measurement and Piaget* (pp. 1–11). NY: McGraw-Hill.
- Pocius, K. E. (1991). Personality factors in human-computer interaction: A review of the literature. *Computers in Human Behavior*, 7(3), 103–135.

- Psychometrics I.T. Tests (2009). *Berger Aptitude for Programming Test B-APT Form D*. Retrieved July 3rd, 2009, from <http://www.psy-test.com/Baptd.html>
- Reder, L.M. (1982). Elaborations: When do they help and when do they hurt? *Text*, 2, 211–224.
- Rieber, R.W. & Robinson D.K. (Eds.) (2004) *The essential Vygotsky*. NY: Kluwer Academic / Plenum Publishers.
- Robins, A., Rountree, J. & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137 - 172.
- Robins, A. (1996) Transfer in cognition. *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research*, 8, 185–203.
- Rountree, J. & Rountree, N. (2009). Issues regarding threshold concepts in computer science. *Proceedings of the Eleventh Australasian Computing Education Conference (ACE 2009)*, 139–145.
- Rountree, N., Rountree, J. & Robins, A. (2002). Predictors of success and failure in a CS1 course. *SIGCSE Bulletin*, 34 (4), 121–124.
- Rountree, N., Rountree, J., Robins, A. & Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. *SIGCSE Bulletin*, 36 (4), 101–104.
- Scholtz, J. & Wiedenbeck, S. (1993). An analysis of novice programmers learning a second language. *Proceedings of the Fifth Workshop on Empirical Studies of Programmers*, 187–205.
- Scott, T. (2003). Bloom's taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, 19 (1), 267–274.

Simon, Cutts, Q., Fincher, S., Haden, P., Robins, A., Sutton, K., Baker, B., Box, I., de Raadt, M., Hamer, J., Hamilton, M., Lister, R., Petre, M., Tolhurst, D. & Tutty, J. (2006b). The ability to articulate strategy as a predictor of programming skill.

Proceedings of the 8th Australasian Conference on Computing Education - Volume 52, 181–188.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D. & Tutty, J.

(2006a). Predictors of success in a first programming course. *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, 189–196.

Simpson, D. (1973). Psychological testing in computing staff selection: a bibliography. *ACM SIGCPR Computer Personnel*, 4 (1-2), 2–5

Soh, L.K., Samal, A., Person, S., Nugent, G. & Lang, J. (2005). Designing, implementing, and analyzing a placement test for introductory CS courses. *ACM SIGCSE Bulletin*, 37 (1), 505–509.

Sternberg, R.J. (Ed.) (1982). *Handbook of Human Intelligence*. NY: Cambridge University Press.

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, 4, 295–312,

Subramanian, A. & Joshi, K. (1996). Computer aptitude tests as predictors of novice computer programmer performance. *Journal of Information Technology Management, Volume VII (1 & 2)*, 31–41.

Tolhurst, D., Baker, B., Hamer, J., Box, I., Lister, R., Cutts, Q., Petre, M., de Raadt, M., Robins, A., Fincher, S., Simon, Haden, P., Sutton, K., Hamilton, M. & Tutty, J. (2006). Do map drawing styles of novice programmers predict success in programming?: a multi-national, multi-institutional study. *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, 213–222.

- Tukiainen, M. & Mönkkönen, E. (2002). Programming aptitude testing as a prediction of learning to program. *Proceedings of PPIG, 14*, 45–57.
- Ventura Jr., P.R. (2005). Identifying predictors of success for an objects-first CS1. *Computer Science Education, 15(3)*, 223–243.
- Vosniadou, S. & Ortony, A. (1989). Similarity and Analogical Reasoning: A Synthesis. In S. Vosniadou & A. Ortony (Eds.) *Similarity and Analogical Reasoning* (pp. 1–17). NY: Cambridge University Press.
- Voss, J.F. (1977). Cognition and instruction: Towards a cognitive theory of learning. In A.M. Lesgold, J.W. Pellegrino, S.D. Fokkema & R. Glasser (Eds.) *Cognitive Psychology and Instruction* (pp. 13–26). NY: Plenum Press.
- Werth, L. H. (1986). Predicting student performance in a beginning computer science class. *SIGCSE Bulletin, 18 (1)*, 138–143.
- White, G. & Sivitanides, M. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education, 13(1)*, 59–66.
- Whitley Jr., B.E. (1996). The relationship of psychological type to computer aptitude, attitudes, and behavior. *Computers in Human Behavior, 12 (3)*, 389–406.
- Whittington, K.J., Bills, D.P & Lawrence, W.H. (2003). Implementation of alternative pacing in an introductory programming sequence. *Proceedings of the 4th conference on Information technology, 47–53*.
- Wiedenbeck, S. (2005). Factors affecting the success of non-majors in learning to program. *Proceedings of the First international Workshop on Computing Education Research (ICER '05)*, 13–24.

Wileman, S.A., Konvalina, J. & Stephens, L.J. (1981). Factors influencing success in beginning computer science courses. *Journal of Educational Research*, 74, 223–226

Wilson, B.C. & Shrock, S. (2006). Contributing to success in an introductory computer science course: a study of twelve factors. *ACM SIGCSE Bulletin*, 33 (1), 184–188

Wolfe, J.M. (1969). Testing for programming aptitude. *Datamation*, April 1969, 67–72.

Woszczyński, A.B., Guthrie, T.C., Chen, T. & Shade, S. (2004). Personality as a predictor of student success in programming principles I. *7th Annual Southern Association for Information Systems (SAIS)*, 1–6

Woszczyński, A., Haddad, H., & Zgambo, A. (2005a). An IS student's worst nightmare: Programming courses. *8th Annual Southern Association for Information Systems (SAIS)*, 130–133.

Woszczyński, A., Haddad, H., & Zgambo, A. (2005b) Towards a model of student success in programming courses. *Proceedings of the 43rd annual Southeast regional conference - Volume 1 (ACM-SE 43)*, 301–302.

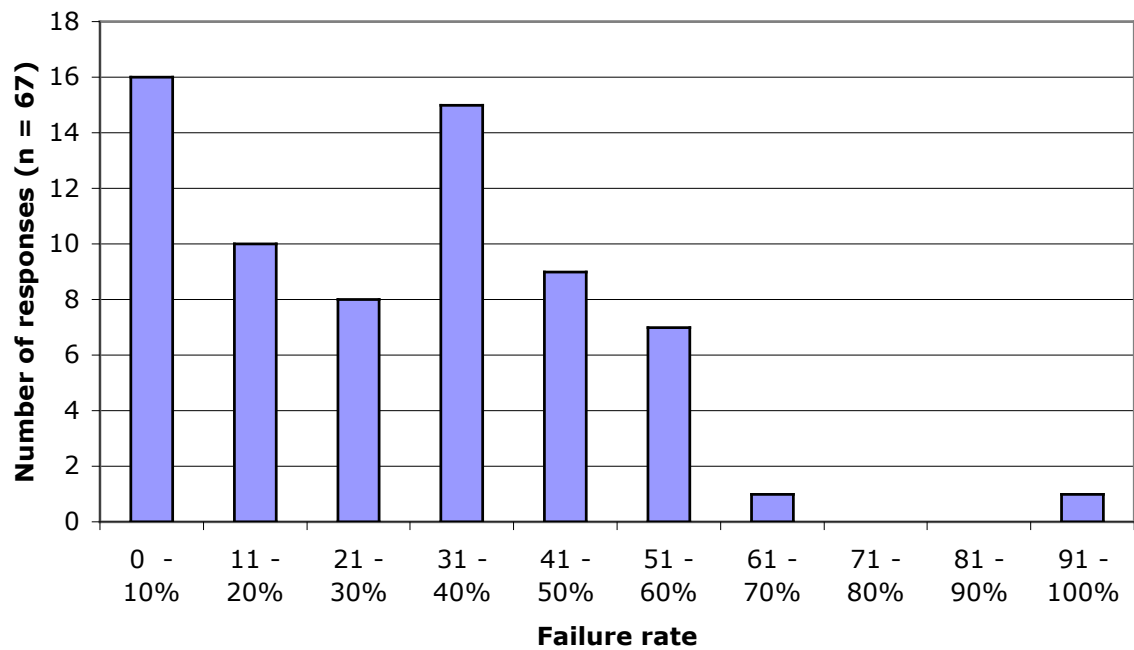


Figure 1: Failure rates in CS1 courses from 67 institutions as reported in Bennedsen & Caspersen (2007). This data aggregates different kinds of failure, such as dropping out of the course, or completing but not achieving a passing grade.

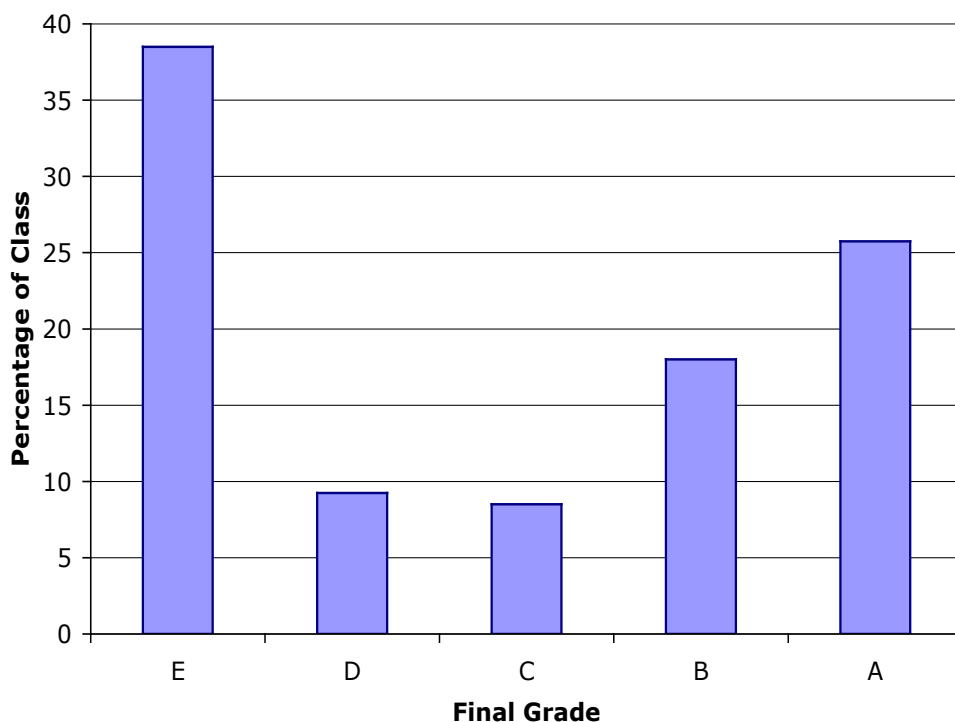


Figure 2: Final grade distribution in an example CS1 course (average of four years of data 2005 to 2008). E and D are fail grades. Percentage scores 0 - 39% correspond to the letter grade E, 40 - 49% to D, 50 - 64% to C, 65 - 79% to B and 80 - 100% to A.

(a)

```
Random rand = new Random();
int chunkMax = 10;
int [ ] scoreFrequency = new int[chunkMax + 1];

for (int run = 0; run < 10000; run++) {
    double threshold = 0.5;
    int score = 0;

    for (int chunk = 0; chunk < chunkMax; chunk++) {
        if (rand.nextDouble() > threshold) {
            score++;
        }
    }
    scoreFrequency[score]++;
}
}
```

(b)

```
for (10000 repetitions to generate 10000 scores) {
    threshold = 0.5
    score = 0

    for (10 repetitions representing "chunks") {
        if (random_number_0-1 > threshold) {
            add one to score
        }
    }
    add one to frequency of score
}
}
```

Figure 3: Version M0 of the model with various example parameter values. Part (a) is a description in Java code. The variables declared above the loops are called the “set up variables”. The contents of the inner loop are referred to as the “kernel” of the model. Part (b) is a description in pseudocode for the same parameters. Further versions of the model (some of which require threshold to be initialised within the outer loop as shown) can be created as variations of M0 as described in the text.

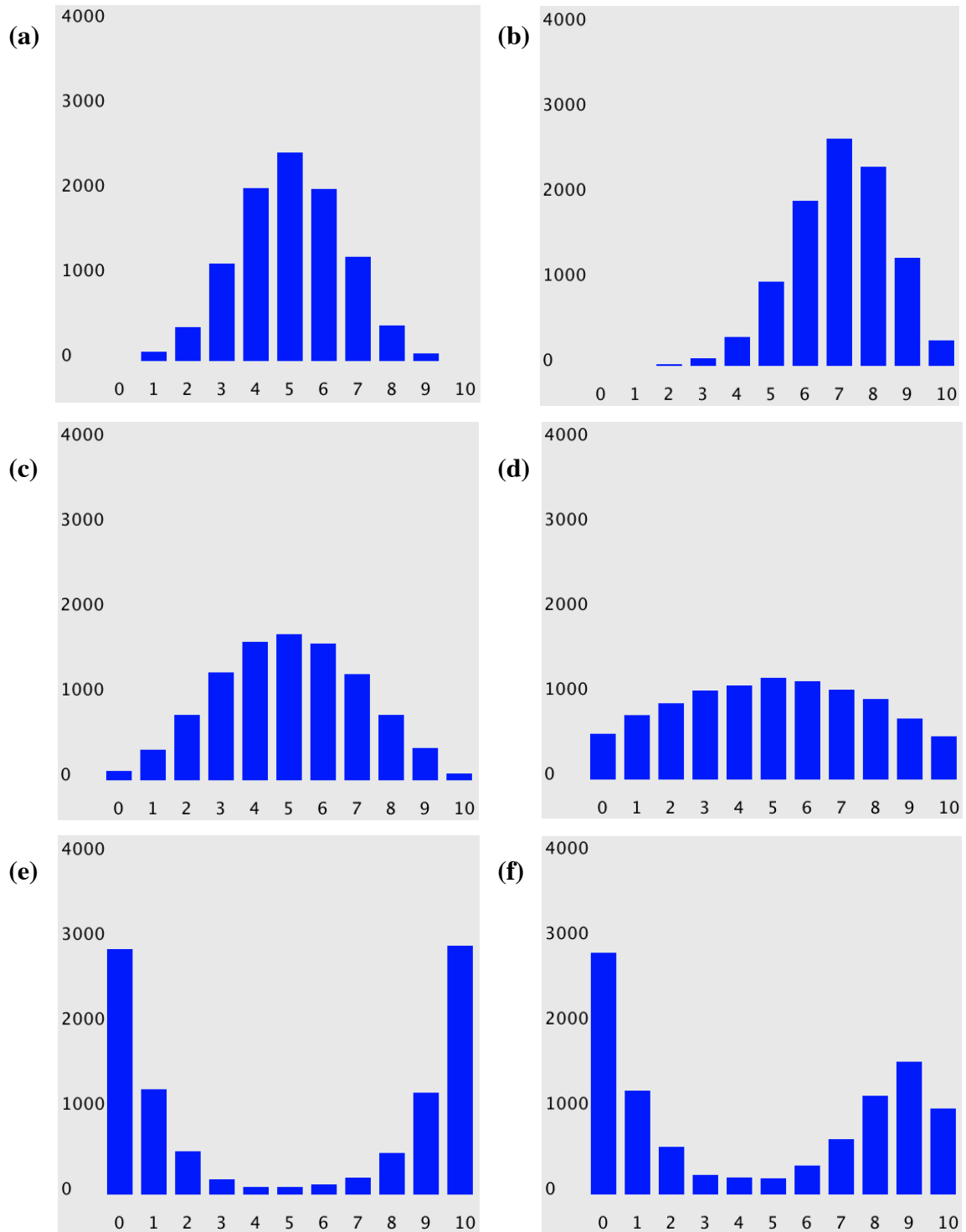


Figure 4: Distributions created by the model with various settings. In each case the x axis represents the score, and the y axis represents the frequency of that score over 10,000 runs. (a) M0 with threshold = 0.5, (b) M0 with threshold = 0.3, (c) M1 with threshold = 0.5 and offset = 0.03, (d) M1 with threshold = 0.5 and offset = 0.06, (e) M1 with threshold = 0.5 and offset = 0.18, (f) M1 with threshold = 0.5 and bounded offset function as described in the text, and M2 as described.

```

// set up variables
Random rand = new Random();
int chunkMax = 10;
int [ ] scoreFrequency = new int[chunkMax + 1];
double threshold = 0.5;
double offset = 0.18;
double upperBound = 0.36;

// population loop
for (int run = 0; run < 10000; run++) {
    // run variables
    int score = 0;
    double momentum = 0.0;

    // individual score loop
    for (int chunk = 0; chunk < chunkMax; chunk++) {
        // "kernel" of the model
        if (rand.nextDouble() + momentum > threshold) {
            // success condition
            score++;
            momentum += offset;
            if (momentum > upperBound) momentum = upperBound;
        } else {
            // fail condition
            momentum -= offset;
        }
    }
    scoreFrequency[score]++;
}
}

```

Figure 5: Version M2 of the model (in Java code) with various example parameter values and a success condition offset function that implements an upper bound on momentum. This code generates the distribution shown in Figure 4(f).